

# QoS-aware Dynamic Service Caching and Updating in Cost-efficient Multi-Access Edge Computing

Shuaibing Lu\*, Xin Jin\*, Jie Wu<sup>†</sup>, Shuyang Zhou<sup>‡</sup>, Shen Wu\*, Jackson Yang<sup>‡</sup>, and Ran Yan\*

\*College of Computer Science, Beijing University of Technology, Beijing, China

<sup>†</sup>Center for Networked Computing, Temple University, USA

<sup>‡</sup>School of Software Engineering, Beijing Jiaotong University, Beijing, China

**Abstract**—In the context of mobile edge computing, achieving dynamic service caching and updating to guarantee the QoS of users and reduce system costs is a challenging problem. However, existing research still has certain deficiencies in considering the dynamic behavior of users and the limited storage resources of edge servers. To address this problem, this paper proposes three novel strategies for the different stages of service caching and updating to jointly optimize the delay and cost. At the initial service caching stage, we propose a caching strategy based on dynamic programming, taking into account the constraint of limited memory resources. Given the dynamic behavior of users, we formulate the joint optimization problem as a Markov Decision Process (MDP) and design a service extension strategy based on Q-learning at the service updating decision-making stage and a replacement strategy taking both the distribution of service replications and service access frequency into account at the service updating replacement stage to guarantee the QoS of users. We effectively tackle the challenges arising from the dynamic behavior of users and limited storage resources. Through extensive comparative experiments, our approach outperforms traditional strategies by significantly reducing user latency and system cost.

**Index Terms**—QoS-aware, dynamic service caching, service updating, multi-access edge computing

## I. INTRODUCTION

With the proliferation of smartphones, IoT devices, and other connected gadgets, the demand for real-time data and services continues to grow. Services such as video streaming, online gaming, and IoT applications require lower latency and higher bandwidth, posing significant challenges to the bandwidth of the backbone network and cloud data center. The traditional high-latency centralized cloud center model may result in longer service response times, and may not be able to meet user demands when network congestion occurs. In order to address these challenges, multi-access edge computing has emerged as an appealing solution that deploys edge servers at the network edge, bringing computation and storage closer to end-users and devices, thereby enabling low-latency and high-bandwidth service delivery. However, as the amount of services and users continues to burgeon, and users exhibit increasingly dynamic behavior, the effective management of service caching and updating on edge servers presents a formidable challenge. Traditional caching strategies often lead to a waste of resources, while inflexible updating mechanisms may fail to ensure the Quality of Service (QoS) required

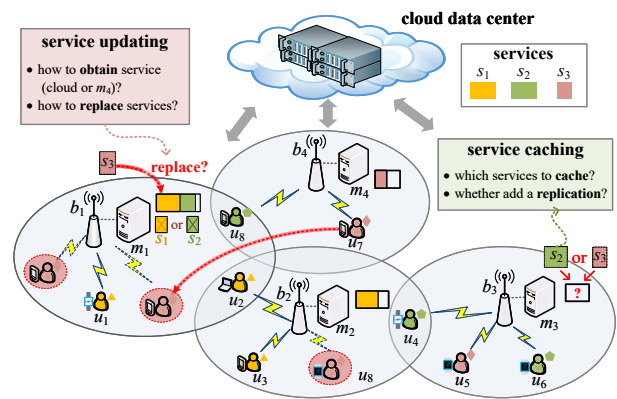


Fig. 1. Example of service caching and updating strategies in the multi-access edge computing network.

by users. Consequently, there arises a critical imperative to develop a QoS-aware dynamic service caching and updating approach that not only caters to users' QoS requirements but also maintains cost efficiency. The limited storage resources of edge servers and the dynamic behavior of users pose challenges to service deployment.

In this paper, we propose a novel framework with three main strategies to address the service caching and updating problem. These strategies are optimized based on the constraints of server storage resources to ensure QoS for users and reduce system costs under dynamic user behavior. Our contributions can be summarized as follows:

- We investigate the dynamic service caching and updating problem with the objective of minimizing the system cost and user latency. We design the QoS model and the cost model for user latency and system cost.
- We develop three algorithms to solve the problem in three phases of service caching and updating. For the first stage of service provisioning, we propose an initial service caching algorithm based on dynamic programming. Since the users requesting services change over time, we formulate the joint optimization problem as an MDP and propose a decision strategy for selecting services to extend and servers to provide replications based on

Q-learning. After entering the caching update stage, we develop a novel service replacement strategy to update the services to be cached.

- We conduct extensive experiments to compare our algorithms with several baselines. In these experiments, we evaluate our algorithm alongside others using three groups of datasets with varying scales. The experimental results demonstrate the effectiveness of the proposed service caching and updating algorithms across various weight settings and datasets. The algorithms we designed effectively provide the optimal service caching and updating strategy, guaranteeing the QoS of users and reducing the system cost.

The structure of the remaining sections of this paper is organized as follows. Section II surveys related work. Section III introduces the model and presents the problem. Section IV explores strategies for addressing service caching and updating problems. Section V presents the experimental results. Finally, Section VI provides a summary of the entire paper.

## II. RELATED WORK

Due to the dynamic user behavior and the limited and heterogeneous computing and storage capacities of edge devices, caching and updating of services is required to guarantee user QoS and optimize costs. Various works have studied this problem from different aspects. Somesula et al. [1] presented a greedy and randomized rounding technique to solve the service caching and request routing problems. Ko et al. [2] decoupled computation offloading and service caching with limited source and latency by considering the preferences of users. Zhong et al. [3] replaced the least utilized service to implement dynamic cache updates. Wang et al. [4] formulated the service placement as an integer linear programming program. Chi et al. [5] proposed an edge service entity placement model, which aims to jointly optimize service delay and analysis quality for MVR systems. Cheng et al. [6] introduced a novel two-stage adaptive robust model. Li et al. [7] proposed a two-stage caching approach that aims to reduce delay for mobile video-on-demand users. Sun et al. [8] proposed a decentralized, recommendation-enabled edge caching system. However, in these works, the service is not measured from the perspective of size, storage cost, and optimizable latency.

To address the challenge of dynamic user behavior, some existing work based on service updating has been proposed. Qiu et al. [9] proposed a cache replacement algorithm based on the oracle approximation named OA-Cache in an end-to-end manner to maximize the cache hit rate. Liu et al. [10] proposed a deep Q-network (DQN)-based solution to jointly optimize the hybrid caching data selection of UAV and the task offloading strategy of users. Li et al. [11] proposed two modules Delayed-Eviction and Unified-Standard for existing caching replacement strategies to improve the caching performance. Zhou et al. [12] modeled the cache replacement process as an MDP and used a distributed edge caching

method for intelligent caching. Zhang et al. [13] introduced a novel digital twin-assisted optimization framework to ensure reliable caching in next wireless networks. Wu et al. [14] proposed a Recursive Deep Reinforcement Learning based Collaborative Caching Relay strategy. Li et al. [15], Wang et al. [16] and Anokye et al. [17] proposed solutions based on deep reinforcement learning. Jiang et al. [18] propose a balance-aware fee-incentivized routing algorithm. However, these works do not consider the impact of churn probability and service replication distribution state when developing service caching and updating strategies.

In this paper, we investigate strategies for caching services and updating services, considering the distribution of service replications and the frequency of service access under limited resources, with the intention of optimizing user QoS and system cost.

## III. PROBLEM FORMULATION

### A. System model

As shown in Figure 1, we consider a three-layer network architecture that is composed of a cloud data center, edge servers, and mobile end users. We assume that the services required by the users have been previously provisioned on the cloud data center by the operators, represented as set  $\mathbf{S} = \{s_1, s_2, \dots, s_j\}$ . For each service requested by users, we use a tuple to record the information as  $s_j = (c_{s_j}, \varphi_{s_j})$ , where  $c_{s_j}$  is the storage capacity occupied by  $s_j$  and  $\varphi_{s_j}$  is the storage cost on an edge server. In order to expand the scale of service areas, we suppose that each service can have multiple replications on several edge servers. Here, we use  $N(s_j)$  to represent the total number of replications where  $N(s_j) = \sum_{\forall m_k \in \mathbf{M}} x(m_k, s_j)$ . We define  $x(m_k, s_j) \in \{0, 1\}$  which denotes the number of replications of  $s_j$  located on server  $m_k$ . The caching cost is  $\varphi_{s_j}$  for each replication of service  $s_j$ . We use the notation  $\mathbf{c}$  to denote the cloud data center, and the computation resource is  $R_{\mathbf{c}}^{cal}$ . In the edge layer, the substrate set of edge servers that the operators support is indicated by the notation  $\mathbf{M} = \{m_1, m_2, \dots, m_k\}$ . The amount of storage resources of edge server  $m_k$  is represented by  $R_{m_k}^s$ . In the user layer, we use the notation  $\mathbf{U} = \{u_1, u_2, \dots, u_i\}$  to represent the set of users, which require services. For each user, we use a triplet  $u_i = (q_{u_i}, z_{u_i}^{cal}, \kappa_{u_i})$  to capture the user's information. Here  $q_{u_i}$  stands for the service that  $u_i$  requests,  $z_{u_i}^{cal}$  for the amount of task data and  $\kappa_{u_i}$  for the server on which  $u_i$  is located.

### B. QoS model

We present a QoS model for users that is composed of delays in computation and communication. We consider the problem of caching and updating services in a three-tier edge cloud topology, where all services provided by operators have already been pre-provisioned in the cloud. Some services are selectively provisioned on edge servers. When users request services that are provided both in the cloud and on edge

servers, we prefer to use the services from edge servers to process the tasks. We use  $\mathbf{D}$  to denote the total delay, which is composed of two parts: the computational delay  $d_{u_i}^{cal}$  and the communication delay  $d_{u_i}^{comm}$ , which can be calculated as  $\mathbf{D} = \sum_{u_i \in \mathbf{U}} (d_{u_i}^{cal} + d_{u_i}^{comm})$ . For the first part  $d_{u_i}^{cal}$ , there may be different delays when using the service due to the different locations of the service. Suppose the service that user  $u_i$  is accessing has already been deployed on the edge server  $m_k$ . In that case, the computation delay on the edge server for each user is defined as  $d_{u_i}^{cal(m_k)} = z_{u_i}^{cal} / R_{m_k}^{cal}$ , where  $z_{u_i}^{cal}$  is the amount of the task data of user  $u_i$  and  $R_{m_k}^{cal}$  represents the computation power of the edge server  $m_k$ . However, suppose the service that user  $u_i$  is accessing is not deployed on the edge server. In that case, the user must access the cloud to use the service, and the computation delay is given by  $d_{u_i}^{cal(c)} = z_{u_i}^{cal} / R_c^{cal}$ , where  $R_c^{cal}$  is the computation power of the cloud. Therefore, the computation delay of  $u_i$  will be  $d_{u_i}^{cal} = d_{u_i}^{cal(m_k)} \cdot x(m_k, s_j) + d_{u_i}^{cal(c)} \cdot (1 - x(m_k, s_j))$ , where  $x(m_k, s_j)$  is denoted as the status indicating whether  $s_j$  requested by  $u_i$  located on edge server  $m_k (\forall m_k \in \mathbf{M})$ , i.e.,  $x(m_k, s_j) \in \{0, 1\}$ .

The second part of the communication delay  $d_{u_i}^{comm}$  can be determined similarly. If the service is located on an edge server, the communication delay from the user  $u_i$  to the edge server  $m_k$  can be calculated by  $d_{u_i}^{comm(m_k)} = z_{u_i}^{cal} \cdot l_{(m_k, m_{k'})} / \lambda_{(m_k, m_{k'})}$ , where  $l_{(m_k, m_{k'})}$  and  $\lambda_{(m_k, m_{k'})}$  are the network propagation distance and the transmission speed between the edge server located near the user and the edge server providing the service. Otherwise, the user needs to obtain services from the cloud. The communication delay will be  $d_{u_i}^{comm(c)} = z_{u_i}^{cal} \cdot l_c / \eta \cdot b_c$ , where  $b_c$  and  $l_c$  are the bandwidth and network propagation distance from users to cloud, and  $\eta$  is the efficiency related to  $b_c$ . Therefore, the communication delay of  $u_i$  will be  $d_{u_i}^{comm} = d_{u_i}^{comm(m_k)} \cdot x(m_k, s_j) + d_{u_i}^{comm(c)} \cdot (1 - x(m_k, s_j))$ , where  $\forall m_k \in \mathbf{M}$ .

### C. Cost Model

The costs that play a role in this work include above all the costs of storing services  $C^{stor}$ , which are incurred by caching services on edge servers, and the migration costs  $C^{migr}$ , which are incurred by the migration of service applications when updating services. The total cost is calculated as  $\mathbf{C} = \sum_{s_j \in \mathbf{S}} (C_{s_j}^{stor} + C_{s_j}^{migr})$ . For service  $s_j$ , the storage cost on one edge server is  $\varphi_{s_j}$ , and the number of replications of  $s_j$  in the system is  $N_{s_j}$ , so the total storage cost of  $s_j$  can be defined as  $C_{s_j}^{stor} = N_{s_j} \cdot \varphi_{s_j}$ . If the edge server at the user's location does not have the requested service, it can request the migration of the service from a nearby node that has already placed it. The migration cost of  $s_j$  from edge server  $m_{k'}$  to  $m_k$  can be calculated as  $C_{s_j}^{migr(m_k, m_{k'})} = \gamma_m \cdot c_{s_j} \cdot l_{(m_k, m_{k'})}$ . While the destination server is the cloud server, the migration cost will be  $C_{s_j}^{migr(m_k, c)} = \gamma_c \cdot c_{s_j} \cdot l_c$ , where  $c_{s_j}$  is the storage capacity occupied by  $s_j$ , and  $l_c$  is the network propagation

distance between *cloud* and  $m_k$ . We use  $\gamma_m$  and  $\gamma_c$  to represent the coefficients about migrating to edge servers and the cloud, respectively.

### D. Problem Formulation

Given the system cost and user QoS, our research objective is to design a dynamic service caching strategy that jointly minimizes the total delay and cost. Thus, our optimization objective is:

$$\mathbf{P}_1 : \min_{x(m_k, s_j)} \{\mathbf{D}, \mathbf{C}\} \quad (1)$$

$$\text{s.t.} \sum_{s_j \in \mathbf{S}_{m_k}} c_{s_j} \leq R_{m_k}^s, \quad \forall m_k \in \mathbf{M} \quad (2)$$

$$x(m_k, s_j) \in \{0, 1\}, \quad \forall m_k \in \mathbf{M}, \forall s_j \in \mathbf{S} \quad (3)$$

where  $\mathbf{S}_{m_k}$  represents the set of services cached on edge server  $m_k$ . The constraint (2) means that the total capacity occupied by services placed on the server  $m_k$  is no more than its storage resources. The constraint (3) restricts the maximum number of replications of one service on the same server to one. Since  $\mathbf{P}_1$  is a multi-objective optimization problem, we can assign different weights to latency  $\mathbf{D}$  and cost  $\mathbf{C}$  based on preferences for QoS and cost, aiming to minimize the weighted sum of latency and cost to solve this problem. Thus, the optimization objective  $\mathbf{P}_1$  can be transformed into  $\mathbf{P}_2$ :

$$\mathbf{P}_2 : \min_{x(m_k, s_j)} \{\alpha \mathbf{D} + \beta \mathbf{C}\} \quad (4)$$

$$\text{s.t. (2)(3)}$$

where  $\alpha$  and  $\beta$  are the weight corresponding to  $\mathbf{D}$  and  $\mathbf{C}$ .

## IV. ALGORITHM DESIGN

### A. Initial Service Caching

The initial service caching problem aims to cache appropriate services on each server to minimize the latency and cost of serving users on servers with limited storage resources. We transform the initial service caching problem into a knapsack problem by introducing the variable of service value, which involves selecting appropriate services with different sizes to maximize the total value of the edge server. Based on the analysis of the area covered by an edge server, we propose the service value to evaluate the value of the requested services in this area at the current time, which can apply to synchronously or asynchronously arriving users. The optimization objective of the algorithm is to maximize the value obtained within the limited capacity for each edge server by adjusting the caching scheme. For each server, we use  $\Psi(s_j)$  to denote the value of  $s_j$ , which differs depending on whether it is cached on the server or not.

**Definition 1 (service value):** Let  $\Psi(s_j)$  indicate the service value of  $s_j$  during the initial caching process, where  $\Psi(s_j) = -\alpha \sum_{u_i \in \mathbf{U}(s_j)} (d_{u_i}^{m_k} \cdot x(m_k, s_j) - d_{u_i}^c \cdot (1 - x(m_k, s_j))) - \beta \varphi_{s_j} \cdot x(m_k, s_j)$ ,  $\mathbf{U}(s_j)$  represents the set of users requesting

---

**Algorithm 1** USC algorithm

---

**Require:**  $\mathbf{S}, \mathbf{M}, \mathbf{U}$ .**Ensure:** initial service caching strategy  $\mathbf{I}$ .

```
1: initial  $\mathbf{I} = \emptyset$ ;  
2: for  $m_k \in \mathbf{M}$  do  
3:   Construct and initialize  $V$  table;  
4:   for  $s_j \in \mathbf{S}$  do  
5:     Calculate the service value  $\Psi_{s_j}$ ;  
6:     for  $r \leftarrow 0$  to  $R_{m_k}^s$  do  
7:       if  $c_{s_j} \leq r$  then  
8:          $V[s_j][r] = \max\{V[s_{j-1}][r] +$   
9:            $\Psi^n(s_j), V[s_{j-1}][r - c_{s_j}] + \Psi^y(s_j)\}$ ;  
10:        else  
11:           $V[s_j][r] = V[s_{j-1}][r] + \Psi^n(s_j)$   
12:        end  
13:   Backtrack  $V$  to obtain the optimal scheme  $I(m_k)$ ;  
14:    $\mathbf{I} = \mathbf{I} \cup I(m_k)$ ;  
15: Return  $\mathbf{I}$ 
```

---

$s_j$ ,  $d_{u_i}^{m_k}$  is the total delay of using replications from  $m_k$ , and  $d_{u_i}^c$  is delay from the cloud.

Here, we use  $x(m_k, s_j)$  to represent whether to cache  $s_j$  on server  $m_k$ . When  $s_j$  is chosen to cache on  $m_k$ , i.e.,  $x(m_k, s_j) = 1$ , the service value will be  $\Psi^y(s_j) = -\alpha \sum_{u_i \in \mathbf{U}(s_j)} d_{u_i}^{m_k} \cdot x(m_k, s_j) - \beta \varphi_{s_j} \cdot x(m_k, s_j)$ . On the contrary, if  $s_j$  is not chosen as a cache for  $m_k$ , i.e.,  $x(m_k, s_j) = 0$ , there is still a service value that is  $\Psi^n(s_j) = -\alpha \sum_{u_i \in \mathbf{U}(s_j)} d_{u_i}^c \cdot (1 - x(m_k, s_j))$ . Based on that, we can use dynamic programming to find the optimal service caching strategy for each edge server within the constraints of storage resources, maximizing the value of the servers. Based on the dynamic programming approach, we decompose the target problem into subproblems that find a caching strategy to maximize the value obtained within the constraints of limited capacities less than its most storage resources to solve it.

In this subsection, we first introduce a novel algorithm to solve the initial service caching problem called USC (User-oriented Service Caching) for multiple users in the edge network. The algorithm aims to find an optimal caching scheme for each edge server based on dynamic programming. The individual steps are shown in Algorithm 1. We need to make some preparations for the algorithm, which include a set of edge servers  $\mathbf{M}$  and the information about the requested services  $\mathbf{S}$ . We also require information concerning the set of users  $\mathbf{U}$  requesting services that are distributed in each area, encompassing details such as task sizes, as well as the computational power and memory resources allocated to each edge server. We initialize the initial placement strategy  $\mathbf{I}$  and set it to empty in line 1. We use the dynamic programming method on each server to obtain the best service provisioning strategy, and the strategies of the edge servers together constitute the initial placement strategy  $\mathbf{I}$ . For each edge server, we construct the value table  $V$  to store the value obtained with

---

**Algorithm 2** PSS algorithm

---

**Require:**  $\mathbf{RS}_{m_k}, \mathbf{U}, \mathbf{I}$ .**Ensure:**  $\mathbf{US}_{m_k}, \mathbf{F}(\mathbf{US}_{m_k})$ .

```
1: for  $s_j \in \mathbf{RS}_{m_k}$  do  
2:   Set  $\mathbf{f}(s_j)$  according to  $\mathbf{I}$ ;  
3:   Calculate latency response gain  $\Delta_{m_k}^{s_j}$ ;  
4:   if  $\Delta_{m_k}^{s_j} > 0$  then  
5:     Calculate benefit-cost ratio  $H_{m_k}^{s_j}$ ;  
6:     if  $H_{m_k}^{s_j} > H_0$  then  
7:        $\mathbf{US}_{m_k} = \mathbf{US}_{m_k} \cup s_j$ ;  
8:   for  $s_j \in \mathbf{US}_{m_k}$  do  
9:      $\mathbf{F}(\mathbf{US}_{m_k}) = \mathbf{F}(\mathbf{US}_{m_k}) \cup \mathbf{f}(s_j)$ ;  
10: Return  $\mathbf{US}_{m_k}, \mathbf{F}(\mathbf{US}_{m_k})$ .
```

---

different capacities in line 3. In line 5, we calculate  $\Psi(s_j)$  for each service requested at the edge server based on the service value. Then, we update the table  $V$  according to the recursion relationship between the subproblems in lines 6 to 10. In this step, we use  $V[s_j][r]$  to denote the maximum value obtained by introducing  $s_j$  under the constraint of capacity  $r$ , which is equal to the higher value between  $V[s_{j-1}][r - c_{s_j}] + \Psi^y(s_j)$  and  $V[s_{j-1}][r] + \Psi^n(s_j)$ , corresponding to caching  $s_j$  and not. When we finalize the  $V$  table of an edge server, we can trace it back to obtain the optimal scheme  $I(m_k)$  of the edge server in line 11. We merge all the optimal schemes of servers to acquire the initial service caching strategy  $\mathbf{I}$  for the system in line 12.

The time complexity of Algorithm 1 is  $O(|\mathbf{M}| \times |\mathbf{S}| \times (\hat{R} + 1))$ . The time complexity of this algorithm primarily depends on the number of iterations in three loops. Since we consider a single server when dividing the subproblem, we have to run through all servers in the set  $\mathbf{M}$  for the outer loop, which corresponds to  $|\mathbf{M}|$  times the number of edge servers. On this basis, we consider that all services can be placed on each server, representing  $|\mathbf{S}|$  times for the second loop representing the number of services. For each server, we use  $\hat{R} = \max\{R_{m_k}^s | m_k \in \mathbf{M}\}$  to represent the maximum storage capacity of the server in the set  $\mathbf{M}$ . The third loop iterates a maximum of  $\hat{R} + 1$  times, which represents the number of steps taken by the edge server with the largest capacity. At the step of backtracking, it needs to search the cache state of each service in the  $V$  table, which corresponds to  $|\mathbf{S}|$  times. In the algorithm, operations other than these loops can be regarded as constant in time and therefore negligible. Consequently, the time complexity of the algorithm is  $O(|\mathbf{M}| \times |\mathbf{S}| \times (\hat{R} + 1))$ .

### B. Service updating decision-making

Due to the dynamic behavior of users, new service requests will appear in the area. To guarantee the QoS of users, we will consider caching new services on the edge servers by migrating service replication from other servers. However, we also need to consider the cost. We use the Markov Decision Process (MDP) to describe the situation. The state space

---

**Algorithm 3** SUD algorithm

---

**Require:**  $\mathbf{US}'_{m_k}, \mathbf{F}(\mathbf{US}_{m_k})$ .**Ensure:**  $\mathbf{US}_{m_k}, \mathbf{F}'(\mathbf{US}_{m_k})$ 

```
1: for  $s_j \in \mathbf{US}_{m_k}$  do
2:   if select  $s_j$  then
3:      $z \leftarrow s \cup s_j$ ;
4: for  $s_j \in s$  do
5:   for  $m_k \in \mathbf{f}(s_j)$  do
6:     if select  $m_k$  then
7:        $a_{s_j} \leftarrow m_k$ ;
8:    $a \leftarrow a \cup a_{s_j}$ ;
9: Initialize  $Q(z, a)$ ;
10: for  $episode \leftarrow 1$  to  $MAX$  do
11:   while not terminal do
12:     if  $Random() < \varepsilon$  then
13:       Select a random action  $a(z_\tau)$ ;
14:     else
15:        $a(z_\tau) = \arg \max Q(z_\tau, a)$ ;
16:     Take action  $a(z_\tau)$  and obtain  $z_{\tau+1}$ , reward
17:     Update  $Q(z, a)$ ;
18:      $z_\tau = z_{\tau+1}$ ;
19: Update  $\mathbf{US}'_{m_k}, \mathbf{F}'(\mathbf{US}_{m_k})$ ;
20: Return  $\mathbf{US}'_{m_k}, \mathbf{F}'(\mathbf{US}_{m_k})$ 
```

---

$\mathbf{Z}$  defines all the possible states of service selection, and we use  $z_\tau = [s_1, s_2, \dots, s_j]_\tau$  to represent the state at step  $\tau$  consisting of the selected services. The action space  $\mathbf{A}$  defines all possible states of server selection, where  $a_\tau = [\mathbf{M}(s_1), \mathbf{M}(s_2), \dots, \mathbf{M}(s_j)]$  represents the action performed in state  $\tau$ , which consists of the selected servers from replication groups for each service. Here we use  $\mathbf{M}(s_j) = [\zeta(m_k)]_{m_k \in \mathbf{M}}$  to specify the set of all servers that place  $s_j$ , and  $\zeta(m_k)$  is an indicator function with  $\zeta(m_k) = 1$  if the replication of service  $s_j$  has been placed on  $m_k$  and  $\zeta(m_k) = 0$  if it has not. We use  $P$  to denote the probability of users reaching other servers, and the reciprocal of the total value that providing services to users as reward  $r_\tau$ , where  $r_\tau = 1/(\alpha\mathbf{D} + \beta\mathbf{C})$ . The decay factor is defined as  $\gamma \in [0, 1]$ .

To find an elegant solution, we solve the problem in two steps: first, we set an appropriate threshold to screen services preliminarily; second, we use the Q-learning algorithm for further selecting services and choosing a suitable edge server to provide the replication for the services selected. In order to reduce the difficulty of subsequent selection, we designed a simple algorithm called PSS (Primary Services Selection) for preliminary selection. The specific steps are shown in Algorithm 2. For each service, we set a replication group  $\mathbf{f}(s_j)$ , which consists of edge servers that have cached  $s_j$  at the current time in line 2. We introduce latency response gain to evaluate the benefit of adding a replication for the service. We calculate the  $\Delta_{m_k}^{s_j}$  for each service in line 3.

---

**Algorithm 4** SRD algorithm

---

**Require:**  $\mathbf{S}_{m_k}, \mathbf{US}'_{m_k}, \mathbf{F}'(\mathbf{US}_{m_k})$ .**Ensure:** Service updating strategy  $\hat{\mathbf{I}}$ .

```
1: for  $s_j \in \mathbf{US}'_{m_k}$  do
2:   Calculate the remaining capacity  $R_{m_k}^{re}$ ;
3:   if  $c_{s_j} > R_{m_k}^{re}$  then
4:     Construct candidate services set  $\mathbf{CS}_{m_k} = \emptyset$ ;
5:     for  $s_{j'} \in \mathbf{S}_{m_k}$  do
6:       Calculate replication density  $\rho_{s_{j'}}^{m_k}$  of  $s_{j'}$ ;
7:       if  $\rho_{s_{j'}}^{m_k} > \rho_0^{m_k}$  then
8:          $\mathbf{CS}_{m_k} = \mathbf{CS}_{m_k} \cup s_{j'}$ ;
9:     for  $s_l \in \mathbf{CS}_{m_k}$  do
10:      Calculate  $v_{s_l}$ ;
11:     while  $c_{s_j} > R_{m_k}^{re}$  do
12:        $s_r = \arg \min \{v_{s_l} | s_l \in \mathbf{CS}_{m_k}\}$ ;
13:       Update  $\mathbf{S}_{m_k} = \mathbf{S}_{m_k} / s_r$ ;
14:       Update  $\mathbf{CS}_{m_k} = \mathbf{CS}_{m_k} / s_r$ ;
15:        $R_{m_k}^{re} = R_{m_k}^{re} + c_{s_r}$ ;
16:    $\mathbf{S}_{m_k} = \mathbf{S}_{m_k} \cup s_j$ ;
17: Return  $\hat{\mathbf{I}}$ 
```

---

**Definition 2** (latency response gain): Let  $\Delta_{m_k}^{s_j}$  be the latency response gain of the service  $s_j$  requested from the edge server  $m_k$ , and  $\Delta_{m_k}^{s_j} = T_{m_k, s_j} - T'_{m_k, s_j}$ , where  $T_{m_k, s_j}$  is the total latency of all users, who request  $s_j$  to  $m_k$  without  $s_j$  being cached on  $m_k$ , and  $T'_{m_k, s_j}$  is the total latency of all users who request  $s_j$  to  $m_k$  after  $s_j$  has been cached on  $m_k$ .

Here,  $T_{m_k, s_j} = d_{u_i}^{m_k}(s_j)$  when there are replications of  $s_j$  near  $m_k$ , where  $m_{k'}$  is random one of the edge servers with a replication of  $s_j$  close to  $m_k$ . Alternatively, when there are no replications of  $s_j$  nearby,  $T_{m_k, s_j} = d_{u_i}^c(s_j)$  represents the delay to use  $s_j$  from cloud. On this basis, the request to extend the service replication will only be considered if the gain  $\Delta_{m_k}^{s_j}$  is positive. Then we define the benefit-cost ratio and establish an appropriate threshold  $H_0$  for primary selection, which indicates that services with a benefit-cost ratio  $H$  higher than the threshold  $H_0$  can proceed to further selection, and we use  $\mathbf{US}_{m_k}$  to denote the set of services.

**Definition 3** (benefit-cost ratio): Let  $H_{m_k}^{s_j}$  indicate the benefit-cost ratio of  $s_j$  requested to  $m_k$ , and  $H_{m_k}^{s_j} = \Delta_{m_k}^{s_j} / C_{m_k}^{s_j}$ , where  $C_{m_k}^{s_j}$  is the cost of migrating the replication from the other server of  $s_j$  to  $m_k$ .

In lines 4 to 7, we calculate  $H_{m_k}^{s_j}$  of services that satisfy  $\Delta_{m_k}^{s_j} > 0$ , and select the services that satisfy  $H_{m_k}^{s_j} > H_0$  to merge into  $\mathbf{US}_{m_k}$ . In lines 8 to 9, we obtain the replication group set  $\mathbf{F}(\mathbf{US}_{m_k})$  of  $\mathbf{US}_{m_k}$ .

For further selection, we propose a decision-making strategy for selecting the services and servers called SUD (Service Updating Decision-making based on the QoS). We select the services from  $\mathbf{US}_{m_k}$  to cache and choose an edge server for each service from  $\mathbf{F}(\mathbf{US}_{m_k})$  based on Q-learning. The specific

steps are shown in Algorithm 3. In lines 1 to 8 we construct the Q-table with the state space and the action space. We initialize the value of entries in the Q-table to zero in line 9.

**Definition 4 (Q-table):** We create a two-dimensional matrix of size  $|Z| \times |A|$  as our Q-table. We use reward to update the Q-table as follows:  $Q(z_\tau, a(z_\tau)) \leftarrow Q(z_\tau, a(z_\tau)) + \mu(r + \sigma \max_{a \in A} Q(z_{\tau+1}, a) - Q(z_\tau, a(z_\tau)))$ .

Here, the entries in the Q-table represent the expected rewards associated with taking specific actions in specific states. For each step, we select which servers to provide replications for the services by choosing the action whose value is the maximum in the current state or randomly with a certain probability in lines 12 to 15. Then we calculate the next state and the reward for taking this action at the current state and update the value of the Q-table and the state in lines 16 to 18. After iterative training, we obtain the updated Q-table, which stores the value of different selections of services and servers. Thus, we obtain the decision  $\mathbf{US}'_{m_k}$  and  $\mathbf{F}'(\mathbf{US}_{m_k})$  regarding which services to add replications and which servers to provide replications by selecting the state and action corresponding to the maximum value in line 19.

### C. Service updating replacement

After selecting which services to add replications, we need to cache the services on the edge server. However, when faced with a situation where the remaining capacity of the edge server can't support caching new services, we should update the caching scheme of the edge servers. To better describe the use of services, we have proposed a new definition that combines the access behavior of users and the probability of activity as the service evaluation indicator.

**Definition 5 (service evaluation indicator):** Let  $e_{s_j}$  indicate the service evaluation indicator of  $s_j$ , and  $e_{s_j} = \sum_{u_i \in \mathbf{U}(s_j)} \Delta_{m_k}^{s_j} \cdot (1 - p_{u_i}) / \varphi_{s_j}$ , where  $p_{u_i}$  is the probability that  $u_i$  leaves the current location.

We consider the distribution of service replications and introduce a definition of density as follows.

**Definition 6 (replication density):** Let  $\rho_{s_j}^{m_k}$  indicate the replication density of  $s_j$ , i.e., the number of replications of  $s_j$  within one hop from the edge server  $m_k$ .

To avoid the frequent exchange of highly accessed services, which can lead to multiple replacements and caching, we introduce a novel definition called access weight to measure the usage of deployed services.

**Definition 7 (visit weight):** Let  $v_{s_j}$  indicate the visit weight of  $s_j$ , and  $v_{s_j} = e_{s_j} \cdot E_1 + (rv_{max} - rv_{s_j}) \cdot E_2$ , where  $rv_{s_j}$  is the average access time interval of  $s_j$  within a certain period,  $rv_{max}$  represents the maximum access time interval among the services cached on the edge server and  $E_1$  and  $E_2$  are the weights, respectively.

By introducing the replication density  $\rho_{s_j}^{m_k}$  and the visit weight  $v_{s_j}$ , we propose a cache replacement strategy called SRD (Service Replacement based on the distribution of services). The individual steps are shown in Algorithm 4. As

inputs in Algorithm 3, we need the set of services that add replications  $\mathbf{US}'_{m_k}$ , the set of servers that provide replications  $\mathbf{F}'(\mathbf{US}_{m_k})$ , and the set of services that have been cached on the edge server  $\mathbf{S}_{m_k}$ . In line 2, we check the remaining capacity  $R_{m_k}^{re}$  of  $m_k$ . If the remaining capacity can support the caching of  $s_j$ , we update it directly in  $\mathbf{S}_{m_k}$ . Otherwise, we calculate the replication density  $\rho_{s_j}^{m_k}$  of the service cached on  $m_k$  in line 6. We use  $\mathbf{CS}_{m_k}$  to indicate the candidate set for updating the services, which consists of the services whose density is higher than the threshold  $\rho_0^{m_k}$  that we defined with Algorithm 3 in lines 3 to 8. Then we calculate the visit weights of the services in  $\mathbf{CS}_{m_k}$  in line 9. We remove the service with the minimum visit weight in  $\mathbf{CS}_{m_k}$  from  $\mathbf{S}_{m_k}$  until the remaining capacity of  $m_k$  can satisfy  $s_j$ , and update  $\mathbf{CS}_{m_k}$  in line 14. We then adjust the remaining capacity. In line 16, update  $s_j$  in  $\mathbf{S}_{m_k}$ . We complete the operations for each service in  $\mathbf{US}'_{m_k}$ , and  $\mathbf{S}_{m_k}$  that finally obtain is the updated caching strategy  $\mathbf{I}$ .

## V. EXPERIMENT

### A. Basic Setting

We conduct extensive experiments to validate the effectiveness of our algorithms. All experiments are conducted using Python 3.8 on Windows 11 with a 12th Gen Intel(R) Core(TM) i9-12900KF CPU @ 3.19 GHz, NVIDIA RTX3080 GPU, and 32GB memory. We use a dataset from China Telecom Shanghai Company [19], which contains information on 3,233 base station locations and corresponding user connections in June 2014. We randomly select subsets of 10, 20, and 30 base stations, each equipped with a server, to construct our server dataset. Subsequently, we generate random configuration information for servers, services, and users. We conduct experiments at three different scales to validate our algorithms in three stages: 1) Group 1: 10 servers, 15 services, 35 initial users, and 5 new users; 2) Group 2: 20 servers, 30 services, 70 initial users, and 10 new users; 3) Group 3: 30 servers, 45 services, 110 initial users, and 10 new users;

1) *Service caching experiment:* We introduce three greedy algorithms which are Greedy based on the Number of users for the services (GN), Greedy based on the Size of services (GS), Greedy based on the Evaluation Indicator of services (GEI), and one random algorithm (RP) to compare with our USC algorithm on datasets of different scales. The GN algorithm is a greedy algorithm based on the number of users for the services which prioritizes caching services with more users. The GS algorithm prioritizes caching services with smaller sizes greedily. Similarly, the GEI algorithm prioritizes caching services with higher service evaluation indicators greedily. The RP algorithm is a random algorithm that caches services randomly. We also adjust the settings of the weights for  $\mathbf{D}$  and  $\mathbf{C}$  to compare the performance of the algorithms. We assign weights that are in the range  $[0.1, 0.9]$  with an interval of 0.1 for  $\alpha$  and  $\beta$  respectively. The weights for  $\mathbf{D}$  and  $\mathbf{C}$  represent the preferences for user QoS and system cost.

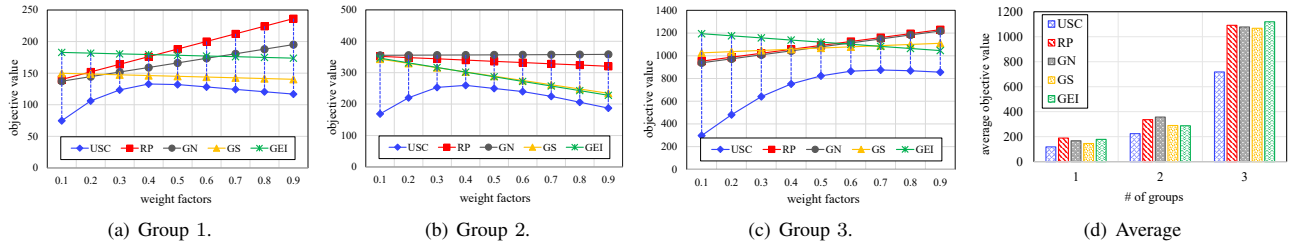


Fig. 2. Comparison of caching algorithms under different weight settings at the initial service caching stage

2) *Service updating experiment*: For service updating decision-making, we compare our SUD algorithm with two baselines, Service Updating from the Cloud (SU-C) and Service updating from the Edge or Cloud (SU-EC) on the datasets of three different scales. The former involves caching all new services and migrating service replications from the cloud, while the latter also caches all new services, but migrates service replications from the edge servers if the service has been deployed on other edge servers, or from the cloud if there are no replications of the service. As for service updating replacement, we introduce the LRU algorithm and the random algorithm in comparison with our proposed SRD algorithm. Since the effectiveness of the replacement strategies is affected by the long-term dynamic behavior, we evaluate their performance by comparing the average performance of these algorithms over five time slots.

### B. Experiment Results

We evaluate these algorithms on different scales and with different preferences to compare their performance on service caching and updating at different stages. In our experiments, servers are heterogeneous with different limited storage resources and computation resources, and it takes different storage costs and storage resources to cache services.

The experimental results of the initial service caching stage are shown in Figures 2(a), 2(b), 2(c), and 2(d). The results indicate that the USC algorithm performs the best in three scales and the higher performance of the USC algorithm becomes more and more significant as the scale of datasets increases. For instance, on a scale of 1 with 10 servers, 15 services, and 40 users, the objective value of the USC algorithm ranges from 70 to 120 across 10 sets of weights. Similarly, on a scale of 2 with 20 servers, 30 services, and 80 users, the value ranges from 160 to 230, and on a scale of 3, it ranges from 290 to 720, all of which outperform other algorithms. Figure 2(d) shows the average results of the caching algorithms across the 10 sets of weights for the three scales of the dataset, and we can see that the superiority of the USC algorithm compared to other algorithms is apparent gradually from scale 1 to scale 3. The USC algorithm stands out because the choice considers both user latency and storage costs, unlike other greedy algorithms that focus on only one aspect. The weights assigned for  $\mathbf{D}$  and  $\mathbf{C}$  represent the preference for the users' QoS and system cost.

TABLE I  
COMPARISON OF DECISION-MAKING ALGORITHMS.

Group	SUD	SU-C	SU-EC
1	48.88 ± <b>8.07</b>	98.40 ± <b>18.18</b>	92.20 ± <b>11.72</b>
2	194.66 ± <b>55.08</b>	327.14 ± <b>141.81</b>	263.31 ± <b>81.80</b>
3	493.06 ± <b>19.16</b>	859.05 ± <b>58.37</b>	577.57 ± <b>182.44</b>

Different weights will result in different performances. Taking Figure 2(a) as an example, the objective value increases as the weight of  $\mathbf{D}$  increases until 0.4, and then the value decreases. Similarly, the value is highest for the weights (0.4,0.6) in Figure 2(b), and for the weights (0.7,0.3) in Figure 2(c).

In our experiment, we compare our SUD algorithm with two baselines across 10 weighting sets and show the average objective value and variance of each algorithm across 10 weighting sets in the table. It is obvious that the SUD has the best performance compared with the SUC and SUEC in three groups. For example, in Group 1, the average value and variance of the SUD are  $48.88 \pm 8.07$ , while the values for the SUC are  $98.40 \pm 18.18$  and those for the SUEC are  $92.20 \pm 11.72$ . This shows that the SUD algorithm achieves the best result with the lowest objective value and the lowest variance in group 1. Similarly, the SUD algorithm also outperforms the SUC and SUEC algorithms in Groups 2 and 3. The superior performance of the SUD algorithm can be attributed to its strategy of selectively migrating certain services for caching on the server, as opposed to the other two algorithms that migrate all requested services without discrimination. The SUD algorithm selects the most appropriate servers to provide replications for service migration instead of relying solely on the cloud server or randomly selecting edge servers.

In the service replacement stage of service updating, we compare our SRD algorithm with the SR-LRU algorithm and the SR-R algorithm on two scales of datasets, and the results are shown in Figure 3. On each dataset, we compare the performance of three algorithms at the weight settings of  $(\alpha = 0.4, \beta = 0.6)$  and  $(\alpha = 0.6, \beta = 0.4)$ . We can see that SRD demonstrates superior performance compared to the others across both weight settings of  $(\alpha = 0.4, \beta = 0.6)$  and  $(\alpha = 0.6, \beta = 0.4)$  on group 1 as shown in Figure 3(a), as well as on group 2 as shown in Figure 3(b). In group 1, the SR-LRU and the SR-R exhibit unstable performance under

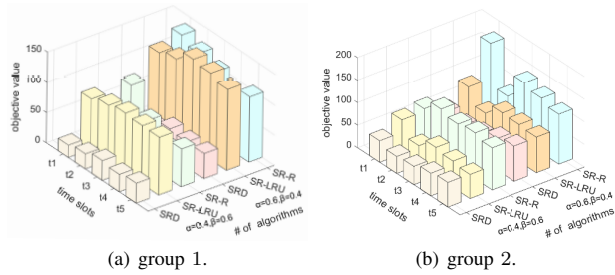


Fig. 3. comparison of replacement algorithms at the service updating stage

the influence of different weight settings, whereas the SRD algorithm consistently achieves the lowest value under both weight settings. The value of SRD is slightly higher values under the weight setting of  $(\alpha = 0.6, \beta = 0.4)$  compared to  $(\alpha = 0.4, \beta = 0.6)$ , which is consistent in group 1 and group 2. The excellence of SRD lies in its consideration of both the service access frequency and the distribution of service replications because when the replications of the same service become too dense, there will be inefficient storage resource usage rendering replication extension unnecessary. In summary, the experimental results effectively demonstrate the effectiveness and superiority of the proposed algorithms in various stages. The USC algorithm performs exceptionally well in the initial service caching stage. The SUD and SRD algorithms exhibit significant advantages in the service updating stage, effectively reducing latency and cost.

## VI. CONCLUSION

This paper focuses on addressing the service caching problem in a dynamic and resource-constrained multi-access edge computing environment. We explore strategies to achieve dynamic service caching and updating in a cost-efficient way while operating under the constraints of limited storage resources and guaranteeing QoS to users. Our objective is to minimize user latency and system costs when dynamically caching and updating services in resource-constrained edge computing environments. We propose innovative strategies at various stages. In the initial service caching stage, we design a service caching algorithm based on dynamic programming to select services to be provisioned at appropriate locations. To cope with the challenges posed by dynamic user behavior, an algorithm based on reinforcement learning is proposed to determine service extension. At the service updating replacement stage, we design a novel replacement algorithm, which considers the distribution of service replications and service access frequency. In the experimental aspect, the effectiveness and performance of the proposed algorithms are validated through extensive experiments. In summary, this paper provides an in-depth theoretical analysis and innovative solutions to the problem of dynamic service caching and updating in edge computing, offering valuable insights for research and applications in related fields.

## ACKNOWLEDGMENT

This work was supported in part by the National Natural Science Foundation of China under Grant 62072016 and the Beijing Natural Science Foundation under Grant 4244074.

## REFERENCES

- [1] M. K. Somesula, S. K. Mothku, S. C. Annadanam. Cooperative service placement and request routing in mobile edge networks for latency-sensitive applications[J]. *IEEE Systems Journal*, 2023, 17(3): 4050-4061.
- [2] S. Ko, S. J. Kim, H. Jung and S. W. Choi. Computation offloading and service caching for mobile edge computing under personalized service preference[J]. *IEEE Transactions on Wireless Communications*, 2022, 21(8): 6568-6583.
- [3] S. Zhong, S. Guo, H. Yu, Q. Wang. Cooperative service caching and computation offloading in multi-access edge computing[J]. *Computer Networks*, 2021, 189: 107916.
- [4] L. Wang, X. Deng, J. Gui, et al. Microservice-oriented service placement for mobile edge computing in sustainable internet of vehicles[J]. *IEEE Transactions on Intelligent Transportation Systems*, 2023, 24(9): 10012-10026.
- [5] X. Chi, H. Chen, G. Li, et al. EDSP-edge: Efficient dynamic edge service entity placement for mobile virtual reality systems[J]. *IEEE Transactions on Wireless Communications*, 2023.
- [6] J. Cheng, D. T. Nguyen, and V. K. Bhargava. Resilient edge service placement under demand and node failure uncertainties[J]. *IEEE Transactions on Network and Service Management*, 2023.
- [7] H. Li, M. Sun, F. Xia, X. Xu and M. Bilal. A Survey of Edge Caching: Key Issues and Challenges[J]. *Tsinghua Science and Technology*, 2023, 29(3): 818-842.
- [8] C. Sun, X. Li, J. Wen, X. Wang, Z. Han and V. C. M. Leung. Federated deep reinforcement learning for recommendation-enabled edge caching in mobile edge-cloud computing networks[J]. *IEEE Journal on Selected Areas in Communications*, 2023, 41(3): 690-705.
- [9] S. Qiu, Q. Fan, X. Li, et al. OA-cache: Oracle approximation-based cache replacement at the network edge[J]. *IEEE Transactions on Network and Service Management*, 2023, 20(3): 3177-3189.
- [10] Y. Liu, C. Yang, X. Chen, et al. Joint hybrid caching and replacement scheme for UAV-assisted vehicular edge computing networks[J]. *IEEE Transactions on Intelligent Vehicles*, 2023.
- [11] P. Li, Y. Zhang, H. Zhang, et al. A delayed eviction caching replacement strategy with unified standard for edge servers[J]. *Computer Networks*, 2023, 230: 109794.
- [12] H. Zhou, K. Jiang, S. He, G. Min and J. Wu. Distributed deep multi-agent reinforcement learning for cooperative edge caching in internet-of-vehicles[J]. *IEEE Transactions on Wireless Communications*, 2023.
- [13] Z. Zhang, Y. Liu, Z. Peng, et al. Digital Twin-Assisted Data-Driven Optimization for Reliable Edge Caching in Wireless Networks[J]. *IEEE Journal on Selected Areas in Communications*, 2024.
- [14] H. Wu, B. Wang, H. Ma, et al. Collaborative caching relay algorithm based on recursive deep reinforcement learning in mobile vehicle edge network[J]. *Ad Hoc Networks*, 2024, 152: 103313.
- [15] Z. Li, C. Yang, X. Huang, W. Zeng and S. Xie. CoOR: collaborative task offloading and service caching replacement for vehicular edge computing networks[J]. *IEEE Transactions on Vehicular Technology*, 2023.
- [16] Z. Wang, J. Hu, G. Min, Z. Zhao, et al. Agile Cache Replacement in Edge Computing via Offline-Online Deep Reinforcement Learning[J]. *IEEE Transactions on Parallel and Distributed Systems*, 2024, 35(4): 663-674.
- [17] S. Anokye, D. Ayepah-Mensah, A. M. Seid, G. O. Boateng and G. Sun. Deep reinforcement learning-based mobility-aware UAV content caching and placement in mobile edge networks. *IEEE Systems Journal* 16.1 (2021): 275-286.
- [18] S. Jiang, J. Wu, F. Zuo, et al. Balance-aware Cost-efficient Routing in the Payment Channel Network[C]. 2023 IEEE/ACIS 21st International Conference on Software Engineering Research, Management and Applications (SERA). IEEE, 2023: 8-15.
- [19] Y. Li, A. Zhou, X. Ma and S. Wang. Profit-aware edge server placement[J]. *IEEE Internet of Things Journal*, 2021, 9(1): 55-67.