2023 IEEE Symposium Series on Computational Intelligence (SSCI)
Mexico City, Mexico. December 5-8, 2023

A Task Scheduler for Mobile Edge Computing Using Priority-based Reinforcement Learning

Amin Avan
Dept. of Electrical, Computer and Software Engineering
Ontario Tech University
Oshawa, ON, L1G 0C5 Canada
amin.avan@ontariotechu.net

Farnaz Kheiri
Dept. of Electrical, Computer and Software Engineering
Ontario Tech University
Oshawa, ON, L1G 0C5 Canada
farnaz.kheiri@ontariotechu.net

Qusay H. Mahmoud
Dept. of Electrical, Computer and Software Engineering
Ontario Tech University
Oshawa, ON, L1G 0C5 Canada
qusay.mahmoud@ontariotechu.ca

Akramul Azim
Dept. of Electrical, Computer and Software Engineering
Ontario Tech University
Oshawa, ON, L1G 0C5 Canada
akramul.azim@ontariotechu.ca

Masoud Makrehchi
Dept. of Electrical, Computer and Software Engineering
Ontario Tech University
Oshawa, ON, L1G 0C5 Canada
masoud.makrehchi@ontariotechu.ca

Shahryar Rahnamayan
Dept. of Engineering
Brock University
St. Catharines, ON, L2S 3A1 Canada
srahnamayan@brocku.ca

***Abstract*—Edge computing offers cloud-like services closer to users and IoT devices, providing high speed and accessibility for network users. Edge computing, often called Mobile Edge Computing (MEC), is a distributed paradigm that utilizes heterogeneous computational and storage resources with well-provisioned capabilities rather than relying on the ample resources of the cloud. In addition, edge users usually refer to portable and mobile devices that connect to and disconnect from the network at will. Therefore, scheduling tasks at the appropriate time and allocating the right resources can be modeled as a multi-objective optimization problem in MEC. Moreover, each task has specific requirements, further adding to the complexity of the optimization problem. In this study, we formulate the scheduling problem as a Markov Decision Process (MDP) to schedule the tasks. The learning time of the task scheduler is minimized when it faces new users and edge servers. Subsequently, we employ the Q-learning (QL) algorithm from the Reinforcement Learning (RL) paradigm to address the optimization problem and effectively adapt the proposed scheduler to the dynamic nature of MEC. Accordingly, we designed the valid state space, action space, and reward function with appropriate conditions and proper rewards for the proposed QL-based technique. We conducted comprehensive experiments to validate the results of the proposed solution, taking into account the inherent randomness of the QL-based technique. The experimental results demonstrate that the proposed technique achieves the lowest learning time compared to Deep learning-based and Deep RL-based approaches. Furthermore, on average, the proposed technique obtains a 72% faster runtime compared to previous works, using 58% fewer computation cycles and 50% less memory. These improvements make the proposed approach an efficient and lightweight task scheduler for MEC.**

***Index Terms*—Edge Computing, Mobile Edge Computing, Machine Learning, Reinforcement Learning, Task Scheduling, Markov Decision Process.**

I. Introduction

The quick progress in data processing and transmission technologies has enabled the development and deployment of mobile and Internet of Things (IoT) applications with bandwidth-intensive, computational-intensive, data-intensive (BCD-intensive), and strict Quality of Service (QoS) requirements [1]. The applications are implemented in diverse domains, such as entertainment, healthcare, vehicular, satellite, and industrial sectors, consisting of various devices, including sensors, actuators, smartphones, and tablets [2]. Despite the intensive demands of the applications, the devices that implement them are limited by constrained hardware resources, including processing power, memory capacity, and battery capacity [3]–[5].

The reasonable response to the limitations of device resources is to offload tasks to servers that have sufficient resources; thus, the initial solution was to utilize cloud servers. While cloud computing provides abundant computation power and memory capacity, the collaboration between devices and cloud can lead to network congestion and high transmission delays. This is primarily due to the fact that modern IoT and mobile applications generate an equal or even greater amount of data compared to what they consume [6]. In addition, Cisco has forecasted that the number of connected IoT devices will surpass 100 billion by 2030 [7]. As a result, data communication is expected to become a significant concern for networks. Furthermore, delays and jitters in Wide Area Networks (WANs) pose significant obstacles to the transmission of data for IoT and mobile applications. In fact, controlling delays in WANs is a significant challenge [8]. Finally, specific applications demand cost-effective services that may exceed affordability when utilizing cloud computing.

Moreover, data security is the primary concern in healthcare and financial applications. Device-cloud collaboration can potentially lead to data exposure because the cloud is a public and long-distance service [9]. Mobile edge computing (MEC), in contrast, offers computational and memory facilities in proximity to users and devices. Consequently, data-sensitive applications can be processed by local edge servers instead of being offloaded to a public or long-distance cloud servers. In addition to the aforementioned data-sensitive applications, there are also time-sensitive applications, such as autonomous vehicles, that rely on consistent response times and real-time processing. However, cloud servers are located at a lengthy distance away from autonomous vehicles, which can result in latency in their data processing [10]. Therefore, a rational solution would involve leveraging road-side units (RSU) as edge servers to enable MEC and provide processing resources in close proximity to autonomous vehicles. Regarding bandwidth-intensive applications and systems, such as artificial intelligence-powered surveillance cameras, the application might experience adverse impacts due to the increased time needed to transmit data to the cloud [11]. Nevertheless, MEC provides processing and memory services near end-user and IoT devices. Accordingly, leveraging MEC can potentially decrease data transmission overhead in the network, thereby improving users' QoS and Quality of Experience (QoE) [12]. Hence, MEC allows users to run BCD-intensive applications with stringent computation, storage, latency, and security requirements [13]. Finally, MEC offers secure processing of sensitive data, facilitates reliable response times, improves accessibility to remote applications in geographically distant locations, and minimizes internet bandwidth usage.

MEC is delivered to end-users and IoT devices through various communication infrastructures, such as Base Stations (BS), Rpi, Macro-cell Stations (MS), Access Points (AP), Femtocells, and RSUs [10]. Whilst the cloud predominantly comprises homogeneous computation and storage elements, the edge servers are characterized by their heterogeneous processing units, such as Central Processing Unit (CPU), Graphics Processing Unit (GPU), Digital Signal Processor (DSP), and Field Programmable Gate Arrays (FPGA) [14]. Additionally, MEC involves resource units that are geographically distributed across the network instead of the cloud's centralized resources. Furthermore, since the users of MEC may move over the networks, the task scheduler should consider the user's mobility in its scheduling procedure. Thus, task scheduling in MEC is defined as a multi-objective optimization problem that is NP-hard [10]. As an optimization problem, there are various methods for modeling it, including Mixed Integer Programming (MIP), Integer Linear Programming (ILP), Mixed Integer Linear Programming (MILP), Mixed Integer Non-Linear Programming (MINLP), Lyapunov, Markov Decision Process (MDP), and Game Theory [10], [14]. Consequently, the task scheduler technique aims to maximize the optimization goal defined in the problem model. Finally, the study makes the following contributions:

- This paper clarifies the three-layer MEC network architecture, explains reinforcement learning, and comprehensively reviews task scheduling techniques in MEC.
- A task scheduling technique based on Q-learning is proposed to address the optimization problem formulated by MDP. The number of default state spaces in the Q-learning technique has been reduced to expedite the proposed task scheduler and minimize learning time, computation cycles, and memory usage.
- The proposed task scheduling technique is evaluated alongside previous works and is assessed based on different criteria, such as scheduling processing time, required computation cycles, and memory usage.

The rest of this paper is structured as follows. Section 2 provides an overview of current task scheduling techniques in MEC, while Section 3 explains the proposed solution. The experimental results of the proposed and previous works are presented in Section 4. Finally, Section 5 concludes the study and highlights the research opportunity for future work.

II. Background Review

Network computing is comprised of three major layers, including cloud, edge, and users, which is depicted in Fig. 1 [13]. The cloud layer possesses ample computation and storage resources; thus, the cloud is a reliable solution for enterprise and large-scale applications. The geographical distance between users and the cloud leads to data transmission overheads and delays. As a result, the cloud requires assistance to fulfill the QoE and QoS requirements of BCD-intensive applications in the network.

The edge layer comprises geographically distributed communication infrastructures and processing resources in the three-layer MEC network architecture. The edge is positioned between the cloud and users, serving as a connection point that helps reduce data transmission overhead by providing computation and storage resources for users. Moreover, the user layer contains diverse users with different software, hardware, and requirements. Although each user or device possesses local computation and storage resources, these resources are limited and dependent on constrained battery capacity and power availability. Accordingly, offloading tasks for processing to either the edge or cloud server could be a reasonable and practical solution for meeting the QoS and QoE requirements of users.

Reinforcement Learning (RL) is a machine learning (ML) algorithm in which an agent learns how to act in a given environment by receiving rewards or punishments as signals for positive or negative actions. The ultimate goal of the agent is to maximize the cumulative reward over time [15]. The agent learns which action should be taken next time, depending on the rewards or punishments it receives. Each RL algorithm consists of several fundamental components, as outlined below:

- Input: The input should represent the starting point for the model.

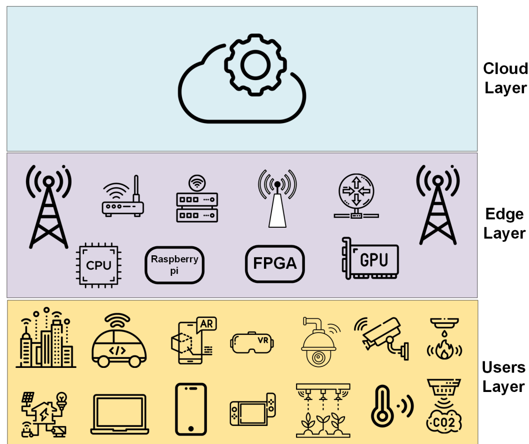

Fig. 1: Three-layer MEC network architecture. The Users layer consists of IoT devices with restricted resources, including computation, memory, and power capacity. The Edge layer comprises diverse edge servers located near the IoT devices, but they have limited resources compared to Cloud. The Cloud offers ample resources despite being located far from IoT devices.

- Output: There are a variety of solutions to a particular problem due to multiple existing possible solutions.
- The output of the defined model is a state resulting from training, and the user decides to provide a negative or positive reward to the model derived from the output.
- The model contains training, and then the best solution is decided based on the maximum reward.
- The model contains training, and then the maximum reward determines the excellent solution.

III. Related Work

The methods from related works are categorized into three subsections: heuristic algorithms, meta-heuristic algorithms, and AI-based methods.

A. Task Scheduling Using Heuristic Algorithms

A heuristic ranking-based task scheduler named COFE is introduced in [16], which considers task dependency for executing the tasks in the workloads and applications. COFE offloads the tasks on both Edge servers and Cloud. COFE's task scheduling algorithm aims to minimize the average makespan.

In [17], Task Continuation Affinity (TCA) is a heuristic-based task scheduler aiming to arrange each task in close proximity to its immediate predecessor. Furthermore, Most Powerful CPU (MPCPU) is another algorithm heuristic that assigns tasks to the processor with the most computing power.

Wu et al. [18] scheduled the tasks of vehicles over MEC using a combination of the greedy algorithm and tabu search algorithm. Indeed, minimizing the response time is the goal of the optimization problem in [18]. Moreover, the tasks and Edge servers modeled as Poisson stream and the M/M/1 queuing system, respectively.

The authors of [19] proposed a method to schedule tasks by considering the tasks' dependencies in a Directed Acyclic Graph (DAG). Therefore, they proposed a heuristic scheduling algorithm anointed CaGTS, which aims to minimize the delay of task response. In addition, in [19], a fault-tolerant task scheduling algorithm called DaTR was designed to reschedule the tasks when an edge server fails.

Liu et al. [20] proposed a priority-aware scheduling algorithm by taking into account tasks' dependencies in vehicular edge computing (VEC). The objective of their technique is to minimize the average completion time of the tasks, and they employed RSU as edge servers and offloaded the tasks of vehicles to them.

B. Task Scheduling Using Meta-Heuristic Algorithms

The proposed multi-objective optimization model [21], called OWPSO, which stands for Opposition-Based Whale Optimization Algorithm with Particle Swarm Optimization to model NP-hard cloud task scheduling problem.

Saravanan et al. [22] presented an Improved Wild Horse Optimization (IWHO) to address the long scheduling time, costly consumption, and the application with high QoS and QoE requirements in cloud computing task scheduling.

Alboaneen et al. [23] addressed joint task scheduling and virtual machine placement (JTSVMP) problems in cloud computing networks. With regard to capacity restrictions in cloud data centers, the presented co-optimization process allocates the chosen virtual machines to the most heavily used physical host while scheduling jobs with the lowest execution costs.

C. Task Scheduling Using AI-based Algorithms

Gao et al. [24] proposed a deep reinforcement learning (DRL) based approach for the task scheduling problem in mobile blockchain networks for IoT applications installed at Small-cell Base Station (SBS). The objective is to maximize SBS's long-term mining reward while reducing resource costs.

Tang et al. [25] proposed an AI-based method, a container-based task scheduling method in cloud-edge computing environments. The suggested methodology utilizes a multi-criteria method and a priority-based greedy outline to specify the best container to implement a task based on response time, power consumption, and cost of executing the task.

Zhan et al. [26] used the concept of DRL in a VEC scheduling problem for compute offloading. The presented model is based on a Markov decision process and the proximal policy optimization algorithm to optimize the scheduling of tasks.

In [27], a distributed deep learning-based scheduling algorithm is proposed, called DDLO, to optimize energy conservation and QoS. The optimization problem is formulated as a MIP problem. Since the dimension of the considered problem is high, the authors of [27] distribute the deep learning-based solution. Consequently, several parallel Deep Neural Networks (DDN) are utilized to solve the decision-making of binary offloading in MEC. In addition to parallel DDNs, a shared

memory is implemented to accelerate the training phase of DNNs.

Huang et al. [28] suggested a scheduling technique founded upon a Deep Reinforcement Learning algorithm named DROO to schedule the tasks in MEC. The total goal is to improve the data processing capacity of low-power networks like the IoT. The proposed method uses a binary offloading procedure to solve combinatorial optimization problems by immediately adjusting decision-making about offloading the tasks to the suitable edge server and allocating the wireless resources regarding the dynamic wireless channel conditions over time. It reduces the need to solve combinatorial optimization problems. Then, it could decrease the computational complexity, especially in high-demanding networks.

IV. Proposed Task Scheduling Model

Q-learning (QL) is a commonly utilized model-free RL algorithm that employs a table called the Q-table. This table is gradually updated during the learning procedure to store estimated future rewards when transitioning from a "current state" to the next state using a selected action [29]. Considering the complexity of task scheduling in MEC, which encompasses numerous potential server allocation solutions, we propose a method for task scheduling based on QL. The QL-based approach leverages the capabilities of QL to handle complex and dynamic environments effectively. Inspired by the earliest deadline first (EDF) scheduling algorithm for real-time systems, we adopt the rule of prioritizing tasks with the closest deadline as high-priority in the proposed task scheduler.

MDP is a mathematical framework used to simulate decision-making scenarios where the outcome of an action depends on both the agent's state at the time of the action and its current state. QL models can effectively address various decision-making problems, especially those characterized by extensive state and action spaces. Therefore, the proposed problem is formulated as a MDP due to the numerous potential solutions involved in task scheduling.

A. Proposed priority-based Q-learning (QL)

1) Define the State Space by Priority Consideration: The problem is defined with a set of n tasks, represented by $\{T_1, T_2, ..., T_n\}$, which can be requested by a group of m users $\{U_1, U_2, ..., U_m\}$ to be executed on one or more different edge servers. The edge server will be allocated from a pool of available servers, denoted as $\{S_1, S_2, ..., S_k\}$, ensuring that the selected server can fulfill the operational requirements of each task.

As a preliminary step, we must generate the state spaces of the QL-based method, composed of potential permutations of binary offloading decisions. Accordingly, each state is represented by a binary matrix with m rows and k columns; each cell of the decision matrix shows which server is allocated to a requested task. As shown in Eq. (1), this is a decision matrix, T_1 and T_3 are allocated to S_2 and S_1, respectively.

$$\text{Definition 1}: \quad \begin{array}{c} \\ T_1 \\ T_2 \\ T_3 \\ T_4 \end{array} \begin{array}{c} \begin{array}{cc} S_1 & S_2 \end{array} \\ \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 0 & 0 \end{bmatrix} \end{array} \tag{1}$$

2) Actions – Choosing Next State: The agent takes an action based on a randomly generated number at each step. If the generated random number is below "0.9", the agent selects the state with the highest reward from the Q-table. Nonetheless, if the generated random number is higher than "0.9", the agent randomly selects the next state. The Q-table is initialized with "zero" values at the beginning of the learning process.

3) Estimating Reward Value: After selecting the next state at each learning round, the reward value would be updated based on the conditions below:

- In the stage where the "next state" causes the edge servers to transition into an "idle" state, the agent receives the highest negative reward, as depicted in Eq. (2). Consequently, the episode concludes with a "GAME OVER" message.

$$\text{new_reward} = \text{total_reward} - ((k \times (n \times k))^n) \tag{2}$$

- If the agent assigns task T_i to server S_i or task T_j to server S_j in the current state and reassigns them to server S_j and S_i, respectively, a negative reward is incurred. This transition involves the transmission of tasks from one server to another, resulting in data transmission, bandwidth overhead, preempting task execution, and additional caching and queuing processes.

$$\text{new_reward} = \text{total_reward} - (k \times (n \times k)) \tag{3}$$

- If the next state is equal to the current state while there exists "execution time" for the assigned tasks, then the agent receives the positive reward.

$$\text{new_reward} = \text{total_reward} + (k \times (n \times k)) \tag{4}$$

- As much as the servers are allocated to the tasks, it gets more positive rewards.

$$\text{new_reward} = \text{total_reward} + (k \times (n \times k)) \tag{5}$$

In the following, Q-table updates regarding "current state", "next_state", and new reward value using the equation Eq. (6):

$$\begin{aligned} &Q_table[S_c, S_n] += \\ &\alpha \times (r + \gamma \times \max(q_table[S_n]) - q_table[S_c, S_n])) \end{aligned} \tag{6}$$

where:

- "$Q_table[S_c, S_n]$" is the updated reward value upon reaching the next state, denoted as S_c, through the transition from the current state, represented by S_c.
- "α" is the learning rate that specifies the weight assigned to the new reward in contrast to the previous reward.

- "r" is the reward received after moving to the chosen next state.
- "γ" is the discount factor, which regulates how much importance is placed on future rewards in comparison to present rewards.

4) Proposed Accelerating for QL-based Task Scheduling: The abundance of state spaces can decrease the speed of the learning process or require substantial memory allocation when utilizing the QL-based method for solving large-scale problems. This is because QL is one of the RL algorithms where the number of state spaces significantly increases as the number of variables rises. Since a state space is represented by a matrix, as the variables of the RL algorithm increase, the size of the matrix also grows. Furthermore, as the number of variables in the RL algorithm increases, the abundance of state spaces will also increase because the number of state spaces equals the possible variations of the variables.

For example, the number of all possible state spaces in with four users and one server is $16 = (2^{1\times4})$, and a 1×4 matrix represents each state space. Moreover, a network with four users and two servers would have $256 = (2^{2\times4})$ state spaces, each of which is a 2×4 matrix. Consequently, the learning process would be faster with 16 state spaces rather than 256 state spaces. In addition to the number of state spaces, the state space's size also impacts the learning process's pace. Therefore, eliminating the invalid, repeated, or any state spaces that can be deleted would accelerate the QL-based algorithm.

As displayed in Fig. 2, we are implementing the following steps to accelerate the proposed task scheduling technique:

- We eliminate all the state spaces that only offload tasks to a single server when we have two or more servers in the edge network. Additionally, this elimination is beneficial for load balancing, as the proposed method aims to utilize all the available servers in the edge network instead of leaving one or more servers idle while overloading the others.
- We eliminate the state spaces when a task is partitioned and offloaded to two or more servers because this would prevent the other tasks from being served, as a single task occupies all the servers.
- In the interest of both acceleration and prioritization, we only consider m (i.e., m refers to the number of edge users in a network) tasks with the earliest deadlines among all tasks at each step of scheduling. Therefore, the proposed technique does not consider all tasks in gigantic matrices; instead, it divides the tasks and then constructs the matrices and state spaces. For instance, we consider a network with two servers and five users, each of whom has five tasks. Instead of representing the state spaces with 25×2 matrices, our proposed technique considers 5×2 matrices. At each step of scheduling, our technique only takes into account the five earliest deadline tasks. Consequently, computing and handling the 5×2 matrices is more straightforward than dealing with the 25×2 matrices.

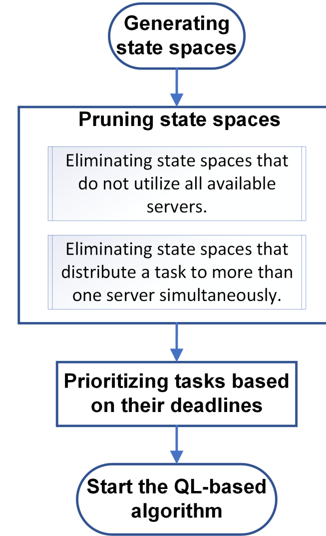

Fig. 2: Workflow of the proposed technique's acceleration steps.

B. Architecture

Although the MEC network architecture consists of cloud, edge, and user layers, as illustrated in Fig. 1, the proposed scheduling technique, called the priority-based QL technique (PBQ), takes into account user-edge collaboration. PBQ schedules the tasks of users based on the number of available edge servers. Hence, PBQ can incorporate new edge servers into its scheduling procedure parameters as soon as they join the network or become available for serving. In contrast, previous works can only handle a single edge server, such as DDLO [27] and DROO [28]. In addition, the task schedulers in MEC offload the tasks utilizing two primary methods: binary and partial. These methods are employed for scheduling tasks on servers within MEC. In binary, the user offloads entire tasks to the edge server; conversely, the user intends to offload some of its tasks to the edge server and execute the remaining tasks using local resources [10]. PBQ is designed and implemented to schedule the tasks in a binary manner; thus, all the users' tasks must be executed on Edge servers.

V. Evaluation

A. Dataset

To the best of our knowledge and regarding the state-of-the-art, there is no widely-used dataset and a standard benchmark for evaluating task scheduling algorithms in MEC. Consequently, we produce a dataset comprising the indexes of users, the execution time (c_i) that each task needs to be completed, and the deadline (d_i) for each task. Indeed, the amount of required process time by each task (τ_i) is called *task utilization* (u_i) and is obtained by Eq. (7).

$$u_i = \frac{c_i}{d_i} \tag{7}$$

A set of tasks is feasible to be executed entirely if the sum of all tasks' utilization becomes lower or equal to the number of servers, which is indicated by Eq. (8) [30].

$$U = \sum_{i=0}^{n} u_i \tag{8}$$

$$U = \begin{cases} (U \leq k) & \text{feasible} \\ (U > k) & \text{infeasible} \end{cases}$$

Accordingly, we adhere to the feasibility rule in producing the tasks' parameters in our dataset. As a result, all of our datasets for the 24, 48, and 96 tasks are entirely feasible.

B. Implementation

In this section, The effectiveness of the proposed model is assessed and compared to two other task scheduling methods, DDLO [27] and DROO [28], in MEC networks. We considered the same implementation conditions to compare algorithms in a fare setting. The implementation environment was Google Colab, a virtual machine environment to run our Python codes. The allocated memory was 12 GB, and an Intel Xeon CPU with 2.20 GHz frequency.

TABLE I: Duration of PBQ Learning Process for 31 Independent Scheduling Rounds. The PBQ is a QL-based technique that learns by making stochastic-based decisions. Statistical analyses demonstrate that the 31 independent runs would be sufficient to ensure reliable results.

Execution Round	Scheduling Process (*seconds*)					
	24 Tasks		48 Tasks		96 Tasks	
	1 Server	2 Servers	1 Server	2 Servers	1 Server	2 Servers
1	772.18	112.92	2403.84	166.74	1619.02	315.22
2	987.37	106.71	3593.38	195.7	2722.01	266.3
3	1120.61	81.03	2201.19	147.36	2579.61	382.46
4	1766.8	84.15	3361.45	131.07	2786.19	307.81
5	1163.95	89.07	1368.31	131.16	2992.64	213.79
6	356.3	112.34	1182.795	100.59	3930.66	330.26
7	1517.84	91	1973.44	158.53	4176.3	340.65
8	1471.22	81.09	2945.48	190.82	4240.91	399.39
9	1856.75	82.09	1871.2	101.43	3598.52	293.36
10	814.13	53.83	2259.39	130.61	3925.08	259.06
11	804.41	123.34	1777.69	133.01	3430.33	443.26
12	779.43	78.5	2419.53	134.94	3227.23	384.8
13	966.02	60.7	2000.62	156.05	3998.82	256.86
14	405.93	65.86	1343.62	216.04	2374.61	345.07
15	445.57	83.16	3263.64	164.35	1993.46	330.57
16	733.9	78.11	1543.78	142.07	3185.83	294.03
17	949.72	126.94	3260.79	159.75	3490.27	385.54
18	1539.35	49.3	2335.29	153.77	4046.78	317.71
19	1393.58	76.29	1913.99	157.35	2130.13	212.71
20	1652.37	131.05	2655.29	225.41	2944.2	328.24
21	940.96	61.23	904.67	132.55	3674.31	313.76
22	985.43	76.56	2778.32	183.84	2643.03	373.93
23	730.16	46.4	1399.93	203.74	3536.76	295.04
24	1710.87	29.47	1226.39	156.89	4060.69	288.94
25	1058.81	70.4	583.08	220.35	2812.25	309.62
26	855.31	64.41	1329.21	223.69	2159	232.06
27	1259.54	122.27	1363.47	178.16	3216.87	300.8
28	1499.83	50.62	2089.51	121.52	2575.58	397.74
29	440.34	86.61	2466.39	120.74	3137.97	308.76
30	1076.21	92.05	2578.18	140.29	3289.98	308.76
31	630.9	63.37	1263.34	169.42	2711.56	280.22
Average	**1054.38**	**81.64**	**2053.45**	**159.61**	**3186.38**	**316.67**
Standard Deviation	**421.69**	**25.43**	**772.05**	**34.58**	**648.11**	**54.91**

C. Validation of Results

Since the proposed model selects the next resource allocation states stochastically, we run our model independently 31 times [31] to verify the validity of our results. Table I presents the results of all 31 independent rounds of task scheduling for 24, 48, and 96 tasks. The *average* and *standard deviation* of 31 different scheduling independent rounds is calculated for all number of tasks. Indeed, the *standard deviation* is calculated to indicate the distribution of the results around the average values.

According to Table I, PBQ is a scalable task scheduler capable of handling any number of edge servers. This feature gives PBQ a distinct advantage in load balancing compared to other task scheduler techniques. Moreover, these advantages contribute to improved algorithm's runtime, computation cost, and memory utilization compared to previous works.

The significant difference in learning time between PBQ with one server and two servers lies in the size of the state space matrix and the number of tasks executed in each step. In particular, the scheduler executes the task of only one user out of six at each step in a network with a single server. However, employing two servers allows PBQ to schedule two users in each step, thereby speeding up the execution of users' tasks and reducing the learning procedure time. Nevertheless, there is a balance between the number of servers added and the speed of the learning process. Therefore, if additional servers are added beyond a certain threshold, the network will experience an increase in the duration of the learning process instead of accelerating it. Moreover, as the network adds more servers, it will encounter general issues such as communication overhead, synchronization overhead, and load balancing. These challenges are commonly faced in parallel and distributed computing paradigms. In addition, PBQ will specifically face issues related to a larger state space matrix, resulting in more complex and time-consuming calculations and increased memory overhead.

D. Parameter Setting and Experimental Results

We consider 24, 48, and 96 tasks with various deadlines and execution times. We assume that six users have requested these tasks, and there are two available edge servers in the network. Furthermore, we executed the proposed model 31 times to account for the stochastic nature of the proposed technique. Subsequently, we computed the average of the results obtained from these 31 iterations for comparison with other works.

One advantage of PBQ is its ability to scale smoothly with respect to the number of users and edge servers in the network. The state space matrix of PBQ expands both vertically and horizontally as the number of users and edge servers changes, respectively. As a result, PBQ is not limited to a specific number of users and edge servers. Instead, it can handle User-Edge collaborations with varying numbers of users and edge servers. Consequently,

As illustrated in Fig. 3, PBQ improved the runtime of task scheduling by approximately 77% and 78%, 55% and 53%, and 26% and 17% in comparison to previous works DDLO

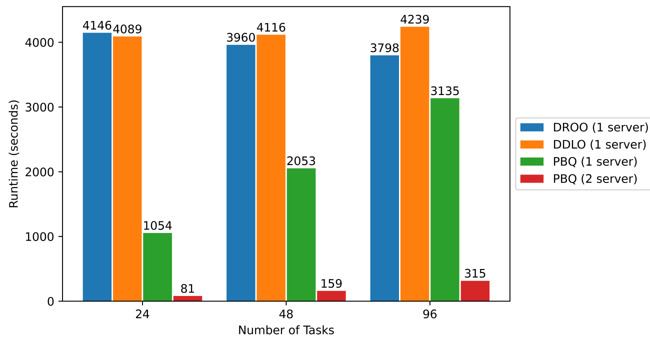

Fig. 3: Runtime of Scheduling Algorithms. The runtimes of all algorithms are recorded to compare the time it takes for each algorithm to learn task scheduling for users. In contrast to previous approaches, PBQ is a scalable technique capable of handling multiple servers in MEC. Scalability is a significant feature of an algorithm, particularly in the context of the dynamic nature of the MEC paradigm.

and DROO with 24, 48, and 96 tasks, respectively, when there is only one edge server in the network. In addition, PBQ leverages its scalability and load-balancing features when the network has more than one edge server. Consequently, PBQ achieved improvements of over 98%, 95%, and 91% compared to DDLO and DROO in scenarios involving 24, 48, and 96 tasks on the network with two servers. Indeed, DDLO and DROO can only consider an edge network with a single edge server, while PBQ can scale to manage varying numbers of edge users and edge servers. Since we intentionally eliminate invalid and unnecessary state spaces in the proposed customized QL method, the state space of PBQ becomes smaller and more efficient. Therefore, the PBQ can learn at a faster rate, resulting in significant improvements in the runtime of the algorithms when compared to DDLO and DROO.

Moreover, DDLO and DROO only support a single edge server and are incapable of scheduling users' tasks in scenarios where the edge network comprises multiple servers. In contrast, PBQ is a scalable and flexible scheduling algorithm that scales up and down based on the number of edge servers and users. Indeed, one of the significant differences between cloud and edge is that edge comprises multiple geographically distributed servers with varying hardware features, whereas cloud is a centralized computation paradigm that utilizes centralized homogeneous resources. Accordingly, an effective task scheduling technique in MEC must be capable of considering and managing multiple servers for task allocation. Consequently, our experiments investigate the PBQ technique by considering two servers for scheduling dataset tasks.

In addition to improving runtime, we also examine hardware-related parameters such as computation cost and memory usage to analyze the PBQ method compared to other methods comprehensively. The computation cost refers to the number of computation cycles an algorithm requires for its execution. According to Fig. 4, PBQ requires the fewest

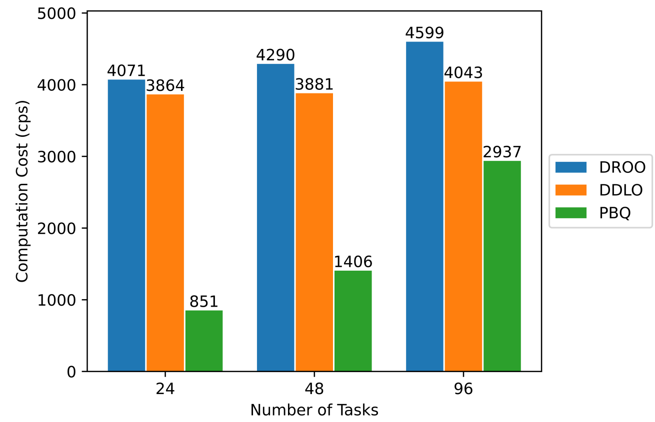

Fig. 4: Computing Cost of Scheduling Algorithms. MEC has limited computational power compared to cloud computing. Therefore, it is reasonable to consider the computational cost of algorithms designed for MEC. Hence, a computationally efficient algorithm would be more practical. Consequently, PBQ demonstrates superior computational efficiency compared to previous works.

computation cycles; compared to DDLO and DROO, PBQ reduces computation cycles by approximately 77% and 79%, 63% and 67%, and 27% and 36% in 24, 48, and 96 tasks, respectively. Furthermore, PBQ consumes the least memory

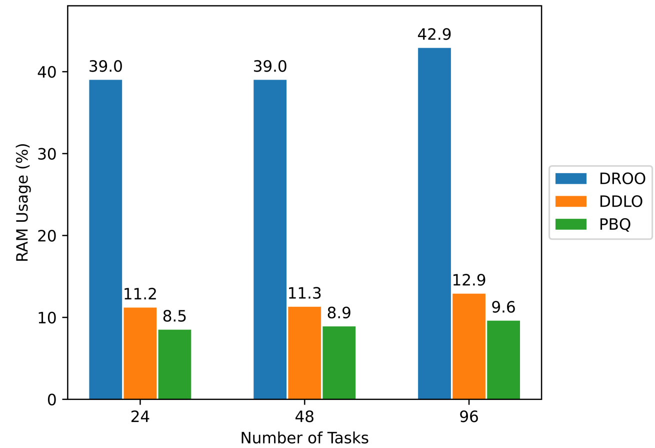

Fig. 5: RAM Utilization of Scheduling Algorithms. There is no memory pool in MEC, and memory crash is a challenge for algorithms that consume a large amount of memory in MEC. Thus, our intentional elimination in PBQ results in fewer state spaces and a smaller Q-table, leading to lower memory consumption.

compared to other methods by efficiently reducing the number of state spaces, resulting in a smaller Q-table employed in the learning process. As shown in Fig. 5, PBQ consumes 24% and 78%, 20% and 76%, and 25% and 77% less memory compared to DDLO and DROO in 24, 48, and 96 tasks.

VI. Conclusion

In this study, we have proposed a scalable QL-based algorithm named PBQ for task scheduling in MEC networks. Our purpose is to minimize scheduling time along with memory consumption. Since we apply MDP method to find a sequence of edge server allocation, creating state space of binary permutations would consume a huge amount of memory. To solve this issue, we employ a state filtering step to preserve states which satisfy our predefined conditions. Then, we come up with a priority-based task scheduling idea to accelerate scheduling speed. In fact, at each step of task allocation, we consider tasks with the earliest deadline. Moreover, we define a customized reward function based on state transaction efficiencies. Finally, we compared PBQ with DDLO and DROO scheduling methods. The experimental results demonstrate that, on average, PBQ has achieved improvements of 72.8% in runtime, 58.1% in computation cost, and 50% in memory usage compared to other methods. Moreover, we ran PBQ 31 times to check the validity of the proposed model, considering its stochastic nature. In the future, we intend to utilize Deep Q-learning for task offloading and consider more users and edge servers for scheduling problems. We also plan to take into account additional factors, including various device types, network connectivity, and user workload on the user and edge server sides. These extra works will give work scheduling decisions more accurate information and help maximize system performance.

References

[1] W. Kassab and K. A. Darabkh, "A–z survey of internet of things: Architectures, protocols, applications, recent advances, future directions and recommendations," *Journal of Network and Computer Applications*, vol. 163, p. 102663, 2020.

[2] D. C. Nguyen, M. Ding, P. N. Pathirana, A. Seneviratne, J. Li, D. Niyato, O. Dobre, and H. V. Poor, "6g internet of things: A comprehensive survey," *IEEE Internet of Things Journal*, vol. 9, no. 1, pp. 359–383, 2021.

[3] A. Avan, M. Taheri, M. H. Moaiyeri, and K. Navi, "Energy-efficient approximate compressor design for error-resilient digital signal processing," *International Journal of Electronics*, pp. 1–23, 2022.

[4] K. Li, X. Wang, Q. He, Q. Ni, M. Yang, M. Huang, and S. Dustdar, "Computation offloading for tasks with bound constraints in multi-access edge computing," *IEEE Internet of Things Journal*, 2023.

[5] A. Avan, A. Azim, and Q. H. Mahmoud, "A robust scheduling algorithm for overload-tolerant real-time systems," in *2023 IEEE 26th International Symposium on Real-Time Distributed Computing (ISORC)*. IEEE, 2023, pp. 1–10.

[6] W. Shi and S. Dustdar, "The promise of edge computing," *Computer*, vol. 49, no. 5, pp. 78–81, 2016.

[7] D. Evans, "The internet of things: How the next evolution of the internet is changing everything," *CISCO white paper*, vol. 1, no. 2011, pp. 1–11, 2011.

[8] S. Tschöke, F. Lynker, H. Buhr, F. Schreiner, A. Willner, A. Vick, and M. Chemnitz, "Time-sensitive networking over metropolitan area networks for remote industrial control," in *2021 IEEE/ACM 25th International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*. IEEE, 2021, pp. 1–4.

[9] M. Hartmann, U. S. Hashmi, and A. Imran, "Edge computing in smart health care systems: Review, challenges, and research directions," *Transactions on Emerging Telecommunications Technologies*, vol. 33, no. 3, p. e3710, 2022.

[10] M. Goudarzi, M. Palaniswami, and R. Buyya, "Scheduling iot applications in edge and fog computing environments: a taxonomy and future directions," *ACM Computing Surveys*, vol. 55, no. 7, pp. 1–41, 2022.

[11] R. Rajavel, S. K. Ravichandran, K. Harimoorthy, P. Nagappan, and K. R. Gobichettipalayam, "Iot-based smart healthcare video surveillance system using edge computing," *Journal of Ambient Intelligence and Humanized Computing*, pp. 1–13, 2022.

[12] Y. Deng, X. Chen, G. Zhu, Y. Fang, Z. Chen, and X. Deng, "Actions at the edge: Jointly optimizing the resources in multi-access edge computing," *IEEE Wireless Communications*, vol. 29, no. 2, pp. 192–198, 2022.

[13] Q. Luo, S. Hu, C. Li, G. Li, and W. Shi, "Resource scheduling in edge computing: A survey," *IEEE Communications Surveys & Tutorials*, vol. 23, no. 4, pp. 2131–2165, 2021.

[14] A. Avan, A. Azim, and Q. H. Mahmoud, "A state-of-the-art review of task scheduling for edge computing: A delay-sensitive application perspective," *Electronics*, vol. 12, no. 12, p. 2599, 2023.

[15] Y. Matsuo, Y. LeCun, M. Sahani, D. Precup, D. Silver, M. Sugiyama, E. Uchibe, and J. Morimoto, "Deep learning, reinforcement learning, and world models," *Neural Networks*, 2022.

[16] J. Liu, J. Ren, Y. Zhang, X. Peng, Y. Zhang, and Y. Yang, "Efficient dependent task offloading for multiple applications in mec-cloud system," *IEEE Transactions on Mobile Computing*, 2021.

[17] A. Mahjoubi, J. Taheri, K.-J. Grinnemo, and S. Deng, "Optimal placement of recurrent service chains on distributed edge-cloud infrastructures," in *2021 IEEE 46th Conference on Local Computer Networks (LCN)*. IEEE, 2021, pp. 495–502.

[18] Y. Wu, J. Wu, L. Chen, J. Yan, and Y. Luo, "Efficient task scheduling for servers with dynamic states in vehicular edge computing," *Computer Communications*, vol. 150, pp. 245–253, 2020.

[19] L. Cai, X. Wei, C. Xing, X. Zou, G. Zhang, and X. Wang, "Failure-resilient dag task scheduling in edge computing," *Computer Networks*, vol. 198, p. 108361, 2021.

[20] Y. Liu, S. Wang, Q. Zhao, S. Du, A. Zhou, X. Ma, and F. Yang, "Dependency-aware task scheduling in vehicular edge computing," *IEEE Internet of Things Journal*, vol. 7, no. 6, pp. 4961–4971, 2020.

[21] A. Chhabra, K.-C. Huang, N. Bacanin, and T. A. Rashid, "Optimizing bag-of-tasks scheduling on cloud data centers using hybrid swarm-intelligence meta-heuristic," *The Journal of Supercomputing*, pp. 1–63, 2022.

[22] G. Saravanan, S. Neelakandan, P. Ezhumalai, and S. Maurya, "Improved wild horse optimization with levy flight algorithm for effective task scheduling in cloud computing," *Journal of Cloud Computing*, vol. 12, no. 1, p. 24, 2023.

[23] D. Alboaneen, H. Tianfield, Y. Zhang, and B. Pranggono, "A metaheuristic method for joint task scheduling and virtual machine placement in cloud data centers," *Future Generation Computer Systems*, vol. 115, pp. 201–212, 2021.

[24] Y. Gao, W. Wu, H. Nan, Y. Sun, and P. Si, "Deep reinforcement learning based task scheduling in mobile blockchain for iot applications," in *ICC 2020-2020 IEEE International Conference on Communications (ICC)*. IEEE, 2020, pp. 1–7.

[25] B. Tang, J. Luo, M. S. Obaidat, and P. Vijayakumar, "Container-based task scheduling in cloud-edge collaborative environment using priority-aware greedy strategy," *Cluster Computing*, pp. 1–17, 2022.

[26] W. Zhan, C. Luo, J. Wang, C. Wang, G. Min, H. Duan, and Q. Zhu, "Deep-reinforcement-learning-based offloading scheduling for vehicular edge computing," *IEEE Internet of Things Journal*, vol. 7, no. 6, pp. 5449–5465, 2020.

[27] L. Huang, X. Feng, A. Feng, Y. Huang, and L. P. Qian, "Distributed deep learning-based offloading for mobile edge computing networks," *Mobile networks and applications*, pp. 1–8, 2018.

[28] L. Huang, S. Bi, and Y.-J. A. Zhang, "Deep reinforcement learning for online computation offloading in wireless powered mobile-edge computing networks," *IEEE Transactions on Mobile Computing*, vol. 19, no. 11, pp. 2581–2593, 2019.

[29] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, pp. 279–292, 1992.

[30] S. Baruah and J. Goossens, "Scheduling real-time tasks: Algorithms and complexity," *Handbook of scheduling: Algorithms, models, and performance analysis*, vol. 3, 2004.

[31] A. A. Bidgoli, S. Rahnamayan, T. Dehkharghanian, A. Riasatian, and H. R. Tizhoosh, "Evolutionary computation in action: Hyperdimensional deep embedding spaces of gigapixel pathology images," *IEEE Transactions on Evolutionary Computation*, vol. 27, no. 1, pp. 52–66, 2022.