Dependency-Aware Task Offloading based on Application Hit Ratio

Junna Zhang, *Member, IEEE,* Xinxin Wang, Peiyan Yuan, *Member, IEEE,* Hai Dong, *Senior Member, IEEE,* Pengcheng Zhang, *Member, IEEE* and Zahir Tari

Abstract—Mobile devices commonly offload latency-sensitive applications to edge servers to meet low-latency requirements. However, existing studies overlook dependency and application hit ratio considerations, hindering effective offloading for multi-applications and multi-tasks. To this end, this paper proposes a Dependent task offloading and Service placement Optimization (DSO) method to maximize the application hit ratio, thereby providing high-quality service. The proposed DSO includes Improved Multi-Agent Q-Learning (IMAQL) and greedy algorithms. IMAQL optimizes service placement via Q-learning, while the greedy algorithm schedules task offloading. Extensive experiments on public datasets demonstrate that the DSO method enhances the application hit ratio by 4.7% to 11.7% and reduces the completion time by about 3.4% to 4.9% compared to alternative approaches.

Index Terms—Edge computing, service placement, dependency, application hit ratio, improved multi-agent Q-learning algorithm

1 Introduction

The rapid advancement of mobile communication technologies has led to a substantial increase in the number of mobile devices [1]. Projections from IHS Markit anticipate that the number of mobile devices is expected to exceed 75 billion by 2025 [2]. The proliferation of mobile devices has facilitated the emergence of latency-sensitive applications, such as intelligent driving and virtual reality [3]. However, mobile devices are limited by their battery life, storage capacity and computing power. It is impractical for mobile devices to process these applications locally to meet low-latency requirements [4]. Edge computing, which provides computing and storage resources at the edge of the network, has emerged as a crucial solution to reduce response delay. It allows mobile devices to offload tasks to edge servers, which enhances the processing capabilities of mobile devices and alleviates the problem of resource shortages [5].

As a research hotspot in edge computing, task offloading has drawn wide attention from both academia and industry. However, existing studies on task offloading often overlook the dependencies between tasks. In fact, the tasks of applications are often characterized by dependencies. Over 75 percent of applications exhibit inter-task dependencies, as evidenced by Alibaba's analysis of 4 million applications [6]. The dataset used in our experiments confirms that approximately 64 percent of the applications have tasks with dependencies [7]. Fig. 1 shows the dependency task graph of a vehicular navigation application. The user first inputs a destination on the navigation device, which activates the task *control* to check the current GPS location. This information is then transmitted to tasks *map* and *traffic* to obtain all the alternative paths and traffic conditions along the direction to the destination. Depending on the available routes and the current traffic conditions, the task *path* can provide an optimal navigation route. Obviously, the execution of the task *path* depends on the execution results of tasks *map* and *traffic*. Therefore, the task *path* has dependencies on the task *map* and the task *traffic*. Task dependency is articulated by the reliance of a task's execution on the outcomes of the preceding task [8]. Existing offloading works [9], [10], [11] treat the above four tasks as an "inseparable" unit. This offloading may increase the overall execution time. In this case, it makes more sense to offload tasks *map* and *traffic* to separate edge servers. This separation allows the parallel execution of these two tasks, thereby minimizing its latency.

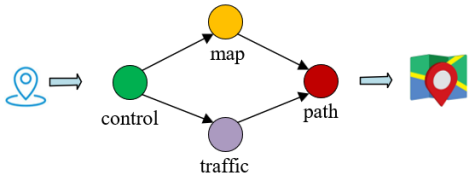


Fig. 1. The dependency task graph of a vehicular navigation application. Depending on the alternative routes provided by the task *map* and the current traffic conditions provided by the task *traffic*, the task *path* gives the optimal navigation route. As a result, the task *path* has dependencies on the task *map* and the task *traffic*.

Certain applications with low latency requirements necessitate offloading their tasks to edge servers rather than the cloud. For example, autonomous driving requires ultra-low latency and high reliability. If vehicle data is transmitted to the cloud and stored in a data center, it increases the chances of information leakage [12]. Meanwhile, latency exceeding anticipated thresholds can result in fatal accidents. Due to the limited storage resources of edge servers, only

- *Junna Zhang, Xinxin Wang and Peiyan Yuan are with the School of Computer and Information Engineering, Henan Normal University, and also with Engineering Lab of Intelligence Business & Internet of Things, Xinxiang, Henan 453007, China. E-mail: jnzhang@htu.edu.cn, 2108283081@stu.htu.edu.cn, peiyan@htu.cn.*
- *H. Dong and Z. Tari are with the School of Computing Technologies and the Centre for Cyber Security Research and Innovation, RMIT University, Melbourne, Australia. E-mails: {hai.dong, zahir.tari}@rmit.edu.au;*
- *Pengcheng Zhang is with the Key Laboratory of Water Big Data Technology of Ministry of Water Resources and the College of Computer and Software, Hohai University, Nanjing, Jiangsu 211100, China. E-mail: pchzhang@hhu.edu.cn*

Manuscript received XX XXX, XXXX; revised XX XXX, XXXX. (Corresponding author: Pengcheng Zhang.)

a subset of services can be placed on each edge server. Here services refer to the necessary programs to perform computing tasks [13]. Therefore, tasks are executed on the edge servers where the required services are available. For instance, the feature extraction task in a face recognition application will be successfully executed if it is offloaded to those edge servers that deploy the machine learning model.

As shown in Fig. 2(a), three tasks are offloaded to two edge servers. Tasks 2 and 3 can only initiate once task 1 has been successfully completed and the necessary data has been transmitted to them. Edge server e_1 can meet two types of service requests (i.e., "green quad" and "yellow diamond") while e_2 can meet one type of service request (i.e., "purple triangle"). Therefore, tasks 1 and 2 are offloaded to e_1, and task 3 is offloaded to e_2. For simplicity, the processing time for all tasks on edge servers is set to 1, and the data transmission time for tasks on different edge servers is set to 0.5. Assuming that the time to return the calculated results to the mobile device is not taken into account. The completion time of the application amounts to 2.5, as illustrated in Fig. 2(c). Likewise, task 1 is offloaded to e_1, and tasks 2 and 3 are offloaded to e_2 in Fig. 2(b). This results in a completion time of 3.5, as depicted in Fig. 2(d). Obviously, the service placement scheme in Fig. 2(a) is more appropriate than Fig. 2(b) because it reduces the response latency. In essence, the services deployed on edge servers significantly influence task offloading decisions. This subsequently impacts the completion time of applications. Existing works [14], [15], [16] concentrate on dependency-aware task offloading in edge computing. However, they overlook the impact of service placement. Limited edge server storage prevents deploying all required services, revealing constraints in practical scenarios.

The existing studies [17], [18], [19] focus on task hit ratio optimization. This task hit ratio is expressed as the proportion of successful service requests to the total number of service requests [17]. As illustrated in Fig. 3(a), edge server e_1 can deploy two types of services while e_2 can deploy a single service type. Each mobile device corresponds to a task requesting a specific service. To maximize task hit ratios, which is an aggregate measure across all participating edge servers, edge servers should preferentially deploy services with the highest number of requests. To this end, edge server e_1 provides the services of "blue square" and "yellow diamond" while e_2 offers the "purple triangle" service. In this case, this hit ratio reflect the scenario where both e_1 and e_2 are missing the red circle and green star services, leading to 6 hits out of 8 requests. However, a typical application usually comprises a series of tasks considering the growing complexity of applications. As a result, the application hit ratio needs to be considered, which is expressed as the percentage of hit applications to all applications. Whether an application is hit depends on two conditions. Firstly, all its tasks have been executed. Secondly, the completion time of the application does not exceed its deadline. This is pivotal as, in entertainment applications, missing deadlines translates into diminished service satisfaction, whereas in industrial settings, it can incur financial losses [20]. As shown in Fig. 3(b), each mobile device requests multiple services. If we preferentially deploy services with the highest number of requests on edge servers, the results of service

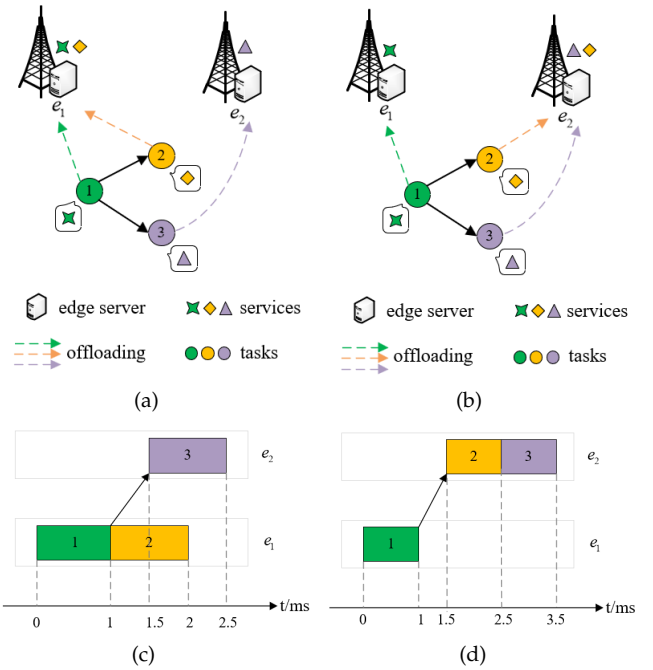


Fig. 2. Effect of service placement results on task offloading decisions. Fig. 2(a) and Fig. 2(b) show task offloading decisions under different service placement results. Fig. 2(c) and Fig. 2(d) show the completion time of the application under corresponding offloading decisions.

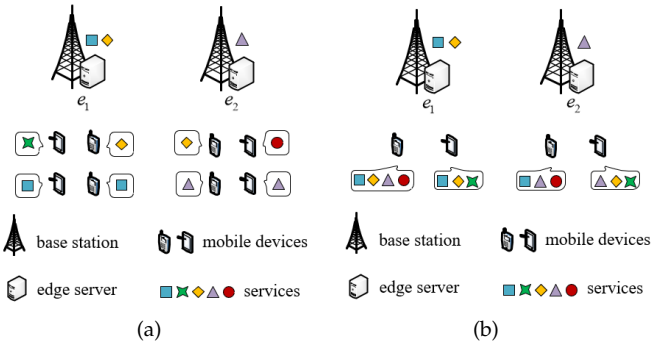


Fig. 3. Difference between task hit ratio and application hit ratio. Fig. 3(a) shows the task hit ratio, and Fig. 3(b) shows the application hit ratio.

placement are consistent with Fig. 3(a). In this case, all applications fail to execute successfully due to missing the essential services. Consequently, the execution of a complete application necessitates to deploy all the necessary services.

Although some studies have considered the application hit ratio, they ignore the impact of dependency [21], [22] or service placement [23]. Both dependency and service placement play crucial roles in influencing the feasibility and performance of task offloading. To address the aforementioned limitations, this paper researches dependent task offloading and service placement problems to maximize the application hit ratio. The main contributions of this paper are summarized as follows.

To the best of our knowledge, this study represents the inaugural exploration of task offloading, specifically emphasizing the application hit ratio in the context of dependency and service placement.

To improve the efficiency of service placement, we propose an Improved Multi-Agent Q-Learning (IMAQL) algorithm. Compared with the original Q-learning algorithm, this algorithm mitigates the risk

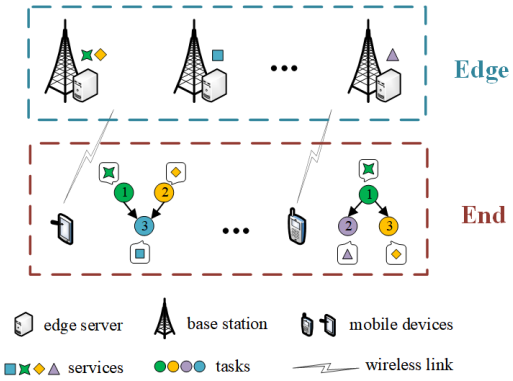


Fig. 4. Network model. Shapes with different colors on edge servers represent different services. The services requested by tasks are in the boxes next to them.

of the model converging to local optima.

Simulation results show that the proposed method can converge to the optimal performance similar to that of the enumerate method after training.

2 System Model

2.1 Network Model

The network model is shown in Fig. 4, which is composed of two layers of architecture. The first layer consists of several base stations and edge servers. Base stations are the communication nodes of the network, and edge servers are the computing nodes of the network. It is assumed that each base station is equipped with an edge server. Consequently, edge servers and base stations have the same index, denoted as $E = \{e_1; e_2; \ldots; e_m; \ldots; e_M\}$. Due to the limited storage and computation resources of edge servers, the maximum computing and storage capacity of the edge server e_m is given by F_m and C_m, respectively. The second layer is comprised of a number of mobile devices, denoted as $U = \{u_1; u_2; \ldots; u_n; \ldots; u_N\}$. Each mobile device consists of a series of dependent tasks, and each task requests a kind of service. Let K represent the total number of service types, indicating the presence of K distinct types of services. As illustrated in Fig. 2(a), three different service types are depicted, symbolized by a green quadrangle, a yellow diamond, and a purple triangle, respectively, hence K=3. The set of services is expressed as $S = \{s_1; s_2; \ldots; s_k \ldots; s_K\}$.

2.2 Application Model

Each mobile device $u_n \in U$ generates an application that comprises multiple interdependent tasks. To represent the logical sequence of these tasks, they are mapped to a Directed Acyclic Graph (DAG) $G_n = < A_n; B_n >$, where A_n represents the set of tasks and B_n denotes the set of directed edges characterizing the dependencies between tasks. Specifically, $A_n = \{a_{n;1}; a_{n;2}; \ldots; a_{n;i}; \ldots; a_{n;I}\}$, where $a_{n;i}$ means the i-th task of application u_n. Each tuple in B_n is structured as $(a_{n;j}; a_{n;i}; d^n_{j;i})$, where $a_{n;j}$ and $a_{n;i}$ are the source and destination nodes of the edge, respectively, and $d^n_{j;i}$ is the weight (or data size) associated with that edge. If there exists an edge between tasks $a_{n;j}$ and $a_{n;i}$, task $a_{n;i}$ will be executed only if task $a_{n;j}$ is completed and the corresponding data is transmitted to it. Task $a_{n;j}$ is called the immediate predecessor of task $a_{n;i}$, while task $a_{n;i}$ is called the immediate successor of task $a_{n;j}$. Each task $a_{n;i}$ is represented by a tuple $(d_{n;i}; w_{n;i})$, where $d_{n;i}$ denotes the size of input data for the task, and $w_{n;i}$ denotes the required total CPU cycles to complete the task. In addition, the maximum deadline for application u_n is expressed as T^{max}_n, which indicates that the application should be completed under this time constraint. Since each mobile device requests an application that is inherently associated with a DAG, a common index is utilized to correlate the mobile device, its application, and the corresponding DAG.

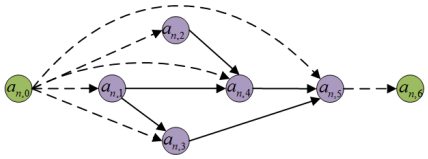


Fig. 5. DAG model. Green circles represent dummy tasks. Purple circles represent real tasks. The dotted lines with arrows indicate the amount of data transferred between dummy tasks and real tasks. The solid lines with arrows denote the amount of data transferred between real tasks.

When an application is completed, its results must be transferred back to the original mobile device. To model the DAG more clearly, we add two dummy tasks with zero workloads, called $a_{n;0}$ and $a_{n;I+1}$, at the beginning and the end of the original DAG. They are forced to execute locally. Fig. 5 shows an example of a DAG model, composed of five tasks with two dummy tasks ($a_{n;0}$ and $a_{n;6}$). There is a dependency between dummy task $a_{n;0}$ and all real tasks. The amount of data transmitted between them expresses the data size of each real task. As a result, the input data size for any real task $a_{n;i}$ is equal to the amount of data transferred from the dummy task $a_{n;0}$ to $a_{n;i}$ (i.e., $d_{n;i} = d^n_{0;i}$). Similarly, there exists a dependency between real task $a_{n;5}$ (without successor) and dummy task $a_{n;6}$. The amount of data between them represents the size of the application result transferred back to the mobile device u_n.

3 Problem Formulation

In this section, we research the service placement and task offloading problems, aiming to maximize application hit ratios and concurrently minimizing the completion time for hit applications. We first define the decision variables, then determine the optimization objective, provide the key constraints and finally formulate the optimization problem.

3.1 Decision Variables

$x^k_m \in \{0; 1\}$: indicating whether service s_k is placed at edge server e_m. If service s_k is placed at edge server e_m, $x^k_m = 1$; otherwise, $x^k_m = 0$. Let $X = (x^k_m \in \{0; 1\}; \forall k \in \{1; \ldots; K\}; \forall m \in \{1; \ldots; M\})$ express the service placement strategy.

$y^m_{n;i} \in \{0; 1\}$: indicating whether task $a_{n;i}$ is offloaded to edge server e_m. If task $a_{n;i}$ is offloaded to edge server e_m, $y^m_{n;i} = 1$, otherwise, $y^m_{n;i} = 0$. Let $Y = (y^m_{n;i} \in \{0; 1\}; \forall n \in \{1; \ldots; N\}; \forall a_{n;i} \in A_n; \forall m \in \{1; \ldots; M\})$ express the offloading strategy.

3.2 Completion Delay

3.2.1 Communication Delay

According to Shannon's formula [24], the uplink transmission rate from mobile device u_n to edge server e_m can be obtained as

$$R^{up}_{n;m} = W^{up} \log_2\left(1 + \frac{p_n h_{n;m}}{2}\right) \quad (1)$$

where W^{up} represents the uplink channel bandwidth. p_n is the transmission power of mobile device u_n. $h_{n;m}$ means the channel gain of the link between mobile device u_n and edge server e_m. ^{2} denotes the background noise power.

Likewise, the downlink transmission rate from edge server e_m to mobile device u_n can be given as

$$R^{down}_{m;n} = W^{down}\log_2(1 + \frac{p_m h_{m;n}}{^2}) \quad (2)$$

Specially, task $a_{n;i}$ is forwarded to edge server e_m via the directly connected edge server e_y. At this point, this uplink transmission time is the sum of the transmission time from mobile device u_n to edge server e_y and the transmission time from edge server e_y to e_m. The transmission rate between edge servers e_y and e_m is denoted as $R_{y;m}$. Hence, the uplink transmission delay of task $a_{n;i}$ is denoted as

$$T^{up;m}_{n;i} = \begin{cases} \frac{d_{n;i}}{R^{up}_{n;m}}; & e_y = e_m \\ \frac{d_{n;i}}{R^{up}_{n;y}} + \frac{d_{n;i}}{R_{y;m}}; & e_y \neq e_m \end{cases} \quad (3)$$

Similarly, the downlink transmission delay is denoted as

$$T^{down;m}_{n;i} = \begin{cases} \frac{d^n_{i;l+1}}{R^{down}_{m;n}}; & e_y = e_m \\ \frac{d^n_{i;l+1}}{R_{m;y}} + \frac{d^n_{i;l+1}}{R^{down}_{y;n}}; & e_y \neq e_m \end{cases} \quad (4)$$

When two dependent tasks $a_{n;j}$ and $a_{n;i}$ are assigned to edge servers e_y and e_m respectively, the data transmission delay between them is 0 if $e_y = e_m$, otherwise it is $\frac{d^n_{j;i}}{R_{y;m}}$. However, in specific cases where task $a_{n;j}$ is the dummy task $a_{n;0}$, the transmission delay is equal to the uplink transmission delay of task $a_{n;i}$. Similarly, if task $a_{n;i}$ is the dummy task $a_{n;l+1}$, the transmission delay is equal to the downlink transmission delay of task $a_{n;j}$. As a result, the data transmission time between them is expressed as

$$T^{tran}_{j;i} = \begin{cases} 0; & e_y = e_m \\ \frac{d^n_{j;i}}{R_{y;m}}; & e_y \neq e_m \\ T^{up;m}_{n;i}; & a_{n;j} = a_{n;0} \\ T^{down;y}_{n;j}; & a_{n;i} = a_{n;l+1} \end{cases} \quad (5)$$

3.2.2 Computation Delay

we define f_m to represent the computing power of edge server e_m, which reflects its capability to perform tasks without exceeding its maximum computing capacity. The computing delay of task $a_{n;i}$ is denoted as

$$T^{exe;m}_{n;i} = \frac{w_{n;i}}{f_m} \quad (6)$$

3.2.3 Wait Delay

It is assumed that each edge server executes tasks serially, so only one task can be performed at a time. The execution order of tasks is determined by the arrival order of tasks. Each edge server has a waiting queue to cache tasks that need to be executed. A task starts to be executed when two conditions are met. Firstly, its all immediate predecessors have been completed, and the execution results are transferred to it. Secondly, all tasks that arrived earlier than it in the waiting queue have been completed.

The set of immediate predecessors of task $a_{n;i}$ is expressed as $pre(a_{n;i})$. The time when task $a_{n;i}$ receives the results of its all immediate predecessors is denoted by

$$T^{rec}_{n;i} = \max_{a_{n;j} \in pre(a_{n;i})} \{T^{fin;y}_{n;j} + T^{tran}_{j;i}\} \quad (7)$$

Notably, if a predecessor task is the dummy task $a_{n;0}$ (typically used to signify the beginning of a task without consuming actual time), its completion time is denoted as 0.

$T^{fin;m}_{n^0;q}$ is used to denote the completion time of the task $a_{n^0;q}$ before task $a_{n;i}$ in the waiting queue. Therefore, the queuing time for task $a_{n;i}$ is expressed as

$$T^{queue;m}_{n;i} = \begin{cases} 0; & l = 1 \\ T^{fin;m}_{n^0;q}; & l > 1 \end{cases} \quad (8)$$

where l indicates the position of task $a_{n;i}$ in the wait queue. As a result, the wait time of task $a_{n;i}$ is denoted as

$$T^{wait;m}_{n;i} = \max\{T^{rec;m}_{n;i}; T^{queue;m}_{n;i}\} \quad (9)$$

The completion time of task $a_{n;i}$ is determined by its wait time and computation time, which can be expressed as

$$T^{fin;m}_{n;i} = T^{wait;m}_{n;i} + T^{exe;m}_{n;i} \quad (10)$$

After all tasks in the DAG are scheduled, the dummy task $a_{n;l+1}$ receives the execution results of offloaded tasks, so the completion time of application u_n is denoted as

$$T^{fin}_n = T^{rec}_{n;l+1} \quad (11)$$

For both linear and nonlinear DAGs, the uplink transmission time of each task is calculated utilizing Equation (3). This is because it solely considers the time required to transmit the task from the original mobile device to the target edge server, neglecting the time needed for the edge server to receive the task. However, it is imperative to ensure that the receiving time in Equation (7) accurately reflects the longest path within the DAG.

3.3 Application Hit Ratio

An application is hit only when both conditions are met. Firstly, all tasks generated by the application are executed. Secondly, the completion time of the application does not exceed its deadline. Let M_n show whether the application u_n is hit, and it is denoted as

$$M_n = \begin{cases} 1; & \sum_{i=1}^{} \sum_{m=1}^{} y^m_{n;i} = 1 \& 0 < T^{fin}_n \quad T^{\max}_n \\ 0; & \text{otherwise} \end{cases} \quad (12)$$

Therefore, the application hit ratio is expressed as

$$HR = \frac{\sum_{n=1}^{} M_n}{N} \quad (13)$$

The completion time of hit applications is expressed as

$$T = \sum_{n=1}^{N} M_n T^{fin}_n \quad (14)$$

The application hit ratio ranges from 0 to 1. The completion time of hit applications is large compared to it. As a result, the application hit ratio has little effect on the optimization goal. To improve the application hit ratio, we introduce a large variable $!$. The value of this variable is discussed in detail in the experimental section. In this case, this optimization objective is expressed as

$$\max_{X;Y} !HR \quad T \quad (15)$$

This optimization objective is not expressed as $\max_{X;Y} \frac{!HR}{T}$ for the following reasons. Assuming that there are three applications and their completion time are 0.2s, 0.3s, 0.5s, respectively. For simplicity, all applications are completed within their deadlines. If the first application is hit, the application hit ratio is expressed as $\frac{!HR}{T} = \frac{!=3}{0:2} = \frac{5!}{3}$. If the first and second applications are hit, $\frac{!HR}{T} = \frac{2=3!}{0:2+0:3} = \frac{4!}{3}$. If three applications are hit, $\frac{!HR}{T} = \frac{!}{0:2+0:3+0:5} = !$. Obviously, this value decreases as the number of hit applications increases. As a result, $\max_{X;Y} \frac{!HR}{T}$ does not meet the goal of improving the application hit ratio.

3.3.1 Service Constraint

Let c_k express the required storage size of service s_k. The service placement decisions are constrained by the storage capacity of edge servers, which is expressed as

$$\sum_{k=1}^{K} x_m^k c_k \quad C_m \tag{16}$$

3.3.2 Offloading Constraint

Each task can be offloaded to only one edge server that provides its required service. Let $M_{n;i}$ denote the set of edge servers that can process task $a_{n;i}$. Therefore, offloading a task needs to satisfy the constraint:

$$\sum_{m2M_{n;i}} y_{n;i}^m = 1 \tag{17}$$

3.3.3 Dependency Constraint

If there is an edge between tasks $a_{n;j}$ and $a_{n;i}$, task $a_{n;i}$ starts to be executed only when task $a_{n;j}$ is completed and the data transmitted to it. Therefore, dependent tasks need to meet the following constraint:

$$T_{n;i}^{wait;m} \quad T_{n;j}^{fin;y} + T_{j;i}^{tran} \tag{18}$$

As a result, the research problem is formulated as

$$\begin{aligned} P1: \max_{X;Y} \; & !HR \quad T \\ C1: \; & 0 < f_m \quad F_m \\ C2: \; & 0 \quad T_n^{fin} \quad T_n^{\max} \\ C3: \; & x_m^k; y_{n;i}^m \; 2 \; f0; 1g \\ & (16) \quad (18) \end{aligned} \tag{19}$$

Constraint $C1$ ensures that the computing power of each edge server does not exceed its maximum computing capacity. Constraint $C2$ guarantees that the completion time of each application does not exceed its deadline. Constraints $C3$ specify that service placement and task offloading decision variables are binary.

3.4 Problem Hardness

Theorem 1. *The P1 problem is an NP-hard problem.*

Proof. As discussed in [20], once a special case of the original problem is NP-hard, the original problem is also NP-hard. Motivated by this insight, we undertake a comparative analysis of the $P1$ problem with the well-known NP-hard Traveling Salesman Problem (TSP) [6]. By demonstrating that TSP is a special case of the $P1$ problem, we conclusively establish that the $P1$ problem is NP-hard.

The definition of TSP: Given a set of cities and the distance between every pair of cities, the problem is to find the shortest route on which each city is visited exactly once and return to the starting point.

We consider an arbitrary TSP instance with a set $C = f1; 2; :::; hg$ of h cities. The distance between each pair of cities $i; j \; 2 \; C$ is denoted by $d_{i;j}$. Now, we construct a special case of the $P1$ problem. It is assumed that there are h identical edge servers equipped with all required services, denoted by $E = f1; 2; :::; hg$, which are one-to-one correspondence with the cities in the set C. The available processing resources of edge servers are the same, denoted by a. An application includes h+1 tasks, denoted by $V = fv_1; v_2; :::; v_{h+1}g$, and them are offloaded onto these h edge servers. The DAG for tasks is $v_1 \; ! \; v_2 \; ! \; ::: \; ! \; v_{h+1}$. Task v_1 and task v_{h+1} require a=2 processing resources, respectively, while the other tasks require a processing resources. Obviously, tasks v_1 and v_{h+1} share one edge server while other tasks separately occupy an edge server. The execution time for any task on any edge server is t. The communication delay between edge servers $i; j \; 2 \; E$ is equal to the distance $d_{i;j}$ between cities i and j (i.e.,$T_{i;j}^{tran} = d_{i;j}$). All tasks can be offloaded when each edge server provides the services required by all tasks. At this point, the objective is converted to finding a feasible offloading solution that minimizes the completion time of applications within their deadlines. The wait time of task v_1 is 0 (i.e.,$T_1^{wait} = 0$), and the wait time of other tasks $v_i \; 2 \; V \quad fv_1g$ are expressed as $T_i^{wait} = T_{i\;1}^{wait} + t + T_{i\;1;i}^{tran}$. Therefore, the completion time of the application is denoted as

$$\begin{aligned} T^{fin} &= T_{h+1}^{wait} + t \\ &= T_h^{wait} + t + T_{h;h+1}^{tran} + t \\ &= T_h^{wait} + t + d_{h;h+1} + t \\ &= (h+1)t + d_{h;h+1} + ::: + d_{1;2} \end{aligned}$$

That means, for each task $v_i \; 2 \; V$, the selection of an edge server turns into the selection of a visited city. This is exactly the TSP instance. Therefore, each TSP instance is a special case of the $P1$ problem.

4 Method Design

Due to the NP-hard nature of the $P1$ problem, it is difficult to find an optimal solution. To address the problem, a Dependent task offloading and Service placement Optimization (DSO) method is proposed to seek the near-optimal solution. Fig 6 shows an overview of the DSO method. It consists of two steps. The first step is to deploy services on the edge servers based on service requests from mobile devices. The second step involves allocating tasks to the most suitable edge server, considering both the scheduling sequence and the services offered by the edge servers. The deployment of services inherently restricts task offloading to those edge servers that are capable of providing the services required by the tasks. These two steps are then iterated repeatedly until convergence.

Step 1: Firstly, edge servers collect services requested by mobile devices, and then determine the action spaces based on their storage capacities and the services. Secondly, the Q-table is constructed and initialized according to the action spaces and the number of edge servers. Thirdly, each edge

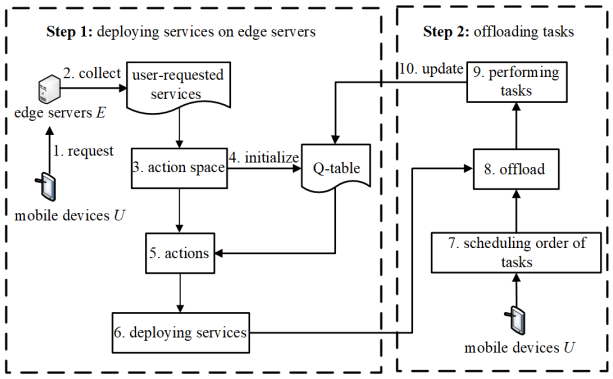


Fig. 6. The overview of the DSO method.

server selects an action from its action space or Q-table and then deploys the services corresponding to that action.

Step 2: Firstly, the scheduling order of tasks in the same application is determined by their dependencies. At the same time, the scheduling order of tasks in different applications is determined by their deadlines. Secondly, tasks are sequentially offloaded to appropriate edge servers based on their scheduling order and the specific services demanded. Thirdly, the corresponding reward is obtained after tasks are performed. Finally, the Q-table is updated based on this reward.

In order to understand the DSO method more clearly, we will describe the corresponding algorithms for **Step 1** and **Step 2** in the following section.

4.1 Service Placement Optimization

To achieve the **Step 1**, we adopt an IMAQL algorithm based on Q-learning techniques to solve the service placement problem. The framework of the IMAQL algorithm is shown in Fig. 7. Firstly, agents interact with the environment to obtain services requested by users to determine their action spaces. Here, agents refer to edge servers. Secondly, each agent uses the "-greedy policy to select an action from its action space or Q-table. To avoid getting into local optimum, each agent selects an action from its action space in the exploration phase. After sufficient exploration, the action with the largest Q-value is selected from the Q-table. Thirdly, agents receive the same reward r from the cooperative environment according to their actions. Finally, the Q-table will be updated if the reward r is greater than the Q-value of selected actions.

We explain the action space, reward function, Q-table and the "-greedy policy in the IMAQL framework below.

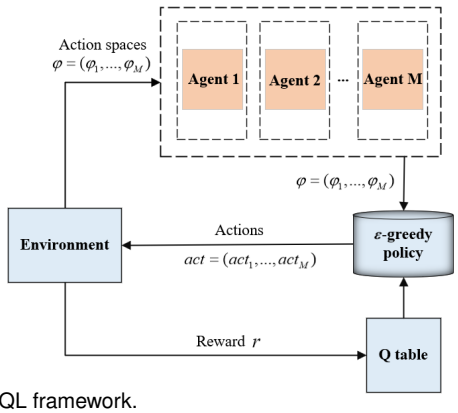


Fig. 7. IMAQL framework.

4.1.1 Action Space

Assuming that the number of edge servers and service types are M and K, respectively. According to the service placement variable x_m^k, the action space size of the service placement is $2^{M \quad K}$. As the number of edge servers and service types increases, the action space size of the service placement will increase exponentially. In order to improve the training efficiency of the algorithm, the following method is used to reconstruct a new action space.

Let a subset of services represent an action. All possible subsets of services on an edge server represent its action space. When a service is included in the subset, it is 1, otherwise, it is 0. $'_m$ represents the action space of edge server e_m. It consists of a series of actions, denoted as

$$'_m = f[s_m^1; \ldots; s_m^k; \ldots; s_m^K]g \tag{20}$$

An example is provided to illustrate the generation process of the action space on edge server e_m, as shown in TABLE 1. It is assumed that the storage size of the edge server is 8. There are five service types $s_1; s_2; s_3; s_4; s_5$ with storage sizes of 1, 2, 3, 3, 4, respectively. The subset is retained when the following two conditions are met. Firstly, the total storage size of services in this subset is no more than 8. Secondly, this subset is the largest subset of services. For example, the subset $[s_1; s_2; s_3]$ is retained rather than $[s_1]$, $[s_2]$, $[s_3]$, $[s_1; s_2]$, $[s_1; s_3]$, $[s_2; s_3]$. This is because each edge server can only accommodate a limited number of services and maximizing their utilization enables more applications to be supported.

TABLE 1
The Formation Process of the Action Space on Edge Server e_m

Service Combinations	Required Storage Size	Actions
(1) $s_1; s_2; s_3$	1+2+3=6	[1,1,1,0,0]
(2) $s_1; s_2; s_4$	1+2+3=6	[1,1,0,1,0]
(3) $s_1; s_2; s_5$	1+2+4=7	[1,1,0,0,1]
(4) $s_1; s_3; s_4$	1+3+3=7	[1,0,1,1,0]
(5) $s_1; s_3; s_5$	1+3+4=8	[1,0,1,0,1]
(6) $s_1; s_4; s_5$	1+3+4=8	[1,0,0,1,1]
(7) $s_2; s_3; s_4$	2+3+3=8	[0,1,1,1,0]

4.1.2 Reward Function

Generally, the agent's goal is to maximize the reward. Since this paper considers maximizing the optimization objective, the reward function can be expressed in terms of the optimization objective

$$r = !HR \quad T \tag{21}$$

4.1.3 Q-Table

The example is provided to illustrate the creation and initialization of the Q-table, as shown in TABLE 2. It is assumed that there exist three edge servers and three services with storage sizes of 3,1,2 and 1,1,1, respectively. From the *Action Space* part, the action spaces of edge servers are represented as $'_1 = f[1; 1; 1]g$, $'_2 = f[1; 0; 0]; [0; 1; 0]; [0; 0; 1]g$, $'_3 = f[1; 1; 0]; [1; 0; 1]; [0; 1; 1]g$, and their corresponding action numbers are 1,3,3, respectively. Thereby, the action index i can take the values 0, 1, 2, and the actions of edge servers are denoted as $'_1[0]$, $'_2[0]$; $'_2[1]$; $'_2[2]$, $'_3[0]$; $'_3[1]$; $'_3[2]$, respectively. Firstly, the Q-table is constructed with edge servers on the horizontal axis and action indexes on the vertical axis. Secondly, we proceed to initialize the Q-table. If edge server e_m has the action $[i]$, $Q(m; i) = 0$, otherwise,

$Q(m;i) = 1$. Thirdly, each edge server e_m selects an action $[i]$, and the reward r is obtained by performing all actions selected by edge servers. Finally, the $Q(m;i)$ will be updated, if $Q(m;i) > r$.

TABLE 2
Q-Table

Edge Servers	Action Indexes		
	0	*1*	*2*
edge server 1	0	-1	-1
edge server 2	0	0	0
edge server 3	0	0	0

4.1.4 "-Greedy Policy

We introduce an exploitation-exploration method. The exploitation process selects the action with the highest Q-value from the Q-table. The exploration process selects an action at random from the action space. If the exploitation processes are too extensive, the model will be prone to fall into local optimization. If there are too many exploration processes, the model will converge too slowly. Consequently, we need to weigh exploitation against exploration processes. A variable $b\ 2\ [0;1]$ is randomly generated. If $" < b$, the agent will enter the exploration process, otherwise, it enters the exploitation process. In the training process, $"$ is set to 0 to allow the agent to explore fully. Then $"$ gradually increases so that the agent slowly selects the action with the highest Q-value. The incremental and maximum values of this variable are discussed in detail in the experimental section.

The IMAQL algorithm is detailed in Algorithm 1. Firstly, each edge server uses the "-greedy policy to select its action from its action space or Q-table (lines 2-4). Secondly, edge servers deploy services according to the chosen actions, and receive the same reward r from the cooperative environment after tasks are performed (line 5). Thirdly, the Q-table will be updated if reward r is greater than the Q-value of selected actions (lines 6-10). Finally, the above is repeated until the number of iterations is completed (lines 1-11).

Algorithm 1 IMAQL Algorithm

Input:
action spaces, Q-table
Output:
service placement policy X

```
1:  for episode = 1 to EPISODE do
2:    for each edge server e_m 2 E do
3:      selects an action ' _m[i] based on the "-greedy policy
4:    end for
5:    edge servers obtain the reward r based on their actions
6:    for each edge server e_m 2 E do
7:      if r >Q(m; i) then
8:        Q(m; i) = r
9:      end if
10:   end for
11: end for
```

4.2 Task Offloading Optimization

To achieve the **Step 2**, we design a greedy algorithm to settle the task offloading problem. It mainly solves the following issues. Firstly, the scheduling order of tasks in the same application needs to be determined to ensure their dependencies. Secondly, the scheduling order of tasks from different applications requires to be determined to satisfy time constraints. Thirdly, each task should be offloaded to the most appropriate edge server according to the scheduling order. Below we describe how to solve these problems.

4.2.1 Task Ranking

In order to efficiently schedule dependent tasks in the same application, their order should be determined. There have been some classic attributes for task ranking in the literature [25], such as top level and bottom level. The top level of task $a_{n;i}$ is defined as the critical path from the dummy task $a_{n;0}$ to task $a_{n;i}$ (excluding $a_{n;i}$). The critical path signifies the longest path length from the start point to the end point within the DAG. The length of a specific path is calculated by the sum of the computation cost on each task node and the communication cost on each directed edge over the path. The bottom level of task $a_{n;i}$ is the length of the critical path from dummy task $a_{n;I+1}$ to task $a_{n;i}$.

We use the top level and the bottom level respectively to calculate the ranking of each task in the application u_n, i.e.,

$$TL_{a_{n;i}} = \begin{cases} 0; & a_{n;i} = a_{n;0} \\ \max\limits_{a_{n;k} 2 pre(a_{n;i})} \{TL_{a_{n;k}} + \frac{!_{n;k}}{f} + \frac{d^n_{k;i}}{R}\}; & \text{otherwise} \end{cases} \quad (22)$$

$$BL_{a_{n;i}} = \begin{cases} 0; & a_{n;i} = a_{n;I+1} \\ \max\limits_{a_{n;k} 2 suc(a_{n;i})} \{BL_{a_{n;k}} + \frac{d^n_{i;k}}{R} + \frac{!_{n;i}}{f}\}; & \text{otherwise} \end{cases} \quad (23)$$

where $suc(a_{n;i})$ denotes the set of immediate successors of task $a_{n;i}$. f and R denote the average processing power and transmission rate of all edge servers, respectively.

The DAG shown in Fig. 8 is taken as an example to determine the scheduling order of tasks in the same application. The node weights represent the total CPU cycles required by tasks, and edge weights represent the amount of data transferred between tasks. We leverage the parameter settings in TABLE 3 to randomly generate the weights and processing capacities of edge servers within the prescribed parameter boundaries. Additionally, the transmission rates between edge servers are also specified. Following this, we compute the average processing power as 8 GHz and the average transmission rate as 20 Mbps. When applying Equation (22), the calculated values for the tasks are sequentially 0, 0.02, 0.03, 0.05, 0.095, 0.09, and 0.16, while those in Equation (23) are sequentially 0.16, 0.095, 0.13, 0.0475, 0.065, 0.0475, and 0. The priority of tasks is sorted by a non-ascending order of their values in the bottom level, while the order is reversed in the top level. Consequently, the scheduling order of tasks in the bottom level is $a_{n;0}$, $a_{n;2}$, $a_{n;1}$, $a_{n;4}$, $a_{n;3}$, $a_{n;5}$, $a_{n;6}$. The scheduling order of tasks in the top level is $a_{n;0}$, $a_{n;1}$, $a_{n;2}$, $a_{n;3}$, $a_{n;5}$, $a_{n;4}$, $a_{n;6}$. The critical path is represented by red lines, where $a_{n;0}$, $a_{n;2}$, $a_{n;4}$ and $a_{n;6}$ are the tasks on the critical path. Obviously, the scheduling order determined by the bottom level can satisfy the requirement that tasks on the critical path are executed firstly. For example, task $a_{n;2}$ is executed before task $a_{n;1}$ in the bottom level, but vice versa in the top level. Since task $a_{n;2}$ is on the critical path, it directly impacts the overall completion time of application u_n. As a result, we use the bottom level method to obtain the scheduling order of tasks within the same application.

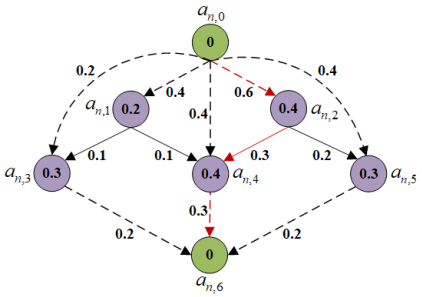


Fig. 8. DAG instance. The node weights represent the total CPU cycles required by tasks. The edge weights represent the amount of data transferred between tasks. The red lines represent the critical path.

4.2.2 Ready List

In order to allow more applications to be completed within their time constraints, the scheduling order of tasks from different applications should be determined. We introduce a list called *readyList*. According to the task ranking, the task is put into the list when all its immediate predecessors have been completed. There may be a number of tasks belonging to different applications in this list. They are sorted in a non-descending order by applications' deadlines.

4.2.3 Edge Server Selection

Based on the scheduling order of tasks in the *readyList*, each task is offloaded sequentially to the most appropriate edge server for execution, which provides the required service and has the shortest completion time.

Algorithm 2 Greedy Algorithm

Input:
service placement policy X
Output:
task offloading policy Y
1: **Initialization** *readyList* = ;
2: **for** each mobile device u_n 2 U **do**
3: obtains its task ranking using the bottom level
4: **end for**
5: **for** each mobile device u_n 2 U **do**
6: tasks without immediate predecessors are put into *readyList*
7: **while** *readyList* 6 ; **do**
8: tasks in *readyList* are sorted by a non-descending order of applications' deadlines
9: **for** each task $a_{n;i}$ 2 *readyList* **do**
10: is offloaded to the edge server with minimum completion time and removed from *readyList*
11: **if** all immediate predecessors of task $a_{n;j}$ 2 $suc(a_{n;i})$ have been completed **then**
12: $a_{n;j}$! *readyList*
13: **end if**
14: **end for**
15: **end while**
16: **end for**

The greedy algorithm is detailed in Algorithm 2. Firstly, the scheduling order of tasks in each application is obtained by using the bottom level (lines 2-4). Secondly, according to the task ranking, tasks whose immediate predecessors have been all completed are put into the *readyList* (line 6). Subsequently, tasks in the list are sorted by a non-descending order of applications' deadlines (line 8). Thirdly, each task in this list is offloaded to the edge server with the shortest completion time and then removed from this list (line 10). Finally, this list is updated until all tasks are scheduled (lines 11-13).

5 Performance Evaluation

The experiments are conducted on a computer with the following configuration: Intel(R) Core(TM) i5-1240P CPU @ 1.70 GHz, 16 GB RAM and Windows 11 OS. All the experimental models are developed in Python 3.9.

5.1 Research Questions

We conduct extensive simulation experiments to explore the following research questions.

Q1: What are the best parameters for the IMAQL algorithm model training?
Q2: How do parameters such as the value of variable ! and the number of collaborating edge servers affect the optimization performance?
Q3: What is the performance of the proposed method compared to other methods?

5.2 Simulation Settings

5.2.1 Data Set Description

The experimental data in this paper are mainly from two datasets. The first is the EUA dataset [26], containing the location distribution of edge servers and users. The second is the dataset of industry clusters made public by Alibaba in 2018 [7], containing the DAG structure of dependent tasks.

The first dataset includes the geographical locations of 125 edge servers and 816 users (e.g., longitude and latitude) provided by the Australian Communications and Media Authority. In the experiment, we select a 6.2 km^2 CBD area in Melbourne, Australia [27], and set the coverage radius of each base station to be uniformly distributed within [150, 400] meters [28]. The second dataset includes 3,175,025 types of batch workloads, with 2,031,910 of these batch jobs composed of tasks that exhibit dependencies. The number of tasks in a job ranges from 2 to 142. In the experiment, we select an appropriate number of jobs to represent the applications, where the task dependencies in a job are used to represent the task dependencies in an application.

5.2.2 Parameters Settings

Similarly to [24], there are five types of tasks requesting services. Meanwhile, the uplink channel gain $h_{n;m} = (dist_{n;m})$, where $dist_{n;m}$ is the distance between mobile device u_n and edge server e_m, and the path loss factor is set to 4. We set the values of the following parameters by referring to [24], [29], [30], which is shown in TABLE 3.

5.2.3 Communication Delay

In the experimental setup, we incorporate the communication delay equations outlined in Section 3.2.1 to accurately model the behavior of the system. Specifically, during the simulation of task transmission from a mobile device to an edge server, we utilize Equation (3) to quantify the uplink transmission delay, which includes the forwarding time from the mobile device to the indirectly connected edge server. Furthermore, when tasks disseminate their computation results to dependent tasks within the system, we apply Equation (5) to calculate the transmission delay

TABLE 3
Simulation Parameters

Parameters	Value
The input data size of each task	[0.2, 1] Mb
The size of data transferred between tasks	[0.1, 0.4] Mb
The CPU cycles of each task	[0.1, 0.5] GHz
The storage size of each service	[1, 5] Mb
The total storage capacity of each edge server	10 Mb
The maximum processing power of each edge server	10 GHz
The transmission rates between edge servers	20 Mbps
W^{up}	20 MHz
W^{down}	100 MHz
^{2}	50 dBm
p_n	23 dBm
p_m	30 dBm
T_n^{max}	[0.5, 1] s

between them. This includes the delay between virtual tasks and real tasks to facilitate the representation of data inputs and execution results outputs. After scheduling all tasks within a DAG, the execution outcomes of offloaded tasks are transmitted back to the original mobile device. For this transmission, we employ Equation (4) to calculate the delay, which takes into account the forwarding time from the indirectly connected edge server to the mobile device.

5.2.4 Wait Delay

In the experimental setup, we use the wait delay equations outlined in Section 3.2.3 to quantify the wait time a task experiences before executing on the edge server. Specifically, when a task receives execution results from all its immediate predecessors, we use Equation (7) to calculate this part of the wait delay, which encapsulates the completion time of precursor tasks and inter-task communication delays. Furthermore, due to the serial execution nature of the edge server, the task must wait for all preceding tasks in the queue to complete before it can commence execution. We use Equation (8) to calculate this queue waiting delay. Combining the above two parts of delay (i.e. the delay in receiving results from precursor tasks and the queue wait delay), we use Equation (9) to calculate the total wait time a task experiences from the moment it is ready to execute until it receives a processing opportunity.

5.3 Comparison Methods

To verify the performance of our proposed method, we introduce the following four methods for comparison.

Q-Learning algorithm + Greedy algorithm (QLG). The method leverages the Q-learning algorithm to place services, while employing our proposed greedy algorithm for task offloading. In this method, the Q-learning algorithm identifies the action with the highest Q-value from the Q-table with a 90% chance, and randomly explores other actions from the action space with the remaining probability.

Multi-Agent Deep Deterministic Policy Gradient (MADDPG) algorithm [31] + Greedy algorithm (MADDPGG). The method uses the MADDPG algorithm to place services, and adopts our proposed greedy algorithm to offload tasks.

IMAQL algorithm + Convex Programming (CP) algorithm [6] (ICP). The method adopts the IMAQL algorithm for service placement and employs the CP algorithm for task offloading. The CP algorithm reformulates the task offloading into a convex problem and resolves it through random rounding. Subsequently, tasks are offloaded based on their scheduling order determined by the obtained solutions.

MSO method [22]. This method adopts the IMAQL algorithm to place services, and utilizes the load-balancing algorithm to offload independent tasks.

5.4 Statistical Validation

To ensure the robustness and credibility of our experimental findings, we implement the hypothesis testing for simulation results, which is an important tool in statistical analysis.

The primary hypothesis of this study examines whether there exists a statistically significant difference in the impacts of our method and the comparative methods on rewards and application hit ratios. Firstly, we collect data from two distinct methods, such as the rewards generated by our method and the QLG method. To reduce the influence of randomness, each experiment is conducted 30 times, assigning a unique random seed for each iteration. Secondly, leveraging the central limit theorem, we use Python software to compute the mean values and standard deviations for the collected data. Subsequently, the two-independent sample t-test method is used to calculate the t-statistic (t-value). Thirdly, following the computation of the t-statistic, the corresponding p-value is calculated using Python's scipy library. By setting the significance level at $= 0.05$, we determine the critical value for rejecting the null hypothesis. If the computed p-value is less than the significance level of 0.05, we have sufficient statistical evidence to reject the null hypothesis, which states that there is no significant difference in the effects of our method and the QLG method on rewards. This outcome supports our alternative hypothesis, indicating their significant differences in enhancing reward effects. Finally, we delve into the magnitude of this difference, estimating the effect size using Cohen's d. This metric provides a quantitative assessment of the size of the difference between the two sample sets.

5.5 Parameter Analysis

5.5.1 The Effect of the Increment of "

In the *"-Greedy Policy* section, the introduction of the variable " aims to mitigate the risk of the model getting trapped in a local optimum and experiencing sluggish convergence. During the training process, " is initially set to 0, enabling agents to fully explore their action spaces. Subsequently, " is gradually increased, causing agents to slowly select actions from the Q-table. The determination of the suitable increment and maximum value for " is crucial. In this section, we discuss the increment of " and assume that the maximum of " is 1. In the experiment, we introduce an enumerate method that explores all possible service placement policies and chooses the most optimal one. Since the time complexity of this algorithm is too high, we use it as a criterion to evaluate the performance of our proposed method.

To ensure clarity and conciseness in the presentation, Fig. 9 integrates both the average reward and the application hit ratio into a single graph featuring two distinct y-axes: the left axis represents the average reward, while the right axis corresponds to the average application hit ratio. As

evident from Fig. 9, with the increment of ", the average reward and application hit ratio of our method exhibit a consistent decrease. This is because a large increment in " hinders agents' ability to adequately select actions from their action spaces, which may result in the model getting trapped in a local optimum. Obviously, when the increment of " is set to 0.0001, the average reward and application hit ratio closely resemble those achieved by the enumerate method. We employ statistical validation to substantiate our conclusion. With regards to the average reward, the computed t-statistic is 0.71, accompanied by a p-value of 0.24, which exceeds the significance level of 0.05. This indicates that the observed difference is not statistically significant. Consequently, the appropriate increment of " should be set to 0.0001 for subsequent experiments.

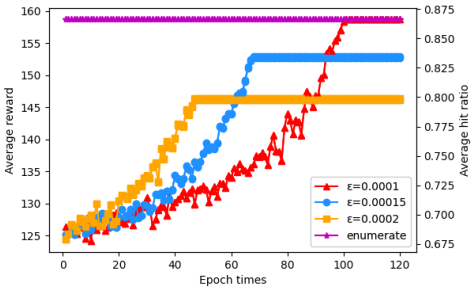


Fig. 9. The effect of the increment of " on average reward and application hit ratio.

5.5.2 The Effect of the Maximum of "

In this section, we discuss the maximum of ". Similarly, Fig. 10 also integrates the average reward and application hit ratio into a single graph with dual y-axes. As evident from Fig. 10, as the maximum of " increases, the average reward and application hit ratio of our method exhibit a corresponding upward trend. This is because when the maximum of " is set to a lower values, agents tend to select actions with the largest Q-value from the Q-table earlier, resulting in a lower value for model convergence. For instance, if the maximum of " is set to 0.9, agents will begin selecting actions solely from the Q-table once " reaches this threshold. Notably, when the maximum of " is set to 1, the average reward and application hit ratio closely align with those achieved by the enumerate method. Regarding the average reward specifically, statistical analysis yields a t-statistic of 1.06 and a corresponding p-value of 0.147, which surpasses the conventional significance threshold of 0.05. Consequently, in subsequent experiments, we establish the maximum of " as 1 to guarantee optimal performance.

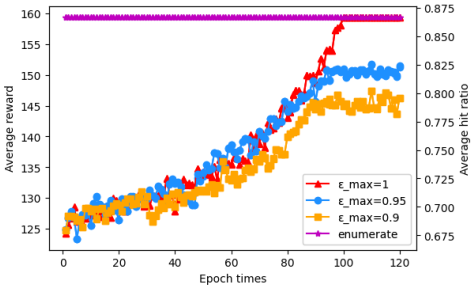


Fig. 10. The effect of the maximum of " on average reward and application hit ratio.

5.5.3 The Effect of the Variable !

Equation (13) shows that the application hit ratio varies between 0 and 1. Notably, the completion time of hit applications is larger compared to this hit ratio. As a result, the application hit ratio has minimal influence on the experimental outcomes. However, our goal is primarily to elevate the application hit ratio, followed by minimizing the completion time of hit applications. We propose the method in (15) to improve the application hit ratio, where the value of the variable ! needs to be determined. Let $!HR{=}T$ represent the difference between $!HR$ and T. As shown in Fig. 11, the difference of all methods increases as the value of ! increases. Specifically, when the value of ! equals 25, $!HR{=}T$ 1:1. We set the value of the variable ! to 200 so that $!HR >> T$. Consequently, the application hit ratio is increased by 200 times in subsequent experiments.

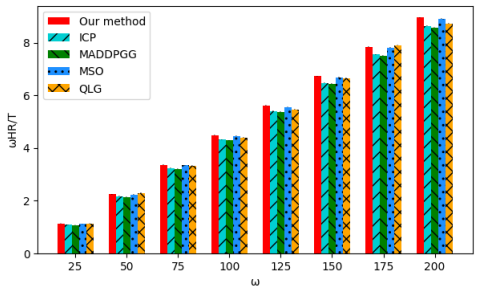


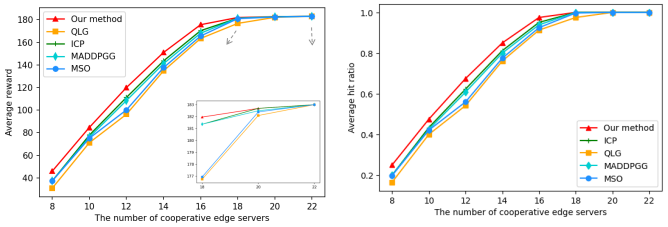
Fig. 11. The effect of ! on !HR/T.

5.5.4 The Effect of the Number of Cooperative Edge Servers

In the experiment, we set the number of applications to 40, and each application includes 5 to 10 tasks. As illustrated in Figs. 12(a) and 12(b), as the number of cooperative edge servers increases from 8 to 18, there is a noticeable surge in both the average reward and application hit ratio for all methods. The reasons are as follows. With more edge servers available, a higher number of tasks can be processed. Additionally, tasks are more likely to be offloaded to the edge server with the shortest completion time, thereby minimizing the overall completion time. As the number of cooperative edge servers increases from 18 to 22, the average application hit ratio for the QLG and MSO methods experiences a slight increase before plateauing, whereas the other methods remain unchanged at 1. However, the average reward for all methods experiences a slow increase. This is because the completion time of the hit applications has a lesser impact on the experimental results, causing the average reward to increase slowly. Logically, the number of cooperative edge servers should be 18. In view of the little difference in performance among the methods with 18 cooperative edge servers, we choose to decrease the number of collaborative edge servers to 14 for subsequent experiments. This adjustment is aimed at enhancing the discriminability of performance among the different methods.

While we do not conduct standalone experiments for the wait delay model, the experiment implicitly validates the model's effectiveness. For instance, in Fig. 12(b), as the number of cooperative edge servers increases, more resources are available for task processing, resulting in reduced waiting time in queues, reflected in a higher application hit ratio.

This observation aligns with our wait delay model's prediction that queuing time decreases with increased resources.



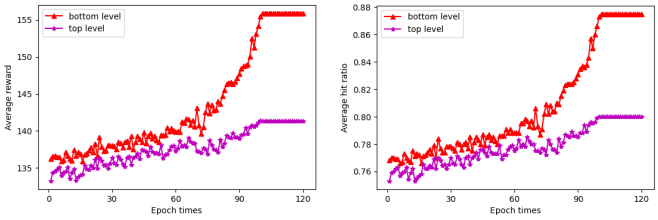
(a) Average reward (b) Average application hit ratio

Fig. 12. The effect of the number of cooperative edge servers on average reward and application hit ratio.

5.6 Experimental Results

5.6.1 The Effect of Task Scheduling Order

In the *Task Ranking* section, we introduce two approaches for determining the scheduling order of dependent tasks, and illustrate the superiority of the bottom level method over the top level approach through an example. In this section, our goal is to meticulously validate this finding through rigorous experiments. As shown in Figs. 13(a) and 13(b), with an increasing number of epochs, both the top level and bottom level schemes exhibit convergence towards a stable performance. Notably, the results underscore the superiority of the b-level scheme over the t-level method, as evidenced by a 10.3% higher average reward and a notable 9.4% improvement in the average application hit ratio. Furthermore, in average reward, their t-statistic is 25.26, with a p-value $<$ 0.001, indicating highly significant differences. The Cohen's d value of 0.326 suggests a medium effect size, indicating a notable superiority of b-level over the t-level.

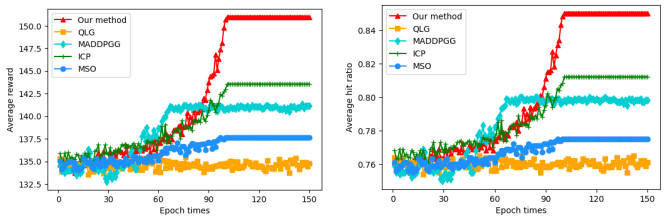


(a) Average reward (b) Average application hit ratio

Fig. 13. The performance comparison of task scheduling schemes.

5.6.2 The Convergence of Methods

In the experiment, 150 epochs are executed, each encompassing 100 episodes, with the epoch's value derived from the average outcome across these episodes. As illustrated in Fig. 14(a), our proposed method, ICP and MSO methods converge to a stable performance after approximately 100 epochs. Notably, our method achieves a 5.2% improvement in average reward compared to the ICP method, with statistical significance supported by the p-value of 0.03 and Cohen's d of 0.26. This suggests that our proposed greedy algorithm enhances the application hit ratio relative to the CP task offloading algorithm. Furthermore, our method outperforms the MSO method by 9.7% in average reward, with a highly significant p-value of less than 0.001, emphasizing the impact of optimal task scheduling order on improving the hit ratio. The MADDPGG method converges earlier but levels off at a lower performance. Additionally, the QLG method exhibites fluctuating results due to the random nature of action selection. By observing the trends, it gets stuck in a local optimum as a result of insufficient action space exploration. Our method exhibits the improvement in average reward, surpassing MADDPGG by 7% and QLG by 12.1%, respectively. This advantage is robustly confirmed through statistical analysis, with p-values of 0.004 and $<$0.001, accompanied by substantial effect sizes (Cohen's d = 0.37 and 1.05, respectively). These results underscore the advantages of our IMAQL algorithm compared to the service placement algorithms, MADDPG and Q-learning.

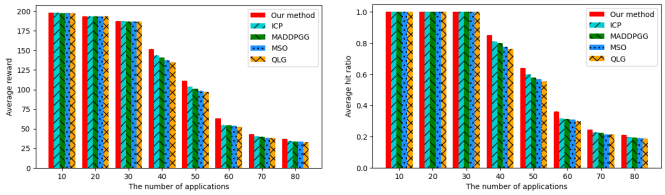


(a) Average reward (b) Average application hit ratio

Fig. 14. The convergence of methods on average reward and application hit ratio.

5.6.3 The Effect of the Number of Applications

As the number of applications increases from 10 to 30, the average application hit ratio for all methods keeps 1 in Fig. 15(b). However, Fig. 15(a) reveals a steady decline in the average reward for all methods. This decline is attributed to the escalating processing requirements of tasks, coupled with the serial processing nature of edge servers, which collectively lead to longer task completion time. As the number of applications increases from 30 to 80, both the average reward and application hit ratio decline consistently for all methods, as evidenced in Figs. 15(a) and 15(b). This decline stems from an increased number of applications failing to meet deadlines, lowering the application hit ratio. When the number of applications is small, our method displays no obvious advantage compared with the other four methods. Nevertheless, as the number of applications increases, our method distinctly surpasses them, with statistical significance confirmed by p-values from the t-test consistently below 0.001 and substantial effect sizes exceeding 0.8, firmly underpinning the superiority of our approach. This advantage arises from two key factors: our method ensures priority execution for tasks on the critical path, expediting overall application completion, and our service deployment strategy closely resembles the solution of the enumerate method, which can be proven in Section 5.5.1, yielding a superior application hit ratio compared to alternative approaches.

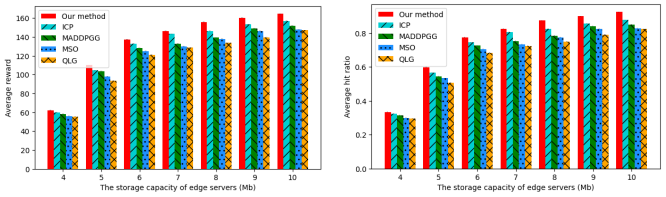


(a) Average reward (b) Average application hit ratio

Fig. 15. The effect of the number of applications on average reward and application hit ratio.

5.6.4 The Effect of the Storage Capacity of Edge Servers

As the storage capacity of edge servers increases from 4 to 10, all methods exhibit an upward trend in both average reward and application hit ratio, as illustrated in Figs. 16(a) and 16(b). This is attributed to the increasing capacity of edge servers to accommodate more services, resulting in more tasks to be executed. Notably, the proposed method outperforms its counterparts due to its more effective service deployment strategy. In the Section 5.5.1, we conducted a comparative experiment, revealing that the proposed service deployment approach closely aligns with the policy derived from the enumerate method. Furthermore, all p-values are below 0.001 from the t-test and Cohen's d values exceed 0.8, underscoring the statistical significance of our findings. Specifically, as shown in Fig. 16(b), when the storage capacity of edge servers reaches 10, the average application hit ratio for our method is higher than that of the ICP, MADDPGG, MSO and QLG methods by about 5.8%, 8.8%, 11.7% and 12.3%, respectively.



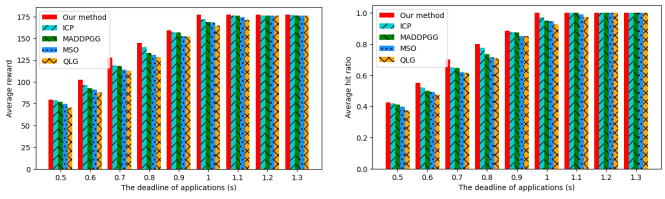
(a) Average reward (b) Average application hit ratio

Fig. 16. The effect of the storage capacity of edge servers on average reward and application hit ratio.

5.6.5 The Effect of the Deadline of Applications

As the deadline of applications increases from 0.5 to 1, the average reward and application hit ratio for all methods exhibit an upward trend, as depicted in Figs. 17(a) and 17(b). This is because one of the conditions for an application to be hit is to be completed within its deadline. Therefore, as the deadline for applications is extended, the likelihood of an application being hit also increases, as it provides applicants with additional time to meet this crucial criterion. As the deadline of applications increases from 1 to 1.2, the average reward and application hit ratio for all methods exhibit a sluggish increase or remain unchanged. This is primarily because the majority of applications have been processed within the deadline of 1. For instance, Fig. 17(b) reveals that when the deadline of applications is 1, the average application hit ratio for our proposed method, ICP, MADDPGG, MSO and QLG methods are 1, 0.97, 0.95, 0.946 and 0.925 respectively. As the deadline of applications increases to 1.2, the average application hit ratio for all methods reaches 1. However, interestingly, even when the storage capacity of each edge server is maximized, the average application hit ratio in Fig 16(b) does not reach 1. This observation underscores the significant influence of the application deadline on the application hit ratio.

When all the applications are hit, (i.e., application hit ratio reaches 1), the application completion time for each method can be determined based on the reward and application hit ratio. Specifically, the application completion time for our proposed method, ICP, MADDPGG, MSO and QLG methods are 23, 23.79, 24.08, 24.1 and 24.13, respectively.



(a) Average reward (b) Average application hit ratio

Fig. 17. The effect of the deadline of applications on average reward and application hit ratio.

6 Related Work

In recent years, task offloading has been a popular research topic in edge computing. Many scholars have proposed various task offloading methods and have achieved satisfactory research results. We analyze some of the studies that are closely related to our research below.

Feng *et al*. [4] investigated task partitioning and user association problems to minimize the average latency. They proposed the dual decomposition and matching methods to obtain near-optimal solutions for user association. Meanwhile, the task partitioning problem was solved by mathematical calculation. Wang *et al*. [32] studied the joint task offloading, power assignment and resource allocation problem. They developed an evolutionary algorithm to minimize the response time, energy consumption and cost. Jiang *et al*. [33] leveraged the game theory to determine the optimal task offloading strategy for improving the quality of service. Meanwhile, they applied a reinforcement learning to realize the dynamic resource allocation of edge servers. Chen *et al*. [34] investigated resource allocation and task offloading problems to minimize the energy consumption. The problem was solved by using the multi-agent deep deterministic policy gradient reinforcement learning algorithm. Liu *et al*. [35] proposed a semidefinite relaxation approach with an adaptive adjustment procedure to minimize the weighted sum of execution time and computation cost of all tasks.

The aforementioned works focus on offloading independent tasks. Considering the increasing complexity of applications, an application usually consists of dependent tasks. Wang *et al*. [14] investigated the dependent task offloading problem optimizing the latency and energy consumption. An offloading scheme embedded with deep reinforcement learning and inference procedures was proposed to adapt dynamic scenarios. Liu *et al*. [15] investigated the dependent task offloading problem to reduce the application execution delay. A heuristic algorithm was designed to solve the research problem. Li *et al*. [36] proposed a deep reinforcement learning based on dependent task offloading scheme to minimize the average cost of energy and time. Ming *et al*. [37] proposed a heuristic scheme to iteratively optimize the delay of processing a series of tasks with dependencies under the constraints of edge servers. Zhou *et al*. [38] proposed an improved non-dominated sorting genetic algorithm-II to minimize the latency and energy cost.

The above studies assume that each edge server can perform any type of offloaded task. In fact, deploying all types of services at each edge server is challenging due to limited storage resource. To this end, the above issues are addressed by considering the impact of service placement. Zhao *et al*. [6] studied the offloading of dependent tasks

to edge servers under limited service caching. A convex programming algorithm was proposed to minimize the makespan of applications. Bi *et al*. [13] considered a single edge server that assisted a user in executing a sequence of computation tasks. An alternating minimization technique was devised to minimize the computation delay and energy consumption. Shen *et al*. [29] studied the jointly problem of dependency-aware task offloading and service caching, and proposed a semi-distributed algorithm based on dynamic programming to maximize the offloading efficiency. Zhang *et al*. [39] studied the dependent task offloading problem with limited service caching constraints. A cloud-edge-device collaborative algorithm was proposed to minimize the completion time of applications. Oskoui *et al*. [40] designed a distributed method based on deep reinforcement learning to optimize the completion time of applications by offloading dependent tasks to edge servers.

The above research overlooks the application hit ratio, which is crucial as applications become more complicated, containing multiple interdependent tasks. Nevertheless, some studies have addressed this issue. Chen *et al*. [21] investigated the joint caching and service placement problem to maximize the reward relevant to the number of accepted applications and corresponding service latency. Our previous work investigated the service placement and multi-task offloading problems to maximize application hit ratios [22]. Nevertheless, a notable limitation in the above studies is the neglect of task dependency. Liao *et al*. [23] optimized the application assigning and scheduling problems to ensure that more applications are completed before their deadlines. Nevertheless, they overlooked the influence of service placement. Addressing the above limitations, this paper investigates dependent task offloading and service placement problems to maximize the application hit ratio.

7 Conclusion and Future Work

In this paper, we investigate service placement and dependent task offloading problems to maximize the application hit ratio. The DSO method is proposed to address this research problem. Firstly, it utilizes the IMAQL algorithm to obtain the optimal service placement strategy. Secondly, tasks are greedily offloaded to the appropriate edge servers according to their scheduling order. Finally, sufficient experimental results coupled with rigorous statistical validation demonstrate that the DSO method surpasses other methods in performance. However, the escalating complexity of edge computing environments, driven by the proliferation of edge servers and diverse service types, poses challenges. Specifically, the Q-table size significantly expands, lengthening the reinforcement learning training phase. This restricts our method's scalability in large-scale environments. To mitigate this, we envision incorporating advanced reinforcement learning techniques, such as deep and hierarchical methods, to enhance scalability and accelerate convergence.

Acknowledgments

This work was supported in part by the National Natural Science Foundation of China (Nos. 62472147, 62272145, 62072159 and U21B2016), and partially by the Australian Government through the Australian Research Council's Discovery Projects funding scheme (project DP220101823).

References

[1] J. Chen, Y. Yang, C. Wang, H. Zhang, C. Qiu, and X. Wang, "Multitask offloading strategy optimization based on directed acyclic graphs for edge computing," *IEEE Internet of Things Journal*, vol. 9, no. 12, pp. 9367–9378, 2021.

[2] G. Yang, L. Hou, X. He, D. He, and S. Chan, "Offloading time optimization via markov decision process in mobile-edge computing," *IEEE internet of things journal*, vol. 8, no. 4, pp. 2483–2493, 2020.

[3] Z. Ning, P. Dong, X. Wang, X. Hu, J. Liu, and L. Guo, "Partial computation offloading and adaptive task scheduling for 5g-enabled vehicular networks," *IEEE Transactions on Mobile Computing*, vol. 21, no. 4, pp. 1319–1333, 2020.

[4] M. Feng, M. Krunz, and W. Zhang, "Joint task partitioning and user association for latency minimization in mobile edge computing networks," *IEEE Transactions on Vehicular Technology*, vol. 70, no. 8, pp. 8108–8121, 2021.

[5] 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.

[6] G. Zhao, H. Xu, Y. Zhao, C. Qiao, and L. Huang, "Offloading tasks with dependency and service caching in mobile edge computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 11, pp. 2777–2792, 2021.

[7] J. Zhang, G. Zhang, X. Bao, C. Ding, P. Yuan, X. Zhang, and S. Wang, "Dependent application offloading in edge computing," *IEEE Transactions on Cloud Computing*, 2023.

[8] X. Li, T. Chen, D. Yuan, J. Xu, and X. Liu, "A novel graph-based computation offloading strategy for workflow applications in mobile edge computing," *IEEE Transactions on Services Computing*, vol. 16, no. 2, pp. 845–857, 2022.

[9] H. Zhang, Y. Yang, B. Shang, and P. Zhang, "Joint resource allocation and multi-part collaborative task offloading in mec systems," *IEEE Transactions on Vehicular Technology*, vol. 71, no. 8, pp. 8877–8890, 2022.

[10] D. Zhang, W. Dong, T. Zhang, J. Zhang, P. Zhang, and G. Sun, "New computing tasks offloading method for mec based on prospect theory framework," *IEEE Transactions on Computational Social Systems*, 2022.

[11] H. Xiao, J. Huang, Z. Hu, M. Zheng, and K. Li, "Collaborative cloud-edge-end task offloading in mec-based small cell networks with distributed wireless backhaul," *IEEE Transactions on Network and Service Management*, 2023.

[12] S. Lin, K. C. Chen, and A. Karimoddini, "Sdvec: software-defined vehicular edge computing with ultra-low latency," *IEEE Communications Magazine*, vol. 59, no. 12, pp. 66–72, 2021.

[13] S. Bi, L. Huang, and Y. A. Zhang, "Joint optimization of service caching placement and computation offloading in mobile edge computing systems," *IEEE Transactions on Wireless Communications*, vol. 19, no. 7, pp. 4947–4963, 2020.

[14] J. Wang, J. Hu, G. Min, W. Zhan, A. Y. Zomaya, and N. Georgalas, "Dependent task offloading for edge computing based on deep reinforcement learning," *IEEE Transactions on Computers*, vol. 71, no. 10, pp. 2449–2461, 2021.

[15] 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*, vol. 22, no. 4, pp. 2147–2162, 2023.

[16] L. Chen, J. Wu, J. Zhang, H. Dai, X. Long, and M. Yao, "Dependency-aware computation offloading for mobile edge computing with edge-cloud cooperation," *IEEE Transactions on Cloud Computing*, vol. 10, no. 4, pp. 2451–2468, 2022.

[17] J. Deng, B. Li, J. Wang, and Y. Zhao, "Microservice pre-deployment based on mobility prediction and service composition in edge," in *Proceedings of the IEEE International Conference on Web Services (ICWS)*, 2021, pp. 569–578.

[18] C. Chen, B. Bhargava, V. Aggarwal, B. Tonshal, and A. Gopal, "A hybrid deep reinforcement learning approach for jointly optimizing offloading and resource management in vehicular networks," *IEEE Transactions on Vehicular Technology*, pp. 1–12, 2023.

[19] Y. Zhang, Y. Zhou, S. Zhang, G. Gui, B. Adebisi, and H. Gacanin, "An efficient caching and offloading resource allocation strategy in vehicular social networks," *IEEE Transactions on Vehicular Technology*, pp. 1–13, 2023.

[20] H. Xiao, C. Xu, Y. Ma, S. Yang, L. Zhong, and G. Muntean, "Edge intelligence: A computational task offloading scheme for depen-