Dispatching and Scheduling Dependent Tasks Based on Multi-agent Deep Reinforcement Learning

Shuaishuai Feng[†], Xi Wu[‡], Yongxin Zhao[*†§], and Yongjian Li[§]
[†] Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, Shanghai, China
[‡] The University of Sydney, Australia
[§] State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China

***Abstract*—With the development of edge computing, a large number of tasks can be offloaded to the edge server for computing, among which the dispatching and scheduling of dependent tasks has attracted extensive attention. The offloading of dependent tasks mainly has the following problems: how to select an appropriate edge server for dispatching, how to arrange the scheduling order of edge servers to better schedule tasks, and how to solve the task dependency problem. In this paper, we proposal a dispatching and scheduling method DAMD, based on reinforcement learning and multi-agent reinforcement learning, to solve the above three problems. Specifically, as the first step of DAMD, a reinforcement learning approach is designed to estimate the network load and dynamically dispatch tasks to the appropriate edge servers. Each edge server is regarded as an agent by a multi-agent reinforcement learning method, the second step of DAMD, which comprehensively considers the dependency relationship between tasks and the scheduling relationship between servers to achieve the efficiency and fairness of task scheduling. Finally, the results show that our method can better complete the task within the deadline and greatly reduce the average response time according to the time sensitivity requirement.**

***Index Terms*—dependent task, deep reinforcement learning, edge computing, multi-agent deep reinforcement learning**

I. Introduction

With the development of the Internet of Things (IoT), more and more applications can be processed on mobile terminals [1]. Limited by their limited (computing, storage, and bandwidth) capabilities, terminal devices may spend a lot of time performing the required tasks, potentially resulting in poor quality of service [2]. In order to obtain higher service quality and computing resources, we can offload tasks to remote cloud data centers, which have a large amount of computing resources but are far away from users [3]. However, the long distance between the cloud and the user also introduces significant communication latency, which is unacceptable for time-sensitive applications/services [4]. In this case, edge computing can play an important role [5]. In edge computing, many edge servers with computing resources are deployed close to users [6]. By offloading tasks from terminal devices to edge servers, users can get edge services with better quality of service (for example, lower latency and higher precision) than in the cloud computing model. But at the same time, resources on the edge server are limited and cannot provide services for all tasks, especially when there are a large number of tasks [7]. Therefore, the edge communication system needs to solve two basic problems: which edge server should be dispatched to accommodate the task and what order each edge server should schedule the task, namely task dispatching and scheduling problem [8]. In addition, task requests from the same user can usually be divided into a group of independent or dependent tasks, which are usually represented as a job. Among these dependent tasks belonging to a job, there is a strict execution sequence, and subsequent tasks need to wait for the completion of the previous task [9]. It is also a very important problem how to dispatch and schedule tasks under the guarantee of task dependency.

In the dispatch phase, the traditional method is to dispatch tasks to the nearest server [10], which may lead to unbalanced task allocation and overload of the server. meng $etal.$ proposed to assign tasks to the edge server with the minimum average response time [11], but did not consider the changes in network conditions and server load. In the scheduling stage, traditional methods, such as First-Come-First-Serve, do not consider the time problem caused by the size of the task. zhong $etal.$ proposed a scheduling scheme based on reinforcement learning [12], but it does not consider the dependence between tasks. Sundar $etal.$ considered dependent tasks [13], but they only considered one type of dependent tasks and the application was limited.

We propose a new task dispatching and scheduling method DAMD that combines reinforcement learning and multi-agent reinforcement learning. Specifically, we propose a dispatcher based on deep reinforcement learning and a scheduler based on multi-agent deep reinforcement learning. For the dispatcher, we take the deep Q-learning (DQN) approach and use the response time of the task as a reward to update the current state of the edge network conditions and server load in real time. This can effectively improve efficiency by selecting edge servers with maximum rewards, avoiding network congestion and server overload. For the scheduler, we adopt multi-agent deep reinforcement learning technology to fully consider the dependencies between tasks and the execution order of dependent tasks in other servers. As a result, each edge server can dynamically allocate resources based on the time-sensitive requirements of each task. Therefore, our approach minimizes the average task response time and maintains efficiency across

*Corresponding author: yxzhao@sei.ecnu.edu.cn
DOI reference number: 10.18293/SEKE2023-059

all tasks. The main contributions of this paper are:

1) We apply reinforcement learning and multi-agent reinforcement learning to the dispatch and scheduling of dependent tasks, comprehensively considering the dependency relationship between tasks and the cooperation between edge servers. As far as we know, this is the first time to solve the dispatch and scheduling of dependent tasks in this scenario.
2) We abstract the edge server and task information, and consider the load and distance of the edge server, which has a good application in real life.

The rest of this paper is structured as follows. In Section II, we introduced relevant scenes and the proposal of problems. In Section III, we introduced system modeling and algorithm process in detail. In Section IV, we compared and analyzed the results. Finally, we summarize this thesis in Section V.

II. Problem Definition and System Modeling

A. Problem Definition

In this section, we introduce the basic assumptions and question formulation.

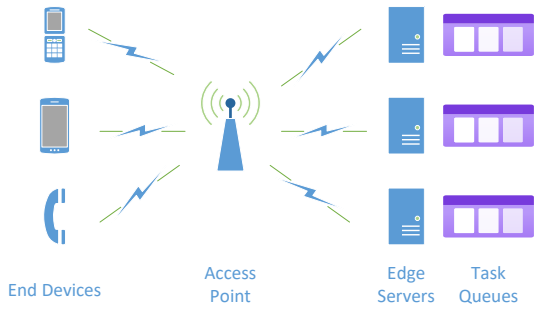


Fig. 1. An example of a scenario

As shown in Figure 1,we consider a target network with edge servers, different edge nodes provide different computing power. For each edge server, there may be multiple applications configured. In this network, users send jobs from their terminal devices to the access points (AP), which then dispatch the jobs to the edge server of the target edge network. When the jobs arrives at the edge server, it waits in the task queue for processing. The jobs generated by users can be dispatched to different edge servers for calculation. A job contains one or more dependent tasks. For dependent tasks, subsequent tasks must wait for the previous tasks to complete, and a task can only be dispatched to one edge server. In this process, we only consider the dispatching of jobs from AP to server and the scheduling within the server.

The problem can be expressed as follows: at a certain time, a group of dependent tasks are dispatched to edge servers in multiple networks. Given edge server information and dependent task information, each task is dispatched a edge server and the scheduling within the edge server, so that the tasks can be completed before the deadline as far as possible.

B. System Modeling

For the dispatching and scheduling of dependent tasks, we mainly consider the information of edge servers and tasks. The main components are described as follows:

Edge server : a group of server nodes $\mathcal{N} = \{N_1, N_2, \ldots, N_m\}$ is deployed in a specific area, and each node has the following characteristics:

- $N_i.CPU$: indicates the computing capacity of the server, that is, the CPU.
- $N_i.bd$: indicates the bandwidth of the server.
- $N_i.size$: indicates the memory size of the server.
- $N_i.size(t)$: indicates the remaining memory of the server
- $N_i.task(t)$: indicates the task running on the N_i server at time t.
- $N_i.wait(t)$: indicates the waiting queue on the Ni server at time t.

Jobs: a set of tasks that accomplish a specific goal, with dependencies between tasks, is a Directed acyclic graph (DAG). Through DAG, we can obtain the dependency relationships between tasks

Task: the smallest, indivisible task in a job. A task must be completed on one server.

- $T_i.size$: indicates the size of the task
- $T_i.start$: indicates the arrival time of the task
- $T_i.deadline$: indicates the end time of the task
- $T_i.pretasks$: indicates the set of previous tasks of the task, which means, the task can work only after the previous tasks of the task are completed.

To minimize task response time, we divide task delay into two parts: external delay and internal delay.

We divide the external delay D of a task into three parts.

- D_1: indicates the time for the task to arrive from the AP to the edge server.
- D_2: indicates the time when the edge server returns to the AP after the task is scheduled.
- D_3: indicates the time when the calculation result of the previous task of the task is returned from an edge server. If the task is on the same edge server as the previous task, the time is 0.

Since the bandwidth of AP is generally large, for an arbitrary time t, we assume that the link bandwidth of the task from AP to the edge server is determined by the bandwidth of the edge server, that is, the link bandwidth of the task to the edge server is the bandwidth of the edge server. In addition, there is a propagation delay k between AP and the edge server. Propagation delay k is related to the distance from the AP to the edge server. Then $D_1 = k + T_i.size/Ni.bd$.

When the task is returned by the edge server after scheduling, we assume that the data volume of the scheduling result is small. In this process, only the propagation delay is considered, that is, $D_2 = k$.

For dependent tasks, each task needs to wait for the result of the previous task. If the previous task of the task is not completed when it comes to the task scheduling, the task needs to wait for the transmission of the result before scheduling.

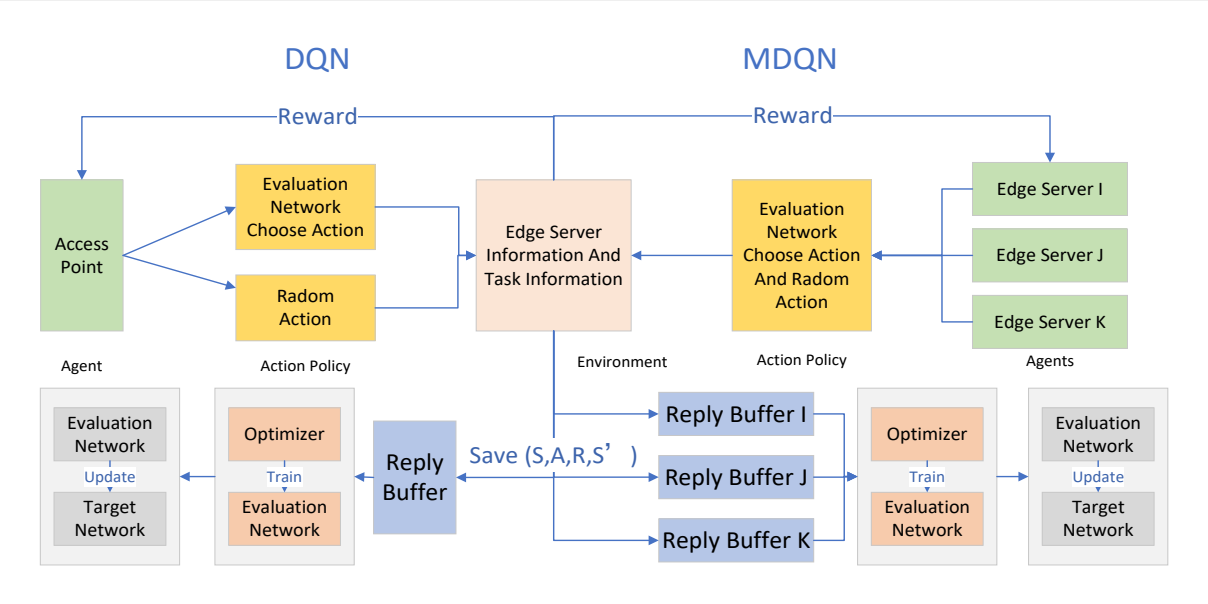


Fig. 2. Dependent task dispatching and scheduling algorithm

Due to the small amount of data processing results, $D_3 = k$; If all previous tasks have been completed when the task is scheduled, and the transmission time of previous task results is included in the waiting time of this task, then $D_3 = 0$.

Internal delay is mainly divided into two parts, task wait time and task processing time. When a task reaches the edge server, it will be placed in a waiting queue. When it comes to task scheduling, since the task is a dependent task, it is necessary to judge whether all the previous tasks of the task have been completed. The waiting time of the task is the completion time of the previous task in the queue minus the arrival time of the task. The processing time of the task is $T_i.size/N_i.CPU$. The storage resources of each edge server are limited. Therefore, the memory of the edge server must be determined before dispatched a task to the edge server. Dispatched can be performed only when the task size is smaller than the remaining memory of the server.

III. Dispatching and Scheduling Algorithm

A. Algorithm Framework

Our algorithm is mainly divided into two steps. First, we use reinforcement learning algorithm to dispatch tasks from AP to edge server. In this step, we do not consider the dependency between tasks, and select appropriate edge server for each task to dispatch. The action set is the edge server, which selects the action by evaluate network and gives the reward according to the response time of the task. The shorter the task response time, the smaller the load on the edge server. After one action is completed, the data is stored in the experience pool and periodically updated with the data in the experience pool.

Then, multi-agent reinforcement learning is used to realize internal scheduling of edge servers. We regard each edge server as an agent and set an experience pool for each edge server. Each edge server selects one task for scheduling at a time and judges whether the previous task of the task is completed before scheduling. The total action set is the tasks selected by each edge server. All edge servers select a task and give a reward. After total action is completed, the data is stored in their own experience pool, and the parameters are updated regularly with the data in the experience pool. In this way, the influence of other server scheduling and the dependency between tasks are fully taken into account.The algorithm flow is shown in Figure 2.

B. Task Dispatching Method

A typical reinforcement learning model consists of states, actions, strategies and rewards. Agents learn by interacting with the environment, make actions and get corresponding rewards.

State: The agent interacts with the environment to obtain the current state and make corresponding decisions at the same time. When a task arrives, the agent will obtain various information of the current edge server.

Action: For any task arrived, the agent finds a suitable edge server to dispatch by observing the environment. Therefore, the scope of the action is the set of all edge servers, defined as:

$$a_t \in \{n_1, n_2, \ldots, n_m\} = a \tag{1}$$

Policy: Task dispatching policy reflects the mapping relationship between state and action. In DQN, we use neural network to generate action.

Reward: After observing the environmental state at time t, the agent makes corresponding actions by strategy, and gets the corresponding reward at time $t+1$, here we define the reward as e^{-T}, T is the response time of the task, which reflects the

load on the edge server, and the shorter the response time, the less the load.

DQN designed two networks, the evaluation network and the target network, both of which initially had the same structure and parameter configuration. One is used to predict Q estimate (MainNet), one is used to predict Q reality (target), the targetQ of Q reality is calculated as:

$$\mathrm{targetQ} = \mathrm{r} + \gamma * \mathrm{Qmax}\left(\mathrm{s}', \mathrm{a}', \theta\right) \tag{2}$$

The loss is estimated by targetQ and Q, and the loss function generally adopts the mean square error loss:

$$\mathrm{LOSS}(\theta) = \mathrm{E}[(targetQ - Q(\mathrm{s}, \mathrm{a}, \theta))^2] \tag{3}$$

Initialize MainNet and target, update the MainNet parameters according to the loss function, and target is fixed. After several iterations, all the MainNet parameters are copied to the target network, and so on. The targetQ is fixed in a period of time, which makes the algorithm update more stable.

Algorithm 1 Task Dispatching Algorithm Based On Deep Q-learning

Input: Task information and edge server information
Output: The edge server to which the task is dispatched

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function Q with weights $\theta^- = \theta$
for episode $i = 1$ to N **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  for each step t **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \mathrm{argmax}_a\, Q(\phi(s_t), a; \theta)$
    Take action a_t and observe reward r_t and next state x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition (s_t, a_t, r_t, s_{t+1}) in D
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    Set $y_j = r_j$ if episode terminates at step j+1
    Set $y_j = r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-)$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j\ ; \theta)^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  end for
end for

C. Task Scheduling Method

When tasks are scheduled within the server, the scheduling sequence within one server may affect other servers due to the dependency between tasks. Therefore, we consider using the Multi-agent DQN (MDQN) to schedule tasks within the server. We create a task queue for each server, and each server gets a reward after selecting a task from the task queue to schedule.

Algorithm 2 Task Scheduling Algorithm Based On MDQN

Input: Task queue and edge server information
Output: Scheduling tasks within the server

Initialize replay memory D to capacity N
Initialize the evaluation network w_e and the target network with random parameter w_t
Update action policy
for episode $i = 1$ to N **do**
  Initializes the server queue information
  for each step t **do**
    for each server M **do**
      Choose action A according to action policy and $Q(S, A, w_e)$
      Take action A, observe R and S'
      Store $e = (S, A, R, S')$
      Sample random pair of e from memory
      Calculate target $y = R + \beta maxQ(S', A', w_t)$
      Train parameter w_e with a gradient descent step $(y - Q(S, A, w_e))^2$
      if update =true **then**
        $w_t \leftarrow w_e$
      end if
      $S \leftarrow S'$
    end for
    Update server queue information
  end for
end for

State value function: The state value function is the expectation of the action value function about the action, and the action performed by the agent depends on the strategy function, so the state value function V_i of the agent also depends on the strategy of all the agents. For a single agent, its state space is the state information of a single server and the actions of other agents. The total state space is the state space of all agents.

Reward: Different relationships between multiple agents will result in different rewards given by the environment. If there is a cooperative relationship between multiple agents, the agents receive the same reward from the environment. If multiple agents are in a competitive relationship, positive reward for one agent from the environment will lead to negative reward for another agent. In the task scheduling environment, we regard multiple agents as cooperative relationships, and give a unified reward after all servers select a task for scheduling.

Action value function: For each server, its action is to select a task from the task queue to schedule. We create an experience pool for each agent, and store these experiences in an experience pool during the early stages of training, when the agent interacts with the environment to make an action. The experience contains S, R, A, and S 'information.

At the same time, we set up a target network with the same structure as the evaluation network for training, because a single network design will make the algorithm fall into the feedback loop between the target and the estimated Q value

and become unstable. Therefore, a target network is used to avoid estimates getting out of control. The parameters of the evaluation network were updated at each training step, and the evaluation network was used to estimate the Q value of the action. The parameters of the target network are relatively stable. After several steps, the target network updates the parameters to the same as the evaluation network.

IV. Experimental Results

In this section, we will evaluate the performance of the proposed approach by generating simulated environment information and data information compared to the baseline approach.

A. Experimental Environments

We use data sets generated by the network, which include information such as arrival time, processing time, data size, etc. At the same time, there is a dependency relationship between tasks. We randomly generate the dependency graph between tasks, and use the network to generate simulated edge server information, including the memory size and CPU of the edge server.

B. Baseline Method Comparison

In order to better evaluate the performance of the method and reflect the efficiency and fairness of the method in task dispatching and scheduling, we conducted comparative experiments with the following three baseline algorithms.

First is dispatching method baseline, this section is the method used by the task to get from the access point (AP) to the edge server. In order to show the performance of our dispatching method, we compare it with the three baseline dispatching methods.

1) Nearest: Dispatch the task to the nearest edge server.

2) Random: Dispatch tasks randomly to an edge server.

3) Least load: Dispatch tasks to edge servers with minimal load, here we use the waiting time of the task to represent, the longer the waiting time of the task, the greater the load of the edge server.

Then is scheduling method baseline,this section describes the method used to schedule tasks internally at the edge server, which we compare with the following three methods.

1) First-Come-First-Serve: Arrange tasks according to the order in which they arrive. Early tasks are scheduled first and final tasks later.

2) Shortest-Job-First:Schedule tasks based on the completion time, the shorter the completion time, the earlier the task is scheduled.

3) Shortest-Deadline-First: Tasks are scheduled based on their deadline time. The earlier the deadline time, the more urgent the task is, and the earlier the task is scheduled.

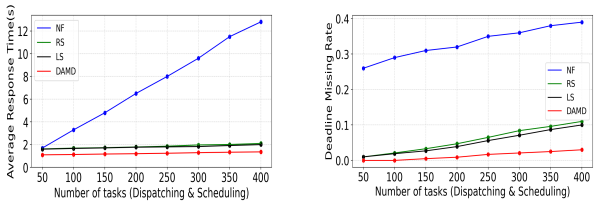
Since tasks are scheduled in two parts, from the AP to the edge server and scheduled in the edge server, we combine a pair of dispatching and scheduling baselines to compare dispatching and scheduling performance. These are the Nearest + First-Come-First-Serve (NF), Random + Shortest-Job-First (RS) and Least load + Shortest-Deadline-First: (LS)

C. Evaluation Result

In this part, we will compare our method with the baseline method in the simulation environment, and verify the superiority of our method over the baseline method by comparing the response time of tasks and the deadline missing rate of tasks under the same environment.

1) The impact of the number of tasks.

We randomly generated the edge server information and the dependent task information to conduct the experiment, and the dependent task was dispatched and scheduled by these methods. By comparing the response time of the task and the deadline missing rate of the task, the performance of our method was compared with that of the baseline method.The result is shown in Figure 3, the vertical axis are average response time and deadline missing rate, the horizontal axis is number of tasks.



(a) (b)

Fig. 3. The effect of different task numbers on average response time and deadline missing rate

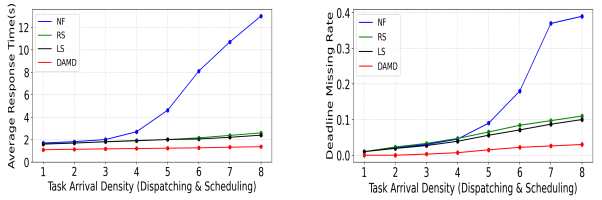
When a task is dispatched by an AP to an edge server, the nearest edge server receives an extremely large number of tasks as only the task is dispatched to the nearest edge server, resulting in a very high average task response time and deadline missing rate. Our approach determines which edge servers to dispatch tasks to based on network conditions and the load on the edge servers, which results in good performance.

When a task is scheduled within an edge server, First-Come-First-Serve first schedules the task that arrives at the edge server first, which will cause the later task to wait longer, and if the later task is the previous task of the task in other edge servers, it will cause the dependent task in other edge servers to wait for a long time. The basic idea of Shortest-Job-First is to give higher priority to smaller tasks, which results in longer waits for larger tasks that arrive first, and does not take into account the impact of dependent tasks in other edge servers. Shortest-Deadline-First also does not take into account dependencies between tasks, and does not take into account the effects of dependencies on other edge servers, which makes it less effective.

2) The impact of task density

Task arrival density represents the number of tasks arriving at the edge server per unit time, and the larger the number, the higher task arrival density. We measured the performance of the different approaches in terms of average task response time and deadline missing rate. Our method and Least Load can

dispatch tasks to edge servers with lower load, and thus better handle higher task arrival densities. Moreover, our method can take into account the dependencies between tasks and the effects of other server scheduling tasks, so the performance is the best.The result is shown in Figure 4, the vertical axis are average response time and deadline missing rate, the horizontal axis is task density.

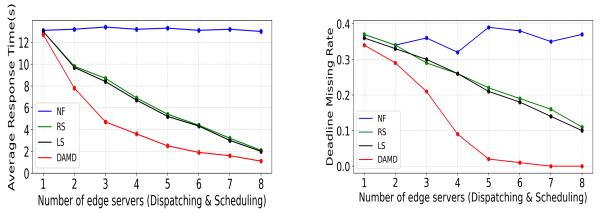


(a) (b)

Fig. 4. The effect of different task arrival density on average response time and deadline missing rate

3) The impact of the number of edge servers

We compare the performance of the different approaches with a different number of edge servers. We can see that when the number of servers is small, the deadline missing rate of tasks is very high. This is because the task is only dispatched to a few servers, which results in server overload. However, as the number of servers increases, the load of server decreases and the deadline missing rate decreases. Among all methods, our method has the best performance in terms of average task response time and deadline missing rate.The result is shown in Figure 5, the vertical axis are average response time and deadline missing rate, the horizontal axis is number of edge servers.



(a) (b)

Fig. 5. The effect of different edge server numbers on average response time and deadline missing rate

V. Conclusion and Future Work

In this paper, we propose a method DAMD combining DQN and MDQN for dispatching and scheduling dependent tasks. It can estimate the network condition and server load, and make reasonable arrangements according to the timeliness requirements of tasks. At the same time, it considers the dependency between tasks comprehensively, and make different servers cooperate with each other. We conducted evaluation experiments on simulated environments, and the result shows that the proposed method can make optimal actions through continuous learning from experience, so it performs well in dispatching and scheduling of dependent tasks. However, the algorithm can still be improved. Considering that the dependent task can be represented as a DAG graph, perhaps the introduction of graph neural network can improve the efficiency, simultaneously considering the competition and cooperation methods of multi-agent reinforcement learning. We will continue to carry out research and make contributions to the dispatching and scheduling of dependent tasks.

VI. Acknowledgement

This work is supported by National Key Research and Development Program (2019YFB2102600), National Natural Science Foundation of China (NSFC 62272165), the "Digital Silk Road" Shanghai International Joint Lab of Trustworthy Intelligent Software (Grant No. 22510750100), and Shanghai Trusted Industry Internet Software Collaborative Innovation Center.

References

[1] J. Pan and J. McElhannon, "Future edge cloud and edge computing for internet of things applications," *IEEE Internet of Things Journal*, vol. 5, no. 1, pp. 439–449, 2018.

[2] Y. Song, S. S. Yau, R. Yu, X. Zhang, and G. Xue, "An approach to qos-based task distribution in edge computing networks for iot applications," in *2017 IEEE International Conference on Edge Computing (EDGE)*, 2017, pp. 32–39.

[3] X. Chen, L. Jiao, W. Li, and X. Fu, "Efficient multi-user computation offloading for mobile-edge cloud computing," *IEEE/ACM Transactions on Networking*, vol. 24, no. 5, pp. 2795–2808, 2016.

[4] K. Boos, D. Chu, and E. Cuervo, "Flashback: Immersive virtual reality on mobile devices via rendering memoization," in *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, 2016, p. 291–304.

[5] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the internet of things," in *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, 2012, p. 13–16.

[6] C. Martín Fernández, M. Díaz Rodríguez, and B. Rubio Muñoz, "An edge computing architecture in the internet of things," in *2018 IEEE 21st International Symposium on Real-Time Distributed Computing (ISORC)*, 2018, pp. 99–102.

[7] H. Tang, H. Wu, Y. Zhao, and R. Li, "Joint computation offloading and resource allocation under task-overflowed situations in mobile-edge computing," *IEEE Transactions on Network and Service Management*, vol. 19, no. 2, pp. 1539–1553, 2022.

[8] H. Yuan, G. Tang, X. Li, D. Guo, L. Luo, and X. Luo, "Online dispatching and fair scheduling of edge computing tasks: A learning-based approach," *IEEE Internet of Things Journal*, vol. 8, no. 19, pp. 14 985–14 998, 2021.

[9] L. Liu, R. Zhong, W. Zhang, Y. Liu, J. Zhang, L. Zhang, and M. Gruteser, "Cutting the cord: Designing a high-quality untethered vr system with low latency remote rendering," in *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*, 2018, p. 68–80.

[10] M. Jia, J. Cao, and W. Liang, "Optimal cloudlet placement and user to cloudlet allocation in wireless metropolitan area networks," *IEEE Transactions on Cloud Computing*, vol. 5, no. 4, pp. 725–737, 2017.

[11] J. Meng, H. Tan, C. Xu, W. Cao, L. Liu, and B. Li, "Dedas: Online task dispatching and scheduling with bandwidth constraint in edge computing," in *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, 2019, pp. 2287–2295.

[12] J. H. Zhong, D. L. Cui, Z. P. Peng, Q. R. Li, and J. G. He, "Multi workflow fair scheduling scheme research based on reinforcement learning," *Procedia Computer Science*, vol. 154, pp. 117–123, 2019.

[13] S. Sundar and B. Liang, "Offloading dependent tasks with communication delay and deadline constraint," in *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, 2018, pp. 37–45.