Hindawi
Wireless Communications and Mobile Computing
Volume 2021, Article ID 3877285, 15 pages
https://doi.org/10.1155/2021/3877285

Research Article

TPD: Temporal and Positional Computation Offloading with Dynamic and Dependent Tasks

Mingzhi Wang,[1] Tao Wu,[1] Xiaochen Fan,[2] Penghao Sun,[3] Yuben Qu,[4] and Panlong Yang[5]

[1]*National University of Defense Technology, Hefei 230031, China*
[2]*Centre for Assessment and Demonstration Research, Academy of Military Science, Beijing 100091, China*
[3]*Academy of Military Science, Beijing 100091, China*
[4]*Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai 200240, China*
[5]*University of Science and Technology of China, Hefei 230031, China*

Correspondence should be addressed to Tao Wu; terence.taowu@gmail.com and Xiaochen Fan; fanxiaochen33@gmail.com

Received 7 August 2021; Revised 10 September 2021; Accepted 11 October 2021; Published 10 November 2021

Academic Editor: Pengfei Wang

With the rapid development of wireless communication technologies and the proliferation of the urban Internet of Things (IoT), the paradigm of mobile computing has been shifting from centralized clouds to edge networks. As an enabling paradigm for computation-intensive and latency-sensitive computation tasks, mobile edge computing (MEC) can provide in-proximity computing services for resource-constrained IoT devices. Nevertheless, it remains challenging to optimize computation offloading from IoT devices to heterogeneous edge servers, considering complex intertask dependency, limited bandwidth, and dynamic networks. In this paper, we address the above challenges in MEC with TPD, that is, temporal and positional computation offloading with dynamic-dependent tasks. In particular, we investigate channel interference and intertask dependency by considering the position and moment of computation offloading simultaneously. We define a novel criterion for assessing the criticality of each task, and we identify the critical path based on a directed acyclic graph of all tasks. Furthermore, we propose an online algorithm for finding the optimal computation offloading strategy with intertask dependency and adjusting the strategy in real-time when facing dynamic tasks. Extensive simulation results show that our algorithm reduces significantly the time to complete all tasks by 30–60% in different scenarios and takes less time to adjust the offloading strategy in dynamic MEC systems.

1. Introduction

With the rapid development of wireless communication technologies and the pervasive use of the urban Internet of Things (IoT), IoT devices have become deeply integrated into daily life [1]. At the same time, the rapid increase in the number of IoT devices has brought new challenges in processing the massive amount of IoT data [2]. With constrained computing resources and limited battery capacity, IoT devices are facing rapid energy drains as a result of processing computation-intensive tasks [3, 4]. While some initial efforts have been made to solve such problems by leveraging centralized cloud computing, the long communication distance and unstable network connection can cause unacceptable delays to end users [5, 6]. To tackle this issue, mobile edge computing (MEC) has emerged as a promising computing paradigm for providing highly responsive computing services with low latency [7–10]. By offloading computation tasks to edge servers, IoT devices incur fewer costs than they would if processing these tasks locally or remotely at cloud data centers [11].

Nevertheless, applications on IoT devices are highly diverse, and there can be intertask dependency in many computation tasks offloaded to MEC [12]. For instance, Figure 1(a) shows an access control system for MEC. When a user wants to enter, they must swipe their card and pass

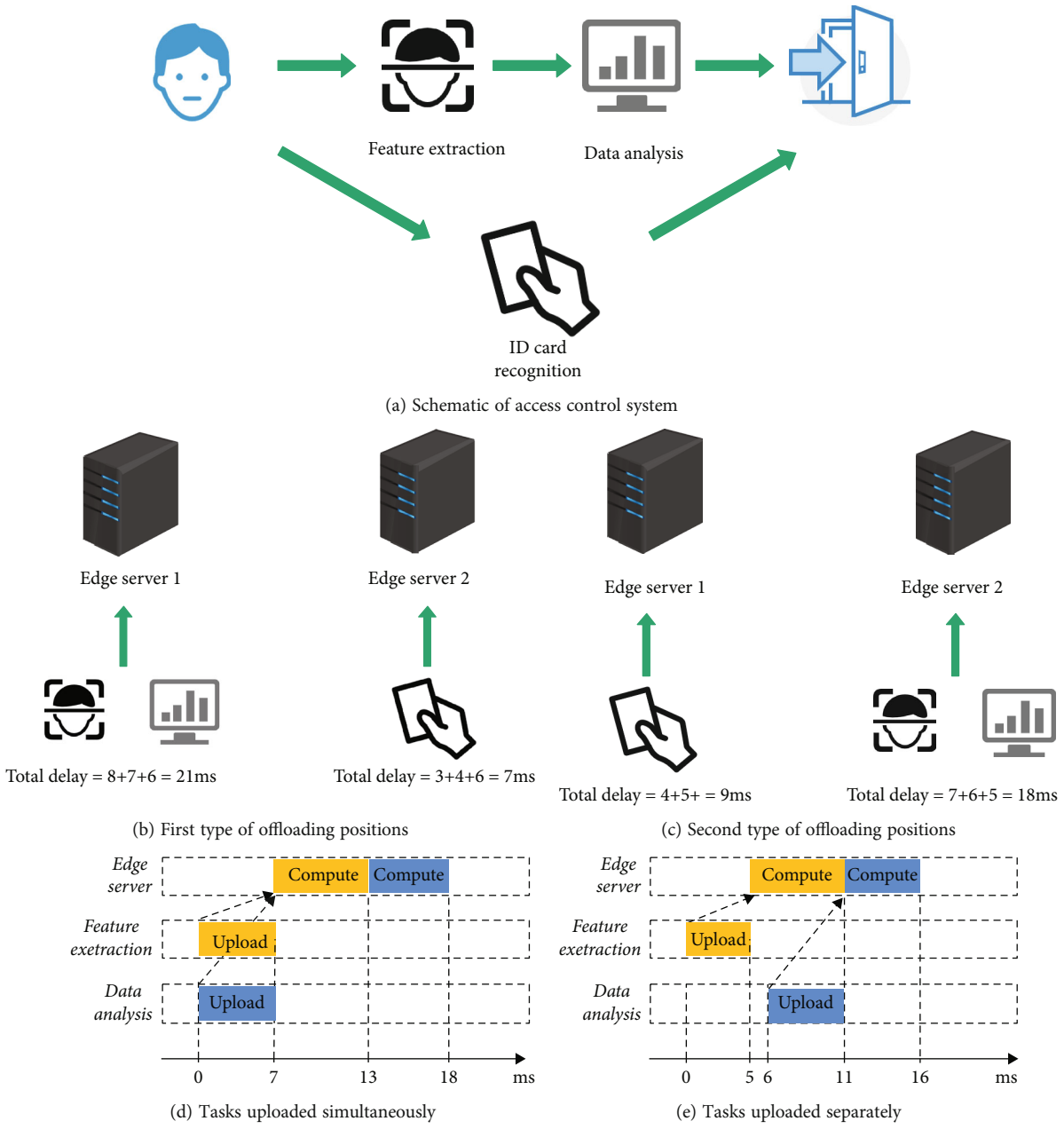


Figure 1: A simple example of an access control system.

facial recognition. The card reader generates the *ID card recognition* task, whereupon the camera takes user pictures and then generates the *feature extraction* task. When the latter is completed, the camera then generates the *data analysis* task to analyze the results of the *feature extraction* task. Intuitively, the door opens only after the *data analysis* and *ID card recognition* tasks have been processed successfully. The arrows in Figure 1(a) show the dependencies among the three computation tasks. Therefore, we must find an optimal offloading strategy for all three tasks. Suppose that there are two edge servers, i.e., edge server 1 and edge server 2, and that edge server 2 has greater computing capability and larger channel bandwidth than those of edge server 1. Correspondingly, the three tasks executed on the edge servers will generate different uploading delays and computation delays, as shown in Table 1. Note that there will be channel interference when tasks are uploaded simultaneously, which will cause longer delays than expected. Specifically, we carefully consider the following two issues of computation offloading with intertask dependency in MEC.

Offloading position: the first step is to find the optimal offloading positions for the three tasks. Here, we have produced two different types of offloading positions for tasks. With the first type (Figure 1(b)), the *ID card recognition* task is offloaded to edge server 2 and its total delay is $3 + 4 = 7$ ms. We assume that the *feature extraction* and *data analysis* tasks are uploaded simultaneously, so that the total delay is $8 + 7 + 6 = 21$ ms (*data analysis* must wait for *feature extraction* to complete). In this case, the door opens after 21 ms. As can be seen, both the *feature extraction* and *data analysis*

Table 1: Delays of three tasks on different servers.

	Feature extraction	Data analysis	ID card recognition
Uploading on S_1 (separated)	6 ms	6 ms	4 ms
Uploading on S_1 (synchronous)	8 ms	8 ms	5 ms
Computing on S_1	7 ms	6 ms	5 ms
Uploading on S_2 (separated)	5 ms	5 ms	3 ms
Uploading on S_2 (synchronous)	7 ms	7 ms	4 ms
Computing on S_2	6 ms	5 ms	4 ms

tasks determine the delay of door opening. However, if we offload both the *feature extraction* and *data analysis* tasks to edge server 2 (Figure 1(c)), then the total delay is reduced to 18 ms. Therefore, it is essential to consider task dependencies when seeking desirable offloading positions.

Offloading moment: if the *feature extraction* and *data analysis* tasks are offloaded simultaneously to edge server 2 at 0 ms (Figure 1(d)), then there is channel interference and the uploading delay of the two tasks becomes 7 ms. Consequently, the door opens after 18 ms. However, if we set the offloading moment of the *data analysis* task as 6 ms (Figure 1(e)), then channel interference is avoided and the overall uploading delay is reduced to 5 ms. Finally, the door opening time is advanced to 16 ms. By introducing this example, we show that the offloading moment also has considerable impact on task processing in mobile computation offloading.

Existing solutions for mobile computation offloading are focused mainly on reducing task response delays, and generally, they offload all computation tasks to MEC and then make optimized adjustments to the edge servers. However, as the above observation cases show, the potential intertask dependency and channel contention will cause significant processing delays in MEC systems with thousands or even millions of IoT devices. Herein, we study the problem of TPD, that is, temporal and positional computation offloading with dynamic-dependent tasks in MEC. In particular, we investigate computation offloading in MEC considering both intertask dependencies and offloading moments. We also address the dynamics of mobile edge networks, where new tasks are involved continuously in computation offloading. Formally, we aim to solve the following three critical challenges.

(i) How to find the optimal offloading positions of tasks under dependency restrictions: with more tasks, the dependencies among them become very complicated. Therefore, a challenge is how to find the optimal offloading position for each task under such complicated dependency restrictions

(ii) How to calculate the offloading moments of tasks to avoid channel interference: the execution and upload delays of tasks are affected by various factors such as channel quality and the computation capabilities of the edge servers. Therefore, it is difficult to estimate these delays, thereby posing the challenge of how to calculate the offloading moments of tasks to reduce channel interference

(iii) How to reduce the time overhead of updating the offloading strategy in real time: in a dynamic MEC system, IoT devices generate new tasks frequently, and the offloading strategy must also be updated frequently. However, doing so incurs a time overhead, so another challenge is how to update the offloading strategy in real time

To overcome the above challenges, we propose an online offloading algorithm known as the critical task first (CTF) algorithm. Combined with the idea of the critical path, the CTF algorithm marks the criticality of tasks based on their dependencies and decides the offloading positions of tasks according to their criticality. When IoT devices generate new tasks in the MEC system, the CTF algorithm can update the offloading strategy in real time with minimal time overhead. The main contributions in this paper are summarized as follows:

(i) We not only decide the offloading positions and execution order of tasks but also calculate the offloading moment of each task to reduce channel interference and uploading delay

(ii) Considering the dependencies among tasks and combining the critical-path idea, we propose an efficient online offloading algorithm, i.e., the CTF algorithm. This uses a backward criticality notation (BCN) method to mark the criticality of tasks, and it can update the offloading strategy with less time overhead in the dynamic MEC system

(iii) We conduct simulations to assess the effectiveness of the CTF algorithm in different scenarios. Compared with previous studies, the CTF algorithm is effective at reducing the time to complete all tasks and updating the offloading strategy with less time overhead

The rest of this paper is organized as follows. Section 2 introduces related work. Section 3 presents the system model and problem formulation. Section 4 introduces the main design of the offloading strategy. Section 5 presents the evaluation results. Section 6 concludes the paper.

2. Related Work

As one of the core technologies in MEC, computation offloading has attracted considerable research attention in recent years [13, 14]. According to the optimization goal, computation offloading can be divided into three categories:

(i) minimizing the delay [15–17], (ii) reducing the energy consumption [18, 19], or (iii) balancing the delay and energy consumption [20, 21]. Herein, we consider mainly computation offloading to minimize task delays.

Previous studies of minimizing task delays can be classified into two broad categories: centralized and distributed. The centralized method [22, 23] sets the MEC controller to collect global information with which it makes an optimal offloading decision for each task. Cooperation among multiple mobile devices is presented in [22], wherein the authors construct a potential game to realize task scheduling aimed at minimizing the overhead of each mobile device. In [23], Chen and Hao propose an efficient task-offloading policy in software-defined ultradense networks; they formulate the task-offloading problem as an NP-hard nonlinear mixed-integer programming problem and transform this optimization problem into a task-placement subproblem and a resource-allocation subproblem. Another direction is distributed resource allocation for multiuser MEC systems [24–26]. In [24], by targeting a multiuser and multiserver scenario involving a small-cell network integrated with MEC, the authors formulate the problem as a distributed overhead-minimization problem to minimize the overhead for users.

Dependency-aware computation offloading has also been studied widely. In [27], the authors propose a dependency-aware offloading scheme in MEC with edge–cloud cooperation under task-dependency constraints. In [28], the authors propose a model-free approach based on reinforcement learning, i.e., a Q-learning approach that adaptively learns to optimize the offloading decision and energy consumption jointly by interacting with the network environment. In [29], the authors study the impact of interuser task dependency on task offloading and resource allocation in a two-user MEC network, and they propose an efficient algorithm to optimize such decisions.

In [30, 31], the authors use game theory to solve the problem of optimal offloading. In [31], the authors also consider the dependency among subtasks and the contention among multiple users, and they propose the distributed earliest finish-time offloading (DEFO) algorithm based on noncooperative game theory to reduce the overall completion time of IoT applications. Compared with our CTF algorithm, the DEFO algorithm only decides the offloading positions and execution order of tasks without considering the offloading moments, thereby causing serious channel interference. Also, the DEFO algorithm is designed for static MEC systems, but actual MEC systems change dynamically. By contrast, the proposed CTF algorithm can perform temporal and positional computation offloading according to task dependencies and update the offloading strategy with less time overhead in dynamic MEC systems.

3. System Model

This section introduces the system model. We begin by describing the network model and then present the communication and computation models in detail. For clarity of presentation, Table 2 summarizes the notations used in the following formulation.

Table 2: Notations used herein.

Notation	Definition
$\mathcal{A}$	Set of IoT devices
$\mathcal{S}$	Set of edge servers
$T_{i,j}$	Task j of A_i
$m_{i,j}$	Computation input data size of $T_{i,j}$
$c_{i,j}$	No. of CPU cycles required for $T_{i,j}$
$a_{i,j}$	Offloading position of $T_{i,j}$
$P_{i,j}$	Predecessor task set of $T_{i,j}$
$S_{i,j}$	Successor task set of $T_{i,j}$
$ST_{i,j}$	Offloading moment of $T_{i,j}$
$FT_{i,j}$	Finishing moment of $T_{i,j}$
$r_{i,k}^t$	Data upload rate from A_i to S_k at time t
B_k	Bandwidth of channel connected to S_k
ω	Background noise
q_i	Transmission power of IoT device A_i
$g_{i,k}$	Channel gain
f_i^l	Computation capability of IoT device A_i
f_k^e	Computation capability of edge server S_k
$t_{i,j}$	Total delay of $T_{i,j}$
Φ	Offloading strategy for all tasks
Ψ	Moment at which all tasks are completed

3.1. Network Model. As shown in Figure 2, we consider a typical multiuser and multiserver MEC system with n IoT devices, l nearby edge servers, and wireless access points such as macro base stations and small-cell base stations with relatively powerful and heterogeneous computation abilities. The IoT devices can offload tasks to any edge server through the base stations and the core network. The sets of IoT devices and edge servers are denoted as $\mathcal{A} = \{A_1, A_2, \cdots, A_i, \cdots, A_n\}$ and $\mathcal{S} = \{S_1, S_2, \cdots, S_k, \cdots, S_l\}$, respectively. We suppose that each IoT device generates a set of computationally intensive tasks to be executed during the work. Let $\{T_{i,1}, T_{i,2}, \cdots, T_{i,j}, \cdots, T_{i,m}\}$ denote the set of tasks from IoT device A_i, where $T_{i,j}$ represents task j. For efficient computation offloading, we exploit the potential dependency correlation among the tasks [32]. As in the example in Figure 1(a), executing the *data analysis* task requires completion of the *feature extraction* task as a prerequisite. Therefore, we define the *feature extraction* task as the predecessor task of the *data analysis* task, and the *data analysis* task is its successor task.

Task $T_{i,j}$ can then be described by the five-tuple $(m_{i,j}, c_{i,j}, a_{i,j}, P_{i,j}, S_{i,j})$, where the terms are as follows: (i) $m_{i,j}$ is the size of the input data for task execution (e.g., the program codes and input parameters); (ii) $c_{i,j}$ is the total number of CPU cycles required to accomplish task $T_{i,j}$; (iii) because each task can be either executed locally or offloaded to a nearby edge server, we use $a_{i,j} \in \{0 \cup S\}$ to denote the

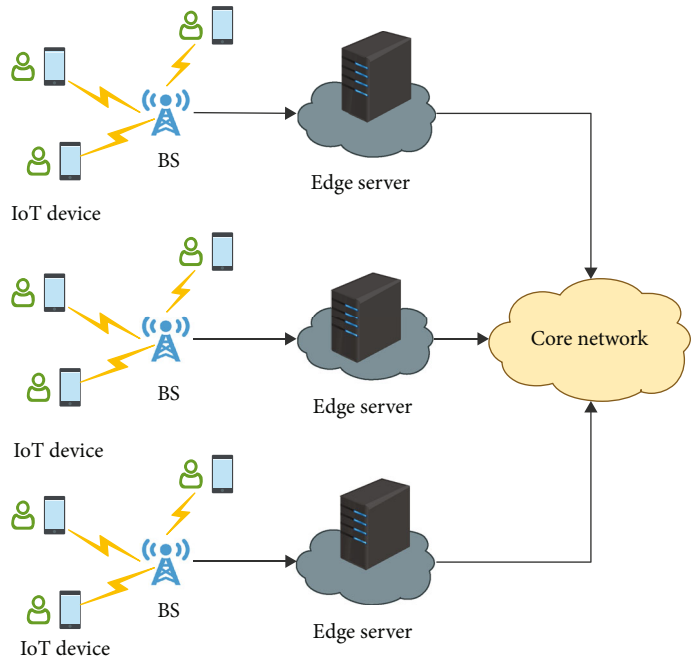


Figure 2: Typical multiuser and multiserver mobile edge computing (MEC) system.

offloading position of task $T_{i,j}$, where $a_{i,j} = 0$ means that task $T_{i,j}$ is executed locally on IoT device A_i and $a_{i,j} = k$ means that $T_{i,j}$ is offloaded to edge server S_k for execution; (iv) $P_{i,j}$ is the predecessor task set of $T_{i,j}$; and (v) $S_{i,j}$ is the successor task set. A task can start only after all its predecessors have been completed. $ST_{i,j}$ is the offloading moment of $T_{i,j}$, and $FT_{i,j}$ is the finishing moment.

3.2. Communication and Computation Models. In this subsection, we introduce how to model the task delay, which can be divided into communication and computation parts. If tasks are executed by the IoT devices themselves, then only a computing delay is incurred; otherwise, the IoT devices select an edge server from the edge server set S for computation offloading, which incurs uploading, waiting, and computing delays. Next, we introduce the communication model followed by the computation model.

Communication model: the communication channel quality changes constantly during the process of computation offloading. In [30, 31], the authors give a formula for calculating the task upload rate. Based on this, we introduce how to calculate the aforementioned communication delays in detail according to the time. $p_{i,k}^t$ indicates whether device A_i is uploading tasks to edge server S_k at time t. If there is a task $T_{i,j}$ such that $a_{i,j} = k$ and its uploading interval contains time t, then $p_{i,k}^t = 1$, otherwise $p_{i,k}^t = 0$. $r_{i,k}^t$ denotes the data upload rate from device A_i to edge server S_k at time t and is given as

$$r_{i,k}^t = B_k \log_2\left(1 + \frac{q_i g_{i,k}}{\varpi + \sum_{i' \in [1,..,n], i' \neq i} q_{i'} g_{i',k} p_{i',k}^t}\right), \quad (1)$$

where B_k is the bandwidth of the communication channel connected to edge server S_k, q_i is the transmission power of A_i, $g_{i,j}$ is the channel gain for the link between A_i and S_k, ϖ is the background noise, and $\sum_{i' \in [1,..,n], i' \neq i} q_{i'} g_{i',k} p_{i',k}^t$ is the wireless interference suffered from other IoT devices when A_i uploads tasks to S_k.

Computation model: a computation task can be either handled locally or offloaded to an edge server for computing, which leads to two different delay computing methods.

(1) *Local computing*: the task is handled on the IoT device itself, and there is only a computing delay in the process. f_i^l is the computing capability of IoT device A_i, which represents the number of CPU cycles that A_i can calculate per second. Given the number of CPU cycles for task $T_{i,j}$, the delay $t_{i,j}^l$ of $T_{i,j}$ with local computing can be represented as

$$t_{i,j}^l = \frac{c_{i,j}}{f_i^l} \quad (2)$$

(2) *Edge computing*: tasks are offloaded to edge servers for computing, and the process involves the following steps: the task is uploaded to the edge server; then, it is queued in the server for execution, and finally, the edge server computes the task to obtain the results. Therefore, these three steps produce a three-part delay (upload delay, queuing delay, and computing delay). The upload delay $t_{i,j}^{k,U}$ of $T_{i,j}$ from

A_i to S_k is determined by the data upload rate $r_{i,k}^t$ and the data volume $m_{i,j}$. Therefore, $t_{i,j}^{k,U}$ is calculated as

$$t_{i,j}^{k,U} = \frac{m_{i,j}}{r_{i,k}^t} \tag{3}$$

After arriving at the edge server, $T_{i,j}$ may need to queue in the server for computing, and $t_{i,j}^{k,Q}$ denotes the queuing delay of $T_{i,j}$ on S_k. We use $Q_{i,j}$ to denote the task set that is queued in front of $T_{i,j}$ on S_k. Therefore, $t_{i,j}^{k,Q}$ is calculated as

$$t_{i,j}^{k,Q} = \sum_{T_{i',j'} \in Q_{i,j}:} \boxtimes \frac{c_{i',j'}}{f_k^e}. \tag{4}$$

f_k^e is the computing capability of edge server S_k, and we calculate the computing delay $t_{i,j}^{k,C}$ as

$$t_{i,j}^{k,C} = \frac{c_{i,j}}{f_k^e}. \tag{5}$$

In *edge computing*, the delay $t_{i,j}^k$ of $T_{i,j}$ comprises the uploading, queuing, and computing delays as

$$t_{i,j}^k = t_{i,j}^{k,U} + t_{i,j}^{k,Q} + t_{i,j}^{k,C}. \tag{6}$$

$t_{i,j}$ is the total delay of $T_{i,j}$ and is denoted as

$$t_{i,j} = \begin{cases} t_{i,j}^l & \text{if } a_{i,j} = 0, \\ t_{i,j}^k & \text{if } a_{i,j} = k. \end{cases} \tag{7}$$

If $T_{i,j}$ chooses *local computing*, then $t_{i,j} = t_{i,j}^l$. If $T_{i,j}$ chooses edge server S_k, then $t_{i,j} = t_{i,j}^k$. Therefore, given $ST_{i,j}$ and $a_{i,j}$ of the task, we calculate the finishing moment as

$$FT_{i,j} = ST_{i,j} + t_{i,j}. \tag{8}$$

3.3. Problem Formulation. Herein, given the dependencies among tasks, the dynamics of the MEC system, and the multiuser wireless interference, we jointly optimize the task offloading position and moment.

(1) Optimizing offloading positions: tasks are either executed locally at the IoT devices or offloaded to different edge servers, thereby causing different delays. Optimizing the offloading positions involves finding the optimal offloading position for each task that minimizes the time to complete all tasks

(2) Optimizing offloading moments: offloading tasks at different moments produces channel interference among IoT devices. Optimizing the offloading moments involves determining the optimal offloading moments to avoid channel interference and reduce the time of uploading tasks

Herein, our goal is to find both (i) the offloading position $a_{i,j}$ at which task $T_{i,j}$ is executed and (ii) the offloading moment $ST_{i,j}$ at which $T_{i,j}$ starts to be offloaded. We use Φ $[(a_{1,1}, ST_{1,1}), \cdots, (a_{i,j}, ST_{i,j}), \cdots, (a_{n,m}, ST_{n,m})]$ to represent the offloading strategy. Our goal is then to find the optimal offloading strategy that minimizes the total time to complete all tasks. Let Ψ denote the moment at which all tasks are completed, which is equivalent to the finishing moment of the final completed task, i.e.,

$$\Psi = \text{Max}\left(ST_{i,j} + t_{i,j}\right), \quad i \in [1,.,n], j \in [1,.,m]. \tag{9}$$

Therefore, we formulate the problem of TPD (temporal and positional computation offloading with dynamic and dependent tasks) mathematically as follows:

$$\text{Opt.} \quad \min\Psi \tag{10}$$

$$\begin{aligned} &\text{s.t.} \quad \left[ST_{i,j}, FT_{i,j}\right] \cap \left[ST_{i',j'}, FT_{i',j'}\right] = \varnothing, \\ &\forall a_{i,j} = a_{i',j'}, i, i' \in [1, n], \quad j, j' \in [1, m], \end{aligned} \tag{11}$$

$$ST_{i,j} \geq \max\left\{ST_{i',j'} + t_{i',j'}\right\}, \quad T_{i',j'} \in P_{i,j}. \tag{12}$$

We seek the optimal offloading strategy Φ with the minimized Ψ (Equation (10)). However, there are two restrictions: (i) an edge server can only calculate one task at a time; therefore, the time intervals of two tasks that select the same server cannot overlap (Equation (11)); (ii) the limitation of dependencies means that a successor task can only start after all its predecessor tasks are completed, and its offloading moment must be later than the finishing moments of all its predecessors (Equation (12)).

3.4. Complexity Analysis of the Problem. Here, we analyze the complexity of the above optimization problem by comparing it with the flexible job-shop scheduling problem (FJSP), which is the problem of scheduling workpiece processing on machines. In the FJSP, a group of workpieces must be processed on a group of machines. Each workpiece involves multiple dependency-aware jobs, a machine can only process one job at a time, and jobs have different execution times on different machines. As a classic NP-hard problem [33, 34], the goal of the FJSP is to determine the execution machines and order of the jobs to reduce the time to complete all jobs.

After abstraction, our optimization problem is similar to the FJSP. The applications running on the IoT devices can be regarded as workpieces, the tasks can be regarded as jobs, and the edge servers can be regarded as machines in the FJSP. Also, the constraints and optimization objectives of the two problems are the same. Therefore, our optimization problem of computation offloading can be regarded as a type of FJSP. However, our optimization problem is more complicated than the classic FJSP because we not only determine

the offloading positions and order of the tasks but must also calculate the precise offloading moments, and we consider the communication time and channel interference. Therefore, the optimal offloading problem proposed by us is also an NP-hard problem.

4. Task Offloading Strategy

In this section, we introduce the detailed task-offloading strategy for our problem. We begin by constructing a directed acyclic graph (DAG) according to the dependency correlation of tasks and proposing a BCN method to mark the criticality of all tasks. We then devise the online CTF offloading algorithm to minimize the time to complete all tasks.

4.1. Marking of Task Criticality. The dependency correlation among tasks means that we can first construct a DAG, as the example shown in Figure 3, to present their topological structure. The nodes in the DAG represent tasks generated by IoT devices, and the directed edges denote the specific dependency relations between tasks. A node is weighted by the number of CPU cycles required by the corresponding task, and the length of a path in the DAG is the sum of the weights of the nodes on the path. The critical path is the longest path in the DAG, and the completion times of tasks on the critical path have a direct effect on the time Ψ.

Intuitively, we can find the critical path and offload tasks on it to edge servers with sufficient computing capability to reduce Ψ. However, searching for the longest path in the DAG using traditional methods such as the Floyd or Dijkstra algorithm would generally incur a large computation time overhead. Also, the critical path will change frequently with the emergence of new tasks. Therefore, we can transform the problem of finding the critical path into marking the criticality of each task. A task with a greater impact on Ψ has higher criticality, and we use $\gamma_{i,j}$ to denote the criticality of task $T_{i,j}$. Next, we introduce the method for calculating the criticality of tasks.

Tasks without successors are called *exit tasks*, such as T_3, T_4, and T_6 in Figure 3, and the path from a task to an *exit tasks* is called an influence path. As shown in Figure 3, T_2 has two influence paths such as $(T_2 \longrightarrow T_4)$ and $(T_2 \longrightarrow T_5 \longrightarrow T_6)$. The length of an influence path of T_2 is the sum of CPU cycles required by tasks on that path and indicates the amount of calculation affected by T_2. We reason that a task that affects a greater amount of calculation is more critical, so we define the criticality of a task as the length of its longest influence path.

On this basis, we propose a BCN method for calculating the length of a task's longest influence path and marking the task's criticality with less time overhead. The criticality of an *exit task* is the number of CPU cycles that it requires. The criticality of a nonexit task depends on its successor tasks and is calculated as

$$\gamma_{i,j} = c_{i,j} + \max \gamma_{i',j'}, \quad T_{i',j'} \in P_{i,j}. \tag{13}$$

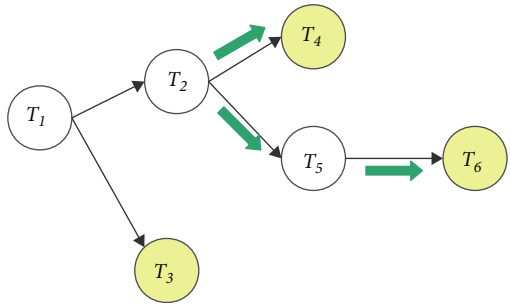


Figure 3: Example of topological diagram of tasks.

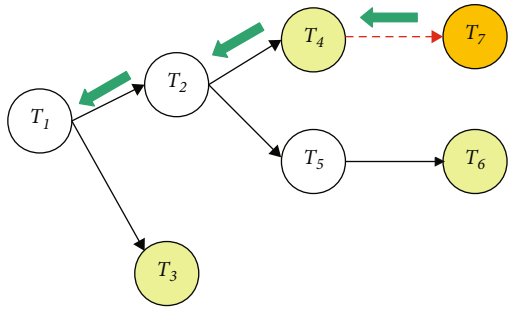


Figure 4: A new task is added to the topological diagram of tasks.

In this way, we can mark the criticality of all tasks. When IoT devices generate new tasks (e.g., T_7), thereby changing the task topology, the BCN method can update the criticality of affected tasks with a small overhead. As an example in Figure 4, T_7 is generated and added to the task topology, thereby affecting its predecessor tasks. Therefore, the BCN method recalculates the criticality of T_4 by means of Equation (13). In this way, the criticality of the other affected tasks (T_1, T_2), as indicated by the arrows in Figure 4, is updated.

4.2. CTF Algorithm. Based on the criticality of tasks, we propose the CTF algorithm. In a dynamic MEC system, the CTF algorithm will monitor the tasks' criticality in real time and find the optimal offloading position and moment for each task to minimize Ψ. The pseudocode of the CTF algorithm is shown in Algorithm 1 and has four stages.

(1) State division

First, the CTF algorithm collects information about all the tasks and divides them (*line 1*) according to the following three states.

(1) *Waiting state*: a task whose predecessor tasks have not been completed

(2) *Ready state*: a task whose predecessor tasks have been completed but have not been offloaded

(3) *Computing state*: a task that has been offloaded and is now computing

Correspondingly, the CTF algorithm generates three types of task queues: the waiting queue, the ready queue,

Input: *Information about tasks, edge servers, and IoT devices*
Output: Offloading strategy with minimized Ψ
1 Put tasks into three types of queues according to task status. Use the BCN method to mark the criticality of tasks. Sort ready tasks from high to low according to criticality. all tasks $T_{i,j}$ in ready queue **do**
2 **for** all S_k in S**do**
3 Calculate $ST_{i,j}^k$ and $FT_{i,j}^k$ on S_k
4 **end**
5 Choose the edge server with the smallest $FT_{i,j}^k$ Calculate $ST_{i,j}^l$ and $FT_{i,j}^l$ on local IoT device $FT_{i,j}^l < FT_{i,j}^k$**then**
6 $FT_{i,j} = FT_{i,j}^l$, $ST_{i,j} = ST_{i,j}^l$, $a_{i,j} = 0$
7 **end**
8 **else**
9 $FT_{i,j} = FT_{i,j}^k$, $ST_{i,j} = ST_{i,j}^k$, $a_{i,j} = k$
10 **end**
11 **end**
12 Go back to step 2 when a new task or new ready task is generated.

Algorithm 1: Critical task first (CTF) algorithm.

and the computing queue. As shown in Figure 5, tasks generated by IoT devices are first put into the waiting queue. After all its predecessors have been completed, a task is moved from the waiting queue to the ready queue. When its offloading moment comes, a ready task is offloaded for computing and then put into the computing queue. Finally, the computation results of tasks are obtained. After the tasks have been divided into the three states, the CTF algorithm uses the BCN method to mark their criticality (*line 2*).

(2) Sorting of ready tasks

Because waiting tasks are not yet ready for computing and computing tasks have been offloaded, we only need to determine when and where the tasks in the ready queue should be offloaded. First, the CTF algorithm sorts the tasks according to their criticality (*line 3*). Based on the principle of critical tasks first, the CTF algorithm decides the offloading positions and moments for critical tasks first and then for other tasks. If two tasks choose the same edge server, the one with the higher criticality is offloaded first, and the other tasks must wait.

(3) Deciding on task offloading

In the process of computation offloading, the CTF algorithm calculates the offloading and finishing moments of the task on each edge server or locally and chooses the position with minimized finishing moment as the final result. Next, using $T_{i,j}$ as an example, we introduce how the CTF algorithm calculates the offloading and finishing moments on edge servers (taking S_k as an example) for ready tasks at time t^{now} (*lines 5–8*). The detailed calculation process can be divided into the following three cases.

Case 1. S_k is idle, and no more-critical tasks have chosen it. In this case, task $T_{i,j}$ can be offloaded directly; the offloading moment on S_k is $ST_{i,j}^k = t^{\text{now}}$, and $T_{i,j}$ is calculated after it arrives on the edge server, as in Figure 6. Therefore, the finishing moment of $T_{i,j}$ on S_k is $FT_{i,j}^k = t^{\text{now}} + t_{i,j}^{k,U} + t_{i,j}^{k,C}$.

Case 2. A task is being executed on S_k but no more-critical tasks have chosen it. As shown in Figure 7, we suppose that $T_{a,b}$ is the task being executed on S_k. We calculate the offloading moment of $T_{i,j}$ as $ST_{i,j}^k = FT_{a,b}^k - t_{i,j}^{k,U}$. Therefore, $T_{i,j}$ can be calculated as soon as it arrives at S_k, and its finishing time is $FT_{i,j}^k = FT_{a,b}^k + t_{i,j}^{k,C}$.

Case 3. A task is being executed on S_k but some more-critical tasks have also chosen it. Although task $T_{a,b}$ is being executed, multiple more-critical tasks have selected S_k, so $T_{i,j}$ must wait for those tasks to complete before it can be executed on the edge server (Figure 8). We use $\varphi_{i,j}$ to represent this more-critical task set. The finishing moment of task $T_{i,j}$ is calculated as

$$FT_{i,j}^k = FT_{a,b}^k + \sum_{T_{i',j'} \in \varphi_{i,j}} t_{i',j'}^{k,C} + t_{i,j}^{k,C}, \tag{14}$$

and the offloading moment of $T_{i,j}$ is $ST_{i,j}^k = FT_{i,j}^k - t_{i,j}^{k,C} - t_{i,j}^{k,U}$.

As shown above, the CTF algorithm controls the offloading moments of tasks so that they (i) do not need to wait on a server, thereby avoiding waiting delay and (ii) are uploaded separately, thereby avoiding channel interference from other IoT devices. Therefore, the upload rate of task $T_{i,j}$ becomes

$$r_{i,k}^t = B_k \log_2\left(1 + \frac{q_i g_{i,k}}{\omega}\right). \tag{15}$$

Finishing moment on local IoT device: On the local IoT device, task $T_{i,j}$ can be calculated directly (*line 9*), and the offloading moment is then $ST_{i,j}^l = t^{\text{now}}$ and $FT_{i,j}^l = ST_{i,j}^l + t_{i,j}^l$.

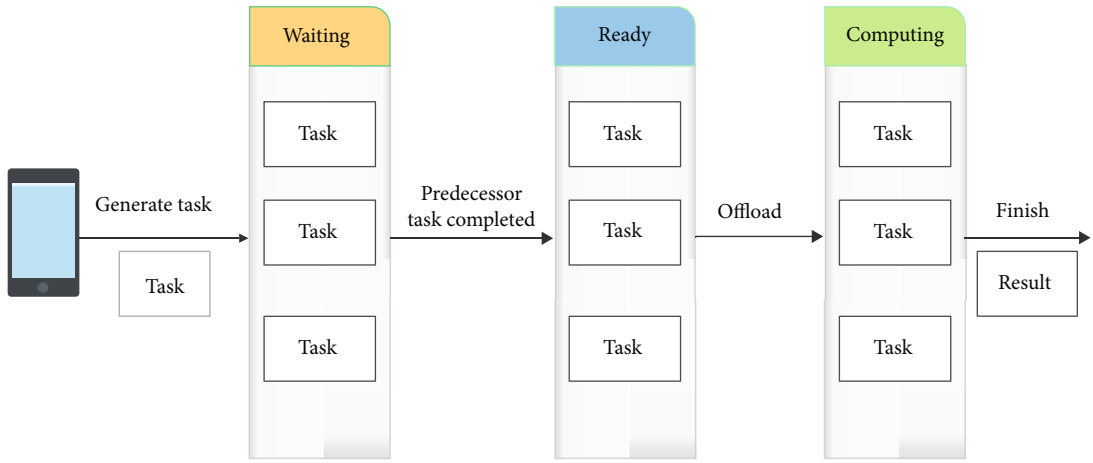


Figure 5: Transformation of the three task states.

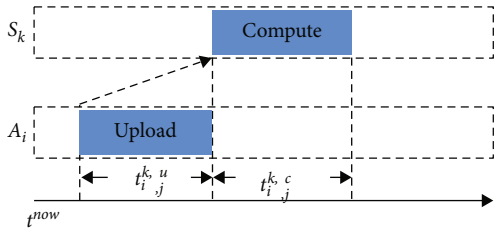


Figure 6: First case of the calculation.

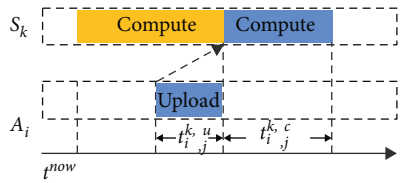


Figure 7: Second case of the calculation.

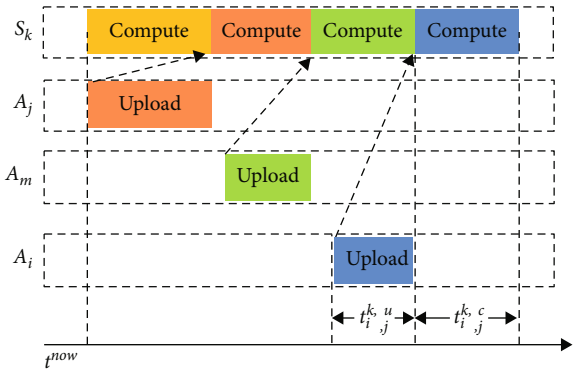


Figure 8: Third case of the calculation.

After calculating the finishing moments of $T_{i,j}$ on all edge servers and locally, the CTF algorithm chooses the offloading position with minimized $FT_{i,j}$, and the offloading moment is calculated according to the offloading position (*lines 10–15*). Therefore, we can obtain the offloading strategy for ready tasks at time t^{now}.

Table 3: Experimental parameters.

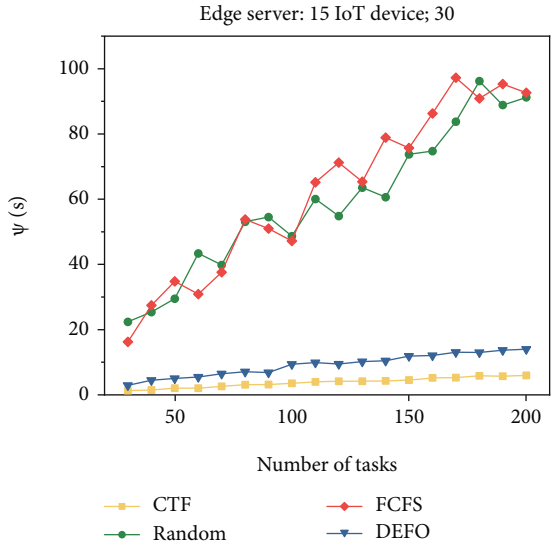
Parameter	Value
Base-station bandwidth	20 MHz ± 20%
Computing capability of edge servers	10 GHz ± 20%
Computing capability of IoT devices	1 GHz ± 20%
Transmission power of IoT devices	100 mW ± 20%
Distance from IoT device to each server	20–50 m
Pass loss factor	4
Task data volume	1000 kB ± 50%
No. of CPU cycles required for task	1000–5000 megacycles
Background noise	−100 dBm

(4) Updating of offloading strategy

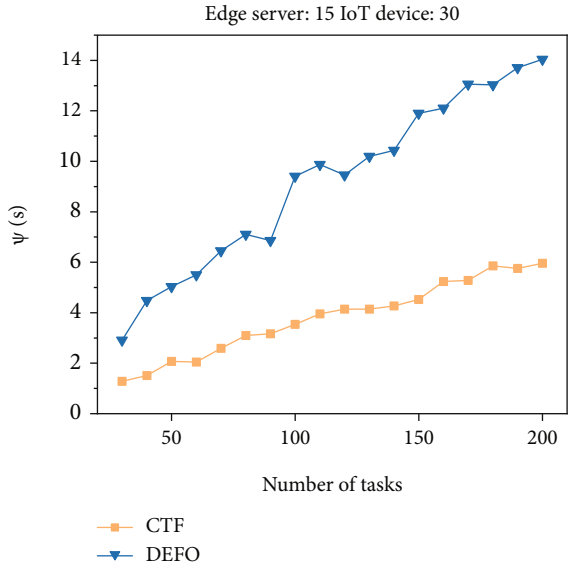
In a dynamic MEC system, the status of tasks changes over time, thereby decreasing the effectiveness of the offloading strategy. In the following two situations, the offloading strategy for ready tasks at time t^{now} may become suboptimal and require recalculation: (i) IoT devices generate new waiting tasks, thereby changing the task topology and thus the task criticality; (ii) new ready tasks are added to the ready queue but the offloading strategy at time t^{now} does not account for them.

As an online algorithm, the CTF algorithm updates the offloading strategy for ready tasks (*line 17*) in time to ensure the best effect. After a new waiting task is generated, the CTF algorithm first uses the BCN method to update the criticality of the affected tasks and then recalculates the best offloading position and moment for each ready task. If only new ready tasks are generated, then the CTF algorithm only needs to recalculate the offloading strategy for ready tasks.

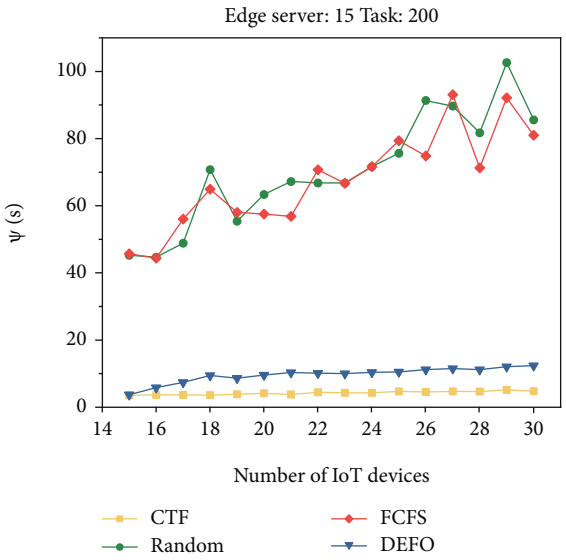
4.3. Complexity Analysis of the CTF Algorithm. Here, we take calculating the finishing moments of tasks as the basic calculation for analyzing the complexity of the CTF algorithm. Suppose that the MEC system has k edge servers and that the IoT devices generate n tasks in total, with Q denoting the number of ready tasks. When deciding the offloading



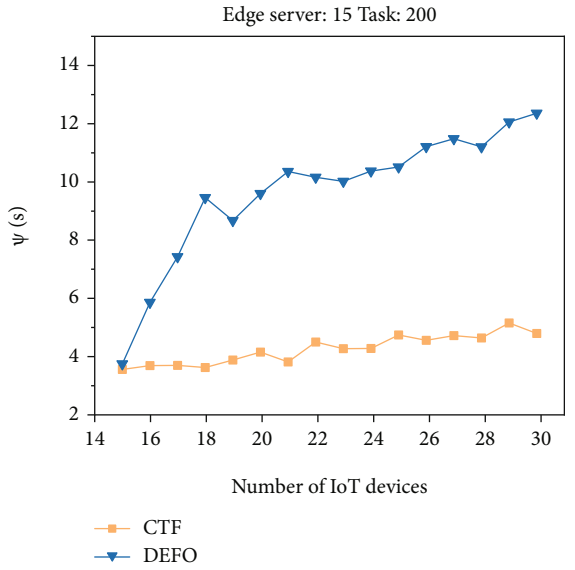
(a) Ψ for different numbers of tasks (4 algorithms)



(b) Ψ for different numbers of tasks (2 algorithms)

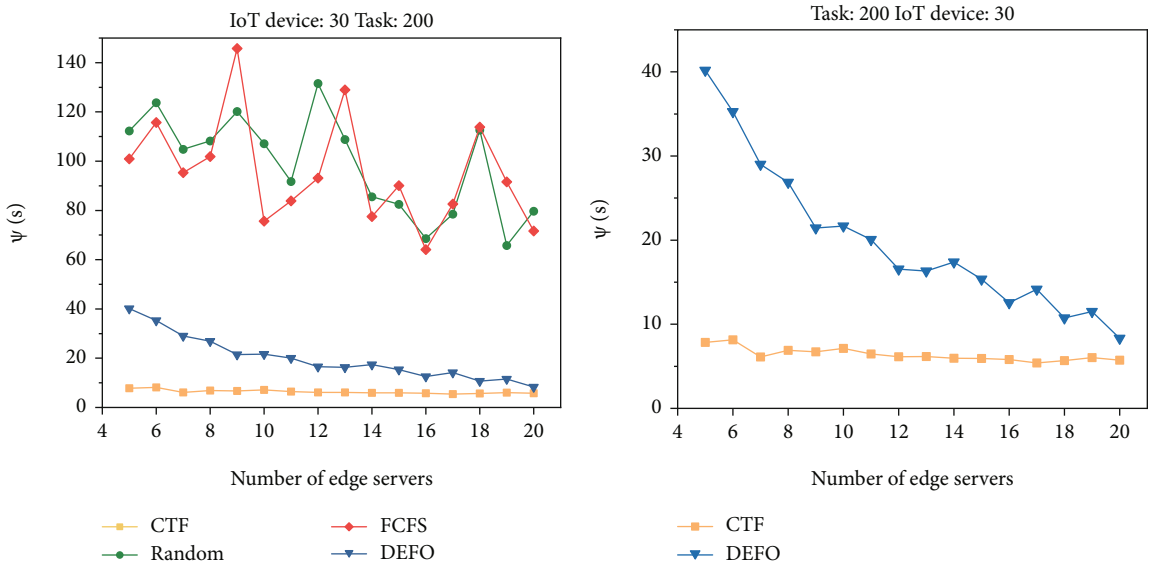


(c) Ψ for different numbers of IoT devices (4 algorithms)



(d) Ψ for different numbers of IoT devices (2 algorithms)

Figure 9: Continued.



(e) Ψ for different numbers of edge servers (4 algorithms)

(f) Ψ for different numbers of edge servers (2 algorithms)

Figure 9: Effects of different algorithms with a static MEC system.

positions of ready tasks, the CTF algorithm calculates the finishing moment of the ready task on each edge server and locally, so it does $k+1$ basic calculations. Therefore, finding the offloading strategy for all ready tasks requires $Q \times (k+1)$ basic calculations.

In the worst case, tasks are generated one at a time and no ready tasks are executed. Therefore, every time a new task is generated by an IoT device and put into the ready queue, the CTF algorithm updates the offloading strategy of all ready tasks. The number of tasks in the ready queue increases gradually from 1 to n, thereby requiring $(1+2+,\cdots,+n)(k+1) = ((n^2+n)/2)(k+1)$ basic calculations in total.

As analyzed above, in a dynamic MEC system containing k edge servers, the CTF algorithm offloads n tasks and requires $((n^2+n)/2)(k+1)$ basic calculations at most. In actual situations, tasks are rarely generated one at a time, and the number of tasks in the ready queue decreases with offloading. Therefore, the actual number of calculations is much less than $((n^2+n)/2)(k+1)$. Every time an IoT device generates a new task, the CTF algorithm requires only $Q \times (k+1)$ basic calculations. However, in some other methods (e.g., noncooperative game methods, genetic algorithms), once a new task is generated, the offloading strategy for all tasks must be recalculated, and the process of iterative convergence is very complicated, thereby generating a large time overhead.

5. Experiments and Evaluation

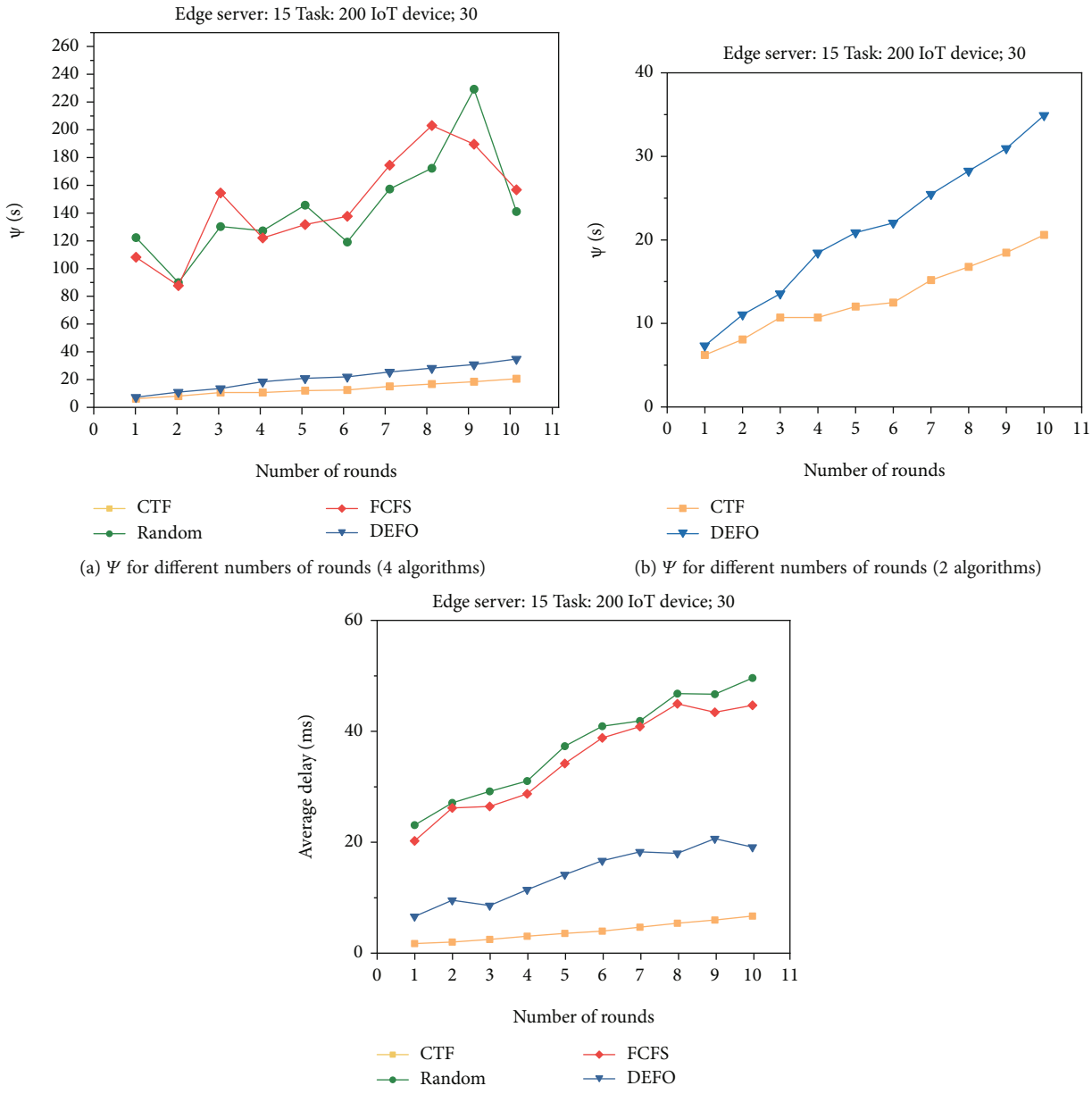
In this section, we report simulation experiments conducted to assess how the CTF algorithm performs with static and dynamic MEC systems. We also compare the CTF algorithm with the DEFO algorithm [31] and two other classic algorithms, i.e., the random algorithm and the first-come first-served (FCFS) algorithm. The random algorithm chooses offloading positions and moments randomly for tasks, whereas in the FCFS algorithm, tasks that are generated first are sent to the nearest edge server and executed first. The design of the simulation experiments and analysis of the results are shown below.

5.1. Experimental Settings. The simulation experiments involved a multiuser and multiserver MEC system with the parameter settings given in Table 3. The channel bandwidth of each base station was 20 MHz ± 20%, and the computing capability of each edge server was 10 GHz ± 20%. The computing capability of the IoT devices was 1 GHz ± 20%, and the transmission power of each IoT device was 100 mW ± 20%. Considering the mobility of IoT devices, the distance l between an IoT device and an edge servers was 20–50 m and changed continuously. The channel gain $g_{i,k}$ from A_i to S_k is $g_{i,k} = l^{\alpha}$, where α is the pass loss factor, and we set $\alpha = 4$. For tasks generated by IoT devices, the task data volume was 1000 kB ± 50%, and the number of CPU cycles required by a task was 1000–5000 megacycles. The background noise was −100 dBm, and the dependencies among tasks were generated randomly.

5.2. Effects of Different Algorithms with Static MEC System. Here, we begin by assessing the performances of the four algorithms with a static MEC system without generating new tasks. To explore how different factors (numbers of tasks, IoT devices, and edge servers) influence the algorithms, we conducted simulation experiments in the following three scenarios.

(1) MEC system with different numbers of tasks

In this experiment, we studied how different task numbers affect Ψ. To prevent the influence of other factors, we set 15 edge servers and 20 IoT devices. The number of tasks in the MEC system was 30 initially and then was increased



(a) Ψ for different numbers of rounds (4 algorithms)

(b) Ψ for different numbers of rounds (2 algorithms)

(c) Average task delay for different numbers of rounds (4 algorithms)

Figure 10: Effects of different algorithms with a dynamic MEC system.

gradually to 200. The Ψ of the offloading strategies generated by the four algorithms varied with the task number, and the results are shown in Figures 9(a) and 9(b).

Figure 9(a) shows that as the number of tasks increases, the computation resources become insufficient, and the Ψ values of the four algorithms increase constantly. In particular, the Ψ growth rates of the FCFS and random algorithms are very high, and their Ψ values are much higher than those of the CTF and DEFO algorithms. For clarity, we plot the results of the CTF and DEFO algorithms separately in Figure 9(b). The CTF algorithm performs better when facing a different number of tasks, and its Ψ values are only approximately half of those for the DEFO algorithm, which shows that the CTF algorithm adapts well to the MEC system with more tasks.

(2) MEC system with different numbers of IoT devices

IoT devices are the sources of tasks in the MEC system, so it is necessary to study how their number influences Ψ. In this experiment, we set 15 edge servers. Because task number is related to the number of IoT devices, we set the former as six times the latter. Starting with a MEC system with only 15 IoT devices, we gradually increased that number to 30, and the results of the experiment are shown in Figures 9(c) and 9(d).

As their number increases, IoT devices compete more fiercely for computing resources, thereby increasing the task execution delay. The Ψ values of the four algorithms continue to increase as the number of IoT devices is increased. However, the experimental results show that the Ψ values

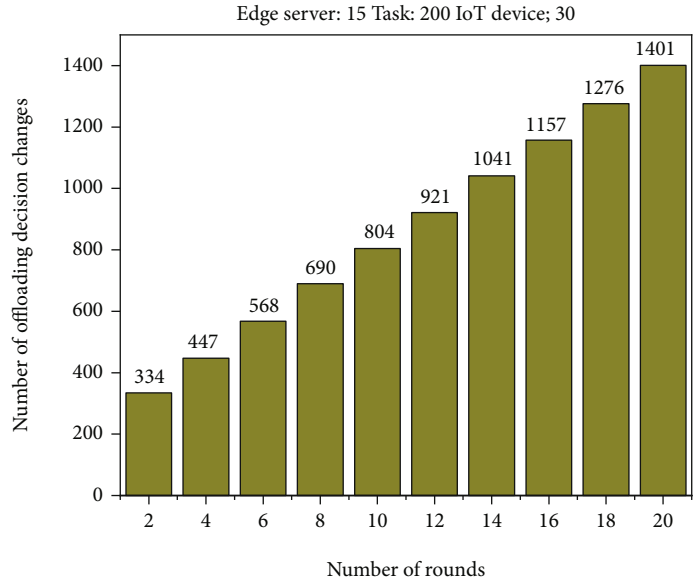


Figure 11: Numbers of decision changes in different rounds.

Table 4: Time overhead and Ψ of DEFO and CTF algorithms.

Round	1	2	3	4	5	6	7	8	9	10
Time overhead of CTF (s)	0.81	1.30	1.16	1.19	1.23	1.30	1.43	1.44	1.27	1.38
Time overhead of DEFO (s)	23.18	30.85	36.42	42.07	60.22	52.93	70.68	105.41	106.22	132.66
Ψ of CTF (s)	6.22	8.07	10.71	10.79	12.01	12.50	15.18	16.76	18.47	20.60
Ψ of DEFO (s)	7.32	11.05	13.56	18.45	20.86 s	22.03	25.45	28.24	30.93	34.91

for the CTF algorithm are 30% of those for the DEFO algorithm when the number of IoT devices exceeds 18 and remain low for the MEC system with different numbers of IoT devices.

(3) MEC system with different numbers of edge servers

As an important computing resource in a MEC system, the number of edge servers has a significant effect on the time to complete all tasks. Therefore, we explore how the number of edge servers influences Ψ. In this experiment, there were 30 IoT devices and 200 tasks. The number of edge servers was 5 initially and was increased gradually to 20, and the results of the experiment are shown in Figures 9(e) and 9(f).

The results show that Ψ can be reduced significantly by having more edge servers. In particular, the Ψ values for the random, FCFS, and DEFO algorithms decrease significantly. By contrast, the CTF algorithm performs better and is more stable in a MEC system with different numbers of servers.

From the three simulation experiments above, we conclude that the CTF algorithm can reduce Ψ significantly in different static MEC systems.

5.3. Effects of Different Algorithms with Dynamic MEC System. Here, we report simulation experiments conducted to assess the performances of the four algorithms with a dynamic MEC system. To build such a system, we added new tasks continuously to simulate the process of IoT devices generating tasks while the algorithms are running. First, we constructed an initial MEC system with 15 edge servers, 30 IoT devices, and 200 tasks. During the execution of the algorithms, we added 20 new tasks to the MEC system in each round. As the number of rounds increased, the MEC system became more dynamic, and the results of the experiment with different numbers of rounds are shown in Figure 10.

The enhanced dynamics of the MEC system decrease the effectiveness of the offloading strategy. Figure 10(a) shows that as the number of rounds increases, the Ψ values of each algorithm increase significantly, but those for the CTF algorithm are lower, as is their growth rate. In particular, the Ψ values for the CTF algorithm are only 60% of those for the DEFO algorithm (Figure 10(b)). In addition to Ψ, the average task delay is also affected. Figure 10(c) shows that as the number of rounds increases, so does the average task delay. However, the average task delay with the CTF algorithm is significantly lower than that with the other three algorithms, which means that the CTF algorithm adapts better to the dynamic MEC system.

In a dynamic MEC system, the CTF algorithm adjusts the offloading positions and moments of tasks continuously to maintain the most effective offloading strategy. Figure 11

shows the numbers of changes in offloading positions and moments of tasks made by the CTF algorithm in different rounds. As the number of rounds increases, the MEC system becomes more dynamic, and the CTF algorithm adjusts the offloading strategy frequently to ensure the best results.

An online algorithm incurs a short time overhead to update the offloading strategy in a dynamic MEC system. Here, we study the time overhead of the DEFO and CTF algorithms with the dynamic MEC system. The DEFO and CTF algorithms were both programmed in Python on a Windows 10 platform and tested in an Intel 2.60 GHz, 16 GB memory environment. We measured the time overhead of running the two algorithms in different rounds, as given in Table 4. Clearly, the time overhead of the CTF algorithm under different rounds is short and stable. With 10 rounds, the time overhead of the DEFO algorithm is 100 times that of the CTF algorithm. Compared with Ψ, the time overhead incurred by the CTF algorithm is acceptable, whereas that for the DEFO algorithm is too long to deal with the dynamic MEC system. Therefore, the CTF algorithm can update the offloading strategy in the dynamic MEC system with less time overhead and deliver better results.

6. Conclusion

Herein, we studied the problem of temporal and positional computation offloading in a dynamic MEC system with dependent tasks. Unlike previous studies, we also considered the impact of dependencies on the offloading positions of tasks and calculated the offloading moments of tasks accurately to reduce channel interference. To cope with dynamic tasks, we proposed the CTF algorithm, which updates the offloading strategy in real time to ensure the most effective computation offloading. Simulation experiments were conducted, and the results showed that the CTF algorithm reduces significantly the time to complete all tasks and incurs less time overhead.

Data Availability

The simulated data used to support the findings of this study are included within the article, In the fifth chapter of our manuscript, "Experiments and Evaluation," we introduce in detail how to generate the simulation data needed for the experiment. Any authors can reproduce the experiment according to our data generation method, so we do not give additional experimental data.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This research was supported partially by the National Natural Science Foundation of China under Grant Nos. 62002377, 62072424, 61772546, 61625205, 61632010, 61751211, 61772488, and 61520106007; Key Research Program of Frontier Sciences, Chinese Academy of Sciences (CAS), No. QYZDY-SSW-JSC002; NSFC with No. NSF ECCS-1247944 and NSF CNS 1526638; and in part by the National Key Research and Development Plan Nos. 2017YFB0801702 and 2018YFB1004704.

References

[1] X. Fan, C. Xiang, C. Chen et al., "BuildSenSys: reusing building sensing data for traffic prediction with cross-domain learning," *IEEE Transactions on Mobile Computing*, vol. 20, no. 6, pp. 2154–2171, 2021.

[2] C. You, K. Huang, and H. Chae, "Energy efficient mobile cloud computing powered by wireless energy transfer," *IEEE Journal on Selected Areas in Communications*, vol. 34, no. 5, pp. 1757–1771, 2016.

[3] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, "Internet of Things (IoT): a vision, architectural elements, and future directions," *Future Generation Computer Systems*, vol. 29, no. 7, pp. 1645–1660, 2013.

[4] I. Lee and K. Lee, "The Internet of Things (IoT): applications, investments, and challenges for enterprises," *Business Horizons*, vol. 58, no. 4, pp. 431–440, 2015.

[5] P. Mell and T. Grance, *The Nist Definition of Cloud Computing*, National Institute of Standards and Technology, NIST Special Publication 800-145, 2011.

[6] M. Armbrust, A. Fox, R. Griffith et al., "Above the clouds: A berkeley view of cloud computing," *Science*, vol. 53, no. 4, pp. 50–58, 2010.

[7] H. T. Dinh, C. Lee, D. Niyato, and P. Wang, "A survey of mobile cloud computing: architecture, applications, and approaches," *Wireless Communications and Mobile Computing*, vol. 13, no. 18, 1611 pages, 2013.

[8] S. Deng, Z. Xiang, P. Zhao et al., "Dynamical resource allocation in edge for trustable Internet-of-Things systems: a reinforcement learning method," *IEEE Transactions on Industrial Informatics*, vol. 16, no. 9, pp. 6103–6113, 2020.

[9] W. Lu, Z. Ren, J. Xu, and S. Chen, "Edge blockchain assisted lightweight privacy-preserving data aggregation for smart grid," *IEEE Transactions on Network and Service Management*, vol. 18, no. 2, pp. 1246–1259, 2021.

[10] Q. Ye, H. Bie, K. C. Li et al., "EdgeLoc: a robust and real-time localization system towards heterogeneous iot devices," *IEEE Internet of Things Journal*, 2021.

[11] T. Wu, X. Fan, Y. Qu, and P. Yang, "Mobiedge: mobile service provisioning for edge clouds with timevarying service demands," in *2021 IEEE 27th International Conference on Parallel and Distributed Systems (ICPADS)*, pp. 1–6, Beijing, China, 2021.

[12] M. Wang, T. Ma, T. Wu, C. Chang, F. Yang, and H. Wang, "Dependency-aware dynamic task scheduling in mobile-edge computing," in *2020 16th International Conference on Mobility, Sensing and Networking (MSN)*, pp. 785–790, Tokyo, Japan, December 2020.

[13] W. Lu, S. Zhang, J. Xu, D. Yang, and L. Xu, "Truthful multiresource transaction mechanism for P2P task offloading based on edge computing," *IEEE Transactions on Vehicular Technology*, vol. 70, no. 6, pp. 6122–6135, 2021.

[14] C. Zhang, H. Tan, H. Huang et al., "Online dispatching and scheduling of jobs with heterogeneous utilities in edge computing," in *Proceedings of the Twenty-First International Symposium on Theory, Algorithmic Foundations, and Protocol*

Design for Mobile Networks and Mobile Computing, pp. 101–110, New York, NY, USA, October 2020.

[15] C. Yi, J. Cai, and Z. Su, "A multi-user mobile computation offloading and transmission scheduling mechanism for delay-sensitive applications," *IEEE Transactions on Mobile Computing*, vol. 19, no. 1, pp. 29–43, 2020.

[16] Y. Nan, W. Li, W. Bao et al., "Adaptive energy-aware computation offloading for cloud of things systems," *IEEE Access*, vol. 5, pp. 23947–23957, 2017.

[17] X. Fan, P. Yang, and Q. Li, "Fairness counts: Simple task allocation scheme for balanced crowdsourcing networks," in *2015 11th International Conference on Mobile Ad-hoc and Sensor Networks (MSN)*, pp. 258–263, Shenzhen, China, December 2015.

[18] X. Liu and N. Ansari, *Green/energy-efficient design for D2D*, American Cancer Society, 2019.

[19] H. Liu, L. Cao, T. Pei, Q. Deng, and J. Zhu, "A fast algorithm for energy-saving offloading With Reliability and latency requirements in multi-access edge computing," *IEEE Access*, vol. 8, pp. 151–161, 2020.

[20] X. Lan, L. Cai, and Q. Chen, "Execution latency and energy consumption tradeoff in mobile-edge computing systems," in *2019 IEEE/CIC International Conference on Communications in China (ICCC)*, pp. 123–128, Changchun, China, August 2019.

[21] X. Fan, X. He, D. Puthal et al., "CTOM: collaborative task offloading mechanism for mobile cloudlet networks," in *2018 IEEE International Conference on Communications (ICC)*, pp. 1–6, Kansas City, MO, USA, May 2018.

[22] L. Tianze, W. Muqing, Z. Min, and L. Wenxing, "An overhead-optimizing task scheduling strategy for ad-hoc based mobile edge computing," *IEEE Access*, vol. 5, pp. 5609–5622, 2017.

[23] M. Chen and Y. Hao, "Task offloading for mobile edge computing in software defined ultra-dense network," *IEEE Journal on Selected Areas in Communications*, vol. 36, no. 3, pp. 587–597, 2018.

[24] X. Ma, C. Lin, X. Xiang, and C. Chen, "Game-theoretic analysis of computation offloading for cloudlet-based mobile cloud computing," in *Proceedings of the 18th ACM International Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems*, pp. 271–278, New York, NY, USA, November 2015.

[25] S. Deng, Z. Xiang, J. Taheri et al., "Optimal application deployment in resource constrained distributed edges," *IEEE Transactions on Mobile Computing*, vol. 20, no. 5, pp. 1907–1923, 2021.

[26] W. Xu, X. Fan, T. Wu, Y. Xi, P. Yang, and C. Tian, "Interest users cumulatively in your ads: a near optimal study for Wi-Fi advertisement scheduling," in *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pp. 1–6, Vancouver, BC, Canada, May 2021.

[27] L. Chen, J. Wu, J. Zhang, H. N. Dai, X. Long, and M. Yao, "Dependency-aware computation offloading for mobile edge computing with edge-cloud cooperation," *IEEE Transactions on Cloud Computing*, p. 1, 2020.

[28] S. Pan, Z. Zhang, Z. Zhang, and D. Zeng, "Dependency-Aware computation offloading in mobile edge computing: a reinforcement learning approach," *IEEE Access*, vol. 7, pp. 134742–134753, 2019.

[29] J. Yan, S. Bi, Y. J. Zhang, and M. Tao, "Optimal task offloading and resource allocation in mobile-edge computing with inter-user task dependency," *IEEE Transactions on Wireless Communications*, vol. 19, no. 1, pp. 235–250, 2020.

[30] L. Yang, H. Zhang, X. Li, H. Ji, and V. C. M. Leung, "A distributed computation offloading strategy in small-cell networks integrated with mobile edge computing," *IEEE/ACM Transactions on Networking*, vol. 26, no. 6, pp. 2762–2773, 2018.

[31] C. Shu, Z. Zhao, Y. Han, G. Min, and H. Duan, "Multiuser offloading for edge computing networks: a dependency-aware and latency-optimal approach," *IEEE Internet of Things Journal*, vol. 7, no. 3, pp. 1678–1689, 2020.

[32] Y. Chen, N. Zhang, Y. Zhang, and X. Chen, "Dynamic computation offloading in edge computing for internet of things," *IEEE Internet of Things Journal*, vol. 6, no. 3, pp. 4242–4251, 2019.

[33] M. R. Garey, D. S. Johnson, and R. Sethi, "The complexity of flowshop and jobshop scheduling," *Mathematics of Operations Research*, vol. 1, no. 2, pp. 117–129, 1976.

[34] A. Soukhal, A. Oulamara, and P. Martineau, "Complexity of flow shop scheduling problems with transportation constraints," *European Journal of Operational Research*, vol. 161, no. 1, pp. 32–41, 2005.