

# Enhanced Multi-Stage Optimization of Dynamic QoS-Aware Service Caching and Updating in Mobile Edge Computing

Shuaibing Lu<sup>1</sup>, Member, IEEE, Xin Jin<sup>1</sup>, Jie Wu<sup>2</sup>, Fellow, IEEE, Shuyang Zhou, Jackson Yang, Ran Yan, Haiming Liu<sup>1</sup>, and Zhi Cai<sup>1</sup>, Member, IEEE

**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 investigates optimizing the service caching and updating problem within multi-stage and proposes a novel framework with three proposed strategies for the different stages to jointly optimize the delay and cost. At the initial service caching stage, we propose a basic caching strategy based on dynamic programming for the single-area scenario, taking into account the constraint of limited memory resources. To improve the caching strategy, we extend our consideration to the multiple-area scenario and design an improved algorithm based on tabu search. 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 reinforcement 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, mobile edge computing.

## I. INTRODUCTION

WITH the proliferation of smartphones, IoT devices, and other connected gadgets, the demand for real-time data and services has surged. Applications such as video streaming, online gaming, and IoT services require low latency and high bandwidth, presenting significant challenges to the backbone network and cloud data centers. The traditional

centralized cloud model, with its high latency, can result in increased service response times and may struggle to meet user demands during network congestion. To address these challenges, mobile edge computing (MEC) has emerged as a promising solution, deploying edge servers at the network's edge to bring computation and storage closer to end-users and devices. This architecture enables low-latency and high-bandwidth service delivery.

However, as the volume of services and users continues to grow and user behaviors become increasingly dynamic, managing service caching and updating on edge servers has become a significant challenge. Traditional caching strategies often result in resource inefficiency, while rigid updating mechanisms may fail to meet the Quality of Service (QoS) requirements of users. Thus, there is an urgent need to develop a dynamic, QoS-aware approach to service caching and updating that ensures both user satisfaction and cost efficiency. The limited storage resources available on edge servers, coupled with the dynamic nature of user behaviors, complicate service deployment. Specifically, three key challenges emerge: (i) How to select services for caching and determine their optimal locations, given resource constraints, while ensuring QoS by avoiding latency from long-distance transmission across multiple regions? (ii) How to balance performance under the dynamic behavior of multiple users with the expansion of service replications, while mitigating the associated cost increase? (iii) How to determine the appropriate replacement services at resource-constrained edge networks to optimize the trade-off between cost and latency?

### A. Motivation and Challenge

To balance the interests of operators and customers across widely dispersed networks and achieve maximum efficiency, dynamic service caching and updating involve determining which services should be provisioned on specific servers within the edge network. Figure 1 illustrates the motivation and challenges of this problem in the context of QoS guarantees. At the top of the figure, a cloud environment hosts five services, denoted as  $s_1$  to  $s_5$ , represented by cylinders of different colors. These services are pre-provisioned at the cloud data center by the operator. Each service is associated with specific user requirements, and users request these services from the edge servers. The figure also shows four

Received 12 September 2024; revised 5 March 2025; accepted 11 May 2025. Date of publication 15 May 2025; date of current version 7 August 2025. The associate editor coordinating the review of this article and approving it for publication was J. J. Yang. (Corresponding author: Haiming Liu.)

Shuaibing Lu, Xin Jin, Ran Yan, and Zhi Cai are with the College of Computer Science, Beijing University of Technology, Beijing 100124, China (e-mail: lushuaibing@bjut.edu.cn).

Jie Wu is with the Department of Computer and Information Sciences, Temple University, Philadelphia, PA 19122 USA (e-mail: jiewu@temple.edu).

Shuyang Zhou, Jackson Yang, and Haiming Liu are with the School of Software Engineering, Beijing Jiaotong University, Beijing 100091, China (e-mail: liuhaiming@bjtu.edu.cn).

Digital Object Identifier 10.1109/TNSM.2025.3570427

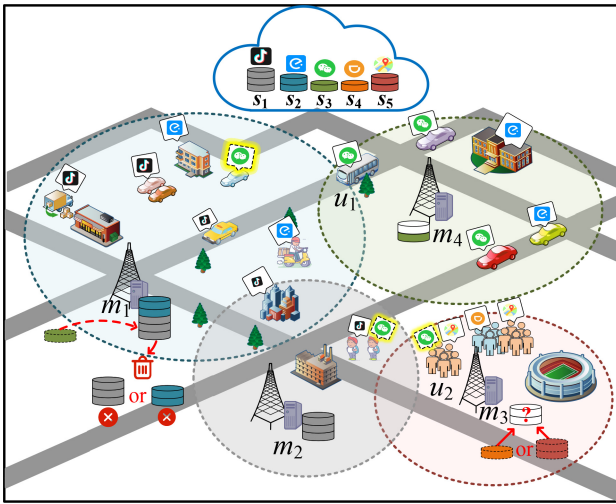


Fig. 1. Example of service caching and updating strategies in the mobile edge computing network.

edge servers, labeled  $m_1$ ,  $m_2$ ,  $m_3$ , and  $m_4$ , each serving distinct areas of the network. These servers cache and provide services to users within their respective areas, with the goal of minimizing latency and enhancing content delivery. Users are represented by icons, with each icon corresponding to a service request. The icons are color-coded to match the service they are requesting, for example, users requesting service  $s_1$  are depicted with icons of the same color as the  $s_1$  cylinder, and similarly for the other services. The icons enclosed by solid lines represent users currently requesting services in the area during the current time slot, while icons enclosed by dashed lines represent new users expected to request services in the area during the upcoming time slot.

1) : In the initial operation phase of the system, all edge servers begin with empty caches. The initial caching placement is particularly crucial, as it directly impacts the system's ability to adapt to dynamic user behavior in the future. A suboptimal initial placement can lead to inefficient resource utilization, resulting in costly cache replacements. The primary challenge is to determine which services should be cached on each edge server to minimize user latency and enhance system performance, considering the varying storage capacities and computational resources of the edge servers. Caching decisions must carefully balance the trade-off between minimizing latency and avoiding high storage costs due to the limited capacity of the edge servers. For example, in the geographic area covered by  $m_3$ , two types of services,  $s_4$  and  $s_5$ , are required, each with different storage costs and sizes. However, due to the limited storage capacity of  $m_3$ , it is not feasible to cache both services. One option is to cache  $s_4$ , which requires less capacity and incurs lower costs, but serves fewer users and does not significantly reduce latency. Alternatively, caching  $s_5$  would serve more users and reduce latency, but comes with a higher storage and cost burden. Therefore, it is essential to develop a strategy that effectively balances the trade-off between cost and latency under constrained storage resources.

2) : As user behavior evolves over time, the distribution of service requests changes accordingly. This dynamic nature

of user demand requires the system to adapt its cache by updating replicas. The challenge lies in expanding service replicas for newly requested services while managing the trade-off between latency reduction and increased system costs. This decision is non-trivial, as increasing the number of service replicas can reduce latency but also introduce higher storage and migration costs. Additionally, dynamic adaptation to changing user mobility and request distribution patterns is crucial. Balancing the trade-off between performance under dynamic user behavior and the costs associated with service replication is a complex task.

Consider service  $s_3$  from Figure 1 as an example. Suppose user  $u_1$ , who is requesting service  $s_3$  on edge server  $m_4$ , switches to  $m_1$ , while new requests for  $s_3$  are made on servers  $m_1$ ,  $m_2$ , and  $m_3$  at time slot  $t + 1$ . Provisioning  $s_3$  on both  $m_1$  and  $m_2$  minimizes latency but incurs extra storage costs. Alternatively, accessing  $s_3$  from the cloud reduces costs but increases latency, making neither solution entirely satisfactory. A compromise would be to provision the replication of  $s_3$  on either  $m_1$  or  $m_2$ . However, determining the optimal location for the replica is non-trivial. For instance, placing the replication of  $s_3$  on  $m_1$  would optimize latency due to the higher number of users requesting  $s_3$  on  $m_1$ . However, for user  $u_2$ , accessing  $s_3$  from  $m_1$  would require a detour via  $m_2$ , leading to significant communication latency when dealing with large data volumes. Conversely, placing the replication of  $s_3$  on  $m_2$  would optimize latency and avoid extreme communication delays due to its complete connectivity. However, because there are fewer user requests for  $s_3$  on  $m_2$ , accessing the service via neighboring servers would be suboptimal for many users. Therefore, selecting the appropriate locations for caching service replicas, considering the distribution of user requests and optimizing both cost and latency, is a non-trivial problem.

3) : When certain services experience decreased demand or when edge servers run out of storage capacity, some cached services must be replaced. The challenge lies in selecting the optimal services for replacement to minimize both latency and cost. Replacing inappropriate services can increase latency and may lead to frequent replacements of the same services, resulting in high overhead costs. The replacement process must ensure that the system operates efficiently without causing excessive delays or incurring unnecessary costs. Consider the geographic area covered by  $m_1$  as an example. Suppose a new replication of service  $s_3$  is to be added to edge server  $m_1$ , but the remaining capacity cannot accommodate  $s_3$ . In this case, a service must be replaced. One potential solution is to replace  $s_1$ , which is significantly larger than  $s_3$ . Replacing  $s_1$  would free up more capacity for other services, and users currently served by  $s_1$  could request services from  $m_2$  without a significant increase in latency. However, migration costs become significant if  $s_1$  is frequently updated and replaced due to its large size. Another solution is to replace  $s_2$ , which serves fewer users and incurs lower migration costs due to its smaller size. However, since  $m_1$  is the only server that caches  $s_2$  based on the distribution, replacing  $s_2$  would result in high latency, as all users requesting  $s_2$  would need to access it through the cloud. Therefore, determining how to appropriately replace

services, considering both the total cost and latency, is a non-trivial problem.

### B. Contributions and Paper Organization

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 four 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 and tabu search. 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 reinforcement 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

In recent years, the promising technology of edge computing has shifted computations from centralized clouds to distributed edges closer to the user. Due to dynamic user behavior and the limited and heterogeneous computing and storage capacities of edge devices, caching and updating services are necessary to guarantee user QoS and optimize costs. Various studies have addressed this issue from different perspectives. Some focus on cost reduction. Xu et al. [1] developed a distributed service caching method to minimize the social cost of all providers by introducing a novel cost-sharing model and coalition formation game. Chen et al. [2] proposed an online market mechanism to achieve cost efficiency in edge demand response programs. Some studies focus on reducing latency. Ko et al. [3] decoupled computation offloading and

service caching with limited resources and latency, taking user preferences into account. Li et al. [4] proposed a two-stage caching approach aimed at reducing delays for mobile video-on-demand users. Jin et al. [5] focused on jointly optimizing the resource utilization of edge servers and physical links under latency constraints. Some studies address the service caching problem by jointly optimizing it with other related problems. Gao et al. [6] optimized access network selection and service placement for MEC to improve QoS in a cost-efficient manner by balancing access delay, communication delay, and service switching cost. Bi et al. [7] and Zhou et al. [8] considered the joint optimization of service caching and computational offloading as a mixed-integer non-linear programming problem. Zhao et al. [9] and Salaht et al. [10] designed efficient convex programming-based algorithms to solve the problem of offloading dependent tasks with service caching to minimize makespan. Sun et al. [11] and Zhou et al. [12] proposed a recommendation-enabled edge caching system. Chen et al. [13] investigated task allocation in Multi-Task Transfer Learning (MTL) in edge computing environments. Chen et al. [14] proposed an admission control mechanism for time-sensitive edge services. Pan et al. [15] optimized container caching jointly with request distribution. Some studies employed deep learning and reinforcement learning algorithms for caching strategies. Wu et al. [16], Wu et al. [17], Pei et al. [18], and Sun et al. [19] made inferences using DNN models. Lei et al. [20] and Wang et al. [21] trained deep neural networks to predict the behavior of terminal devices. Qu et al. [22] formulated a parameter-sharing model placement problem to maximize the cache hit ratio in multi-edge wireless networks by balancing the tradeoff between storage efficiency and service latency. Some studies, such as [23], on dynamic service caching, often assume relatively stable user request distributions or model changes within a predefined framework. However, as user behavior and request patterns in edge computing environments can evolve rapidly, many of these methods fail to account for such dynamic fluctuations. This creates a gap in addressing the complexity of real-world edge computing applications, where service requests can vary significantly over time due to factors like user mobility and changing content demand. In contrast, our work introduces a dynamic approach that considers user behavior and request distribution changes over time. We propose caching strategies that can dynamically adjust based on real-time user demands, which is critical for large-scale edge computing systems.

To address the challenge of dynamic user behavior, some existing works on service updating have been proposed, which focus not only on the hit ratio but also on service updating. Huang et al. [24] proposed a refined segment-based cache update strategy to improve system throughput and segment hit ratio. Li et al. [25] proposed a priority- and LRU-based cache replacement strategy that selects cache files by size, heat, and user access characteristics, dynamically increasing replicas as needed. Ugwuanyi et al. [26] proposed a modified least frequently used cache replacement method that considers not only the frequency of data in the cache but also the frequency of newly received requests. Zhou et al. [27] modeled the

cache replacement process as an MDP and used a distributed edge caching method for intelligent caching. Yuan et al. [28] focused on the number of file copies and designed two greedy algorithms for caching and replacing content. Wei et al. [29] defined the cache value based on the priority of the computational task and proposed a cache replacement strategy using the ideal point method. Li et al. [30], Anokye et al. [31], and Wang et al. [32] proposed solutions based on deep reinforcement learning. Pham et al. [33] proposed an efficient strategy for selecting caching services and scheduling tasks in edge computing networks with multiple access points. Kazari et al. [34] proposed a cache update policy considering dynamic popularity for file requests. Yu et al. [35] optimized caching decisions based on predicting vehicle driving trajectories and travel preferences. Li et al. [36] proposed two modules, Delayed-Eviction and Unified-Standard, for existing caching replacement strategies to improve caching performance. Zhang et al. [37] transformed a traditional file caching algorithm to handle delayed hits.

Existing methods typically treat service caching and updating as a single-stage optimization task. While effective in simpler, single-phase systems, these methods struggle to handle the complexity of multi-stage caching management. Our work introduces a phased strategy that divides the caching process into three stages. This approach allows us to target specific problems at each stage and apply appropriate optimization techniques, thereby improving overall performance. In this paper, we investigate strategies for caching and updating services, considering service replication distribution and the frequency of service access under limited resources, with the goal 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 consisting of a cloud data center, edge servers, and mobile end users. We assume that the services required by the users have been pre-provisioned on the cloud data center by the operators, represented as the set  $\mathbf{S} = s_1, s_2, \dots, s_j$ . For each service requested by users, we record the information in a tuple 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. To scale the service areas, we assume that each service can have multiple replications on several edge servers. Let  $N(s_j)$  represent the total number of replications, where  $N(s_j) = \sum_{\forall m_k \in \mathbf{M}} x(m_k, s_j)$ . Here,  $x(m_k, s_j) \in 0, 1$  denotes whether service  $s_j$  is replicated on server  $m_k$ . The caching cost is  $\varphi_{s_j}$  for each replication of service  $s_j$ . We use  $\mathbf{c}$  to denote the cloud data center, with its computation resource given by  $R_c^{cal}$ . In the edge layer, the set of edge servers supported by the operators is denoted as  $\mathbf{M} = m_1, m_2, \dots, m_k$ . The storage resource of edge server  $m_k$  is represented by  $R_{m_k}^s$ . In the user layer, we denote the set of users as  $\mathbf{U} = u_1, u_2, \dots, u_i$ . Each user requires a service, and for each user  $u_i$ , we use a triplet  $u_i = (q_{u_i}, z_{u_i}^{cal}, \kappa_{u_i})$  to capture the user's information. Here,  $q_{u_i}$  represents the service requested by user  $u_i$ ,  $z_{u_i}^{cal}$  is the amount of task data, and  $\kappa_{u_i}$  denotes the server on which user

TABLE I  
SUMMARY OF MAIN NOTATIONS IN THE SYSTEM MODEL

Notation	Definition
$\mathbf{c}$	The cloud server, where $\mathbf{c} = (R_c^{cal})$ .
$\mathbf{S}$	Set of services, where $\mathbf{S} = \{s_j\}$ .
$s_j$	The $j$ -th service, where $s_j = (c_{s_j}, \varphi_{s_j})$ .
$\mathbf{M}$	Set of edge servers, where $\mathbf{M} = \{m_k\}$ .
$m_k$	The $k$ -th edge server, where $m_k = \{R_{m_k}^s, R_{m_k}^{cal}\}$ .
$\mathbf{U}$	Set of users, where $\mathbf{U} = \{u_i\}$ .
$u_i$	The $i$ -th user, where $u_i = \{q_{u_i}, z_{u_i}^{cal}, \kappa_{u_i}\}$ .
$z_{u_i}^{cal}$	The amount of task data of $u_i$ .
$d_{u_i}^{cal}$	The latency to calculate the task from $u_i$ .
$d_{u_i}^{comm}$	The latency of $u_i$ to communicate to other servers.
$l(m_k, m_{k'})$	The network propagation distance between $m_k$ and $m_{k'}$ .
$\lambda(m_k, m_{k'})$	The transmission speed between $m_k$ to $m_{k'}$ .
$l_c$	The network propagation distance from users to cloud server.
$b_c$	The bandwidth between users and cloud server.
$\eta$	The efficiency about $b_c$ .
$C^{stor}$	The total cost to store services.
$C^{migr}$	The total cost to migrate service replications.

$u_i$  is located. The symbols and definitions used in this paper are summarized in Table I.

#### B. QoS Model

We present a Quality of Service (QoS) model for users, which consists 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, and some services are selectively provisioned on edge servers. When users request services that are available both in the cloud and on edge servers, we prefer to use the services from edge servers to process the tasks.

We define  $\mathbf{D}$  as the total delay, which is the sum of two components: the computational delay  $d_{u_i}^{cal}$  and the communication delay  $d_{u_i}^{comm}$ . The total delay can be calculated as  $\mathbf{D} = \sum_{u_i \in \mathbf{U}} (d_{u_i}^{cal} + d_{u_i}^{comm})$ . For the computational delay,  $d_{u_i}^{cal}$ , there may be different delays based on the location of the service. If the service that user  $u_i$  is accessing is deployed on edge server  $m_k$ , the computation delay on the edge server is given by  $d_{u_i}^{cal(m_k)} = z_{u_i}^{cal} / R_{m_k}^{cal}$ , where  $z_{u_i}^{cal}$  is the amount of task data for user  $u_i$ , and  $R_{m_k}^{cal}$  represents the computation power of edge server  $m_k$ . However, if the service is not deployed on the edge server, 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 for  $u_i$  is:  $d_{u_i}^{cal} = d_{u_i}^{cal(m_k)} \cdot x(m_k, s_j) + d_{u_i}^{cal(c)} \cdot (1 - x(m_k, s_j))$ , 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 communication delay  $d_{u_i}^{comm}$  can be determined similarly. If the service is located on an edge server, the communication delay from user  $u_i$  to edge server  $m_k$  is:  $d_{u_i}^{comm(m_k)} = z_{u_i}^{cal} \cdot l(m_k, m_{k'}) / \lambda(m_k, m_{k'})$ , where  $l(m_k, m_{k'})$  is the network propagation distance and  $\lambda(m_k, m_{k'})$  is the

transmission speed between the edge server located near the user and the edge server providing the service. If the service is accessed from the cloud, the communication delay is given by:  $d_{u_i}^{comm(c)} = z_{u_i}^{cal} \cdot l_c / \eta \cdot b_c$ , where  $b_c$  is the bandwidth,  $l_c$  is the network propagation distance from the user to the cloud, and  $\eta$  is the efficiency related to  $b_c$ . Thus, the communication delay for  $u_i$  is:  $d_{u_i}^{comm} = d_{u_i}^{comm(m_k)} \cdot x(m_k, s_j) + d_{u_i}^{comm(c)} \cdot (1 - x(m_k, s_j))$ , where  $\forall m_k \in \mathbf{M}$ .

### C. Cost Model

The costs involved in this work primarily include the storage costs of services,  $C^{stor}$ , incurred by caching services on edge servers, and the migration costs,  $C^{migr}$ , which are incurred when migrating service applications during service updates. 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 denoted by  $\varphi_{s_j}$ , and the total number of replications of  $s_j$  in the system is  $N_{s_j}$ . Therefore, 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, the user can request the migration of the service from a nearby node where it is already deployed. The migration cost of  $s_j$  from edge server  $m_{k'}$  to  $m_k$  can be calculated as:  $C_{s_j}^{migr(m_k, m_{k'})} = \gamma_m \cdot c_{s_j} \cdot l_{(m_k, m_{k'})}$ . If the destination server is the cloud, the migration cost is given by:  $C_{s_j}^{migr(m_k, c)} = \gamma_c \cdot c_{s_j} \cdot l_c$ , where  $c_{s_j}$  is the storage capacity occupied by  $s_j$ , and  $l_c$  is the network propagation distance between the cloud and  $m_k$ . The coefficients  $\gamma_m$  and  $\gamma_c$  represent the factors associated with migration 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:

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

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

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

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  $\mathbf{P}_2$ :

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

s.t. (2)(3)

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

## IV. ALGORITHM DESIGN

In this section, we present three main strategies to solve the dynamic caching and service update problems in a cost-effective manner. These strategies are designed to align with the service placement process and user behavior patterns, and the entire process is divided into three stages: initial service caching, service update decision-making, and service update replacement, with the goal of maintaining an optimal objective value at each stage. During the initial service caching stage, a strategy must be developed to enable edge servers to cache the most appropriate services, thereby achieving a favorable objective value. As the process advances to the service update decision-making stage, caching extension decisions are made based on the dynamic changes in user requests and the service caching status from the previous stage. In the service update replacement stage, decisions are made regarding which services to remove from the cache and which services to extend. The quality of the caching results from the initial service caching stage has a direct impact on the outcomes of the subsequent stages. To address the specific challenges at each stage, strategies have been developed to enhance the overall objective value. First, an initial service caching strategy is introduced, which considers the distribution of user requests using a dynamic programming approach. Additionally, user mobility is accounted for by implementing an innovative selection strategy to determine the inclusion of service replications and designate the servers responsible for hosting these replications. Finally, a caching replacement algorithm is formulated to manage situations where the available capacity is insufficient to accommodate the designated services for caching.

### A. Initial Service Caching

The initial service caching problem is aimed at selecting appropriate services to cache on each edge server to minimize latency and cost, particularly given the constraints of limited storage resources. We address the problem from two perspectives: the single-area scenario and the multiple-area scenario.

1) *Single-Area Scenario*: In the single-area scenario, we transform the initial service caching problem into a knapsack problem by introducing the concept of service value. This approach involves selecting services of varying sizes to maximize the total value that an edge server can offer. 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:        Backtrack  $V$  to obtain the optimal scheme  $I(m_k)$ ;
13:         $\mathbf{I} = \mathbf{I} \cup I(m_k)$ ;
14:   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. Using dynamic programming, we decompose the target problem into subproblems that find a caching strategy to maximize the value obtained within the constraints of limited capacities.

In this subsection, we introduce a novel algorithm to address the initial service caching problem, named USC (User-oriented Service Caching), designed for multiple users within an edge network. The goal of the algorithm is to determine an optimal caching scheme for each edge server using a dynamic programming approach. The detailed steps of the algorithm are outlined in Algorithm 1. To begin, several preparatory elements are required for the algorithm, including a set of edge servers denoted by  $\mathbf{M}$  and information about the requested services represented by  $\mathbf{S}$ . We also require information concerning the set of users  $\mathbf{U}$  requesting services that are distributed in each area, including details such as task sizes, 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. The strategy stores the service placement information for each server, recording which service is placed on which server. 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 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

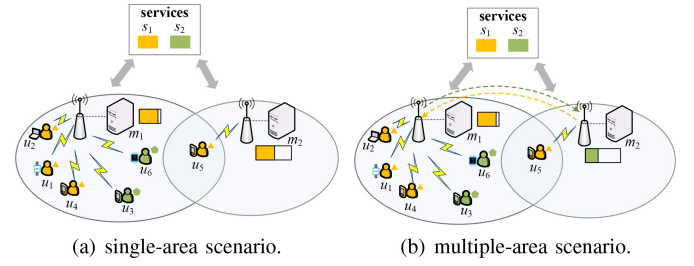


Fig. 2. Examples of the influence of the regions on the results.

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 caching it. 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, and we return the initial service caching strategy in line 13.

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  $|\mathbf{M}|$ . On this basis, we consider that all services can be placed on each server, representing  $|\mathbf{S}|$  for the second loop. For each server, we use  $\hat{R} = \max_{m_k \in \mathbf{M}} R_{m_k}^s$  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  $|\mathbf{S}|$ . Therefore, the time complexity of the algorithm is  $O(|\mathbf{M}| \times |\mathbf{S}| \times (\hat{R} + 1))$ .

2) *Multiple-Area Scenario:* The solution derived from the USC algorithm in a single-area scenario has certain limitations. For example, consider the simple scenario depicted in Figure 2, where there is a significant difference in user requests between the areas covered by  $m_1$  and  $m_2$ , as well as in their storage capacities. Figure 2(a) shows the service caching scheme that considers only the request situation in a single area. Although  $m_1$  receives numerous requests, its storage capacity is limited to caching only one service, either  $s_1$  or  $s_2$ . Since the value of service  $s_1$  is higher than that of  $s_2$ , the scheme for  $m_1$  caches  $s_1$ . Despite the large storage capacity of  $m_2$ , there is only one request for the  $s_1$  service, so  $m_2$  also caches  $s_1$ . In this scenario, users in the area covered by  $m_1$  who request  $s_2$  can only access it through the cloud, leading to increased service delays and unnecessary overhead due to the redundant placement of  $s_1$ . This demonstrates that a caching scheme focused solely on a single area is not ideal, as it results in excessive service replication and fails to effectively utilize the less busy, large-capacity edge servers. To improve

**Algorithm 2** USC-TS Algorithm**Require:**  $\mathbf{I}$ .**Ensure:** improved service caching strategy  $\mathbf{I}^*$ .

---

```

1: Set  $\mathbb{I} \leftarrow \mathbf{I}$ ;
2: Set  $\mathbf{I}^* \leftarrow \mathbb{I}$  and calculate the objective value  $\mathbf{O}^*$ ;
3: Set the maximum length of the tabu list to be  $\mathbb{L}$ ;
4: Set the number of solutions in set  $\mathbb{N}$  to be  $\mathfrak{N}$ ;
5: Set the maximum number of iterations  $\Gamma$ ;
6: Initialize  $\mathbf{TL} = \emptyset$ ;
7: while  $h \leq \Gamma$  do
8:   Generate  $\mathfrak{N}$  neighbor solutions based on  $\mathbb{I}$ ;
9:   Update set  $\mathbb{N}$ ;
10:  Choose the solution with  $\mathbb{I}' = \arg \min\{\mathbf{O}(\mathbb{I}') | \mathbb{I}' \in \mathbb{N}\}$ ;
11:  if  $\mathbb{I}'$  not in  $\mathbf{TL}$  or  $\mathbf{O}(\mathbb{I}') < \mathbf{O}^*$  then
12:    Update  $\mathbb{I} \leftarrow \mathbb{I}'$ ;
13:    if the length  $|\mathbf{TL}| \geq \mathbb{L}$  then
14:      Remove the oldest solution in  $\mathbf{TL}$ ;
15:    Add  $\mathbb{I}$  to  $\mathbf{TL}$ ;
16:    if  $\mathbf{O}(\mathbb{I}') < \mathbf{O}^*$  then
17:      Update  $\mathbf{I}^* \leftarrow \mathbb{I}'$ ;
18:      Update  $\mathbf{O}^* \leftarrow \mathbf{O}(\mathbb{I}')$ ;
19:  Update  $h \leftarrow h + 1$ ;
20: Return  $\mathbf{I}^*$ .

```

---

the original service caching algorithm, we extend our analysis by considering the distribution of user requests in these two areas,  $m_1$  and  $m_2$ , and allocate the resources in such a way that the quality of service and overhead requirements are met simultaneously. Figure 2(b) illustrates a caching scheme for the multi-area scenario, which efficiently leverages adjacent edge servers. To address the above problem, we propose a more efficient caching scheme by aggregating server resources and user request data within a multi-area scenario. According to the feasible solution obtained by the USC algorithm as the initial solution, we introduce an improved strategy based on tabu search to explore the approximate optimal solution called USC-TS to avoid the redundant overhead caused by the lack of timely exchange of regional information.

The main idea of our scheme is to start with an initial feasible solution and then explore a series of specific search directions, selecting the direction that yields the most significant improvement in a specific objective function value. The specific steps are outlined in Algorithm 2. We use the caching strategy  $\mathbf{I}$  obtained from the USC algorithm as the initial feasible solution and aim to derive the best possible solution,  $\mathbf{I}^*$ , through our algorithm. Additionally, we calculate the corresponding best objective value  $f(\mathbf{I}^*)$ .

To further refine the search process, we determine the number of solutions to explore by setting the number of neighbor solutions as  $\mathfrak{N}$  in line 4 of the algorithm. To discourage the search from coming back to previously-visited solutions, the tabu list is introduced, and we set the maximum length of the tabu list  $\mathbf{TL}$  to be  $\mathbb{L}$  and initialize the list as empty in line 3. Lines 7 to 18 demonstrate how the algorithm iteratively searches for the approximate optimal solution. We use  $\Gamma$  to represent the maximum number of iterations. Based on the

current solution  $\mathbb{I}$ , we generate a group of neighbor solutions and update set  $\mathbb{N}$ . Then, we select the best solution with the minimum value of  $\mathbf{O}(\mathbb{I}')$  where  $\mathbb{I}' = \arg \min\{\mathbf{O}(\mathbb{I}') | \mathbb{I}' \in \mathbb{N}\}$  in line 10. The selection of the solution must fulfill the following conditions, either  $\mathbb{I}'$  does not exist in the tabu list or the tabu tenure condition is fulfilled, whereby the objective value of the chosen solution is less than  $\mathbf{O}^*$ . Otherwise, the neighbor solutions must be recreated, i.e., go back to line 8. We then update the current solution  $\mathbb{I}$  with the selected one  $\mathbb{I}'$  and add it to the tabu list  $\mathbf{TL}$  in lines 12 to 15. If the tabu list is full, i.e., the length of the tabu list exceeds the limit  $|\mathbf{TL}| \geq \mathbb{L}$ , we need to remove the oldest solution in  $\mathbf{TL}$ . Subsequently, we update the best solution  $\mathbf{I}^*$  and  $\mathbf{O}^*$  if the objective value of the selected solution  $\mathbf{O}(\mathbb{I}')$  is lower than  $\mathbf{O}^*$ . At the end of the iterations, we obtain the final best solution,  $\mathbf{I}^*$ , representing the improved service caching strategy obtained by the algorithm.

The time complexity of Algorithm 2 is  $O(\Gamma \times \mathfrak{N} \times \mathbb{L})$ . The initialization operations can be regarded as constant in time and therefore negligible. The time complexity of generating neighbor solutions is proportional to the number of solutions in the set, which is  $O(\mathfrak{N})$ . Selecting the best solution from a set of neighbor solutions requires traversing the neighbor solutions, resulting in a time complexity of  $O(\mathfrak{N})$ . Checking whether the selected solution exists in the tabu list can be performed up to  $\mathbb{L}$  times per iteration, so the time complexity of this operation is  $O(\mathbb{L})$ . Given that the entire tabu search process iterates  $\Gamma$  times, the overall time complexity of the algorithm is  $O(\Gamma \times \mathfrak{N} \times \mathbb{L})$ .

### B. Service Updating Decision-Making

In this subsection, we investigate the problem of the service update decision-making stage considering the dynamics of user requests and positions.

1) *MDP*: Due to the dynamic nature of user behavior, new service requests frequently arise within the area. To ensure Quality of Service (QoS) for users, we propose caching new services on edge servers by migrating service replications from other servers, while also considering associated costs. We model this scenario using a Markov Decision Process (MDP). The state space  $\mathbf{Z}$  defines all possible states of service selection, and we represent the state at step  $\tau$  as  $z_\tau = [s_1, s_2, \dots, s_j]\tau$ , where the selected services are denoted by the vector. The action space  $\mathbf{A}$  defines all possible states of server selection, with  $a_\tau = [\mathbf{M}(s_1), \mathbf{M}(s_2), \dots, \mathbf{M}(s_j)]$  representing the action taken at step  $\tau$ . This action consists of selecting servers from replication groups for each service. The function  $\mathbf{M}(s_j) = [\zeta(m_k)] | m_k \in \mathbf{M}$  specifies the set of all servers that host the service  $s_j$ , where  $\zeta(m_k)$  is an indicator function:  $\zeta(m_k) = 1$  if the replication of service  $s_j$  is placed on server  $m_k$ , and  $\zeta(m_k) = 0$  if it is not. We use  $P$  to denote the probability of users reaching other servers. The reciprocal of the total value provided by the service to users is considered the reward,  $r_\tau$ , where  $r_\tau = 1/(\alpha\mathbf{D} + \beta\mathbf{C})$ . The decay factor  $\gamma$  is defined in the range  $[0, 1]$ .

The long-term reward  $G_\tau$  associated with the current state is determined by the Markov chain, with different

chains corresponding to different long-term rewards. The long-term reward is given by:  $G_\tau = r_{\tau+1} + \gamma r_{\tau+2} + \dots = \sum_{k=0}^{\infty} \gamma^k r_{\tau+k+1}$ . Consequently, the value function  $V(z_\tau)$  represents the expected long-term reward for the current state, which is the expectation of all state transition chains starting from  $z_\tau$ :  $V(z_\tau) = E[G_\tau | z=z_\tau]$ . According to the definition of the value function, we can derive the Bellman expectation equation:

$$\begin{aligned} V(z_\tau) &= E[r_{\tau+1} + \gamma V(z_{\tau+1} | z=z_\tau)] \\ &= r(z_\tau) + \gamma \sum_{z_{\tau+1} \in \mathbf{Z}} P_{z_\tau \rightarrow z_{\tau+1}} V(z_{\tau+1}) \\ &= \sum_{a \in \mathbf{A}} \pi(a|z) \left[ r_{S_\tau}^a + \gamma \sum_{z_{\tau+1} \in \mathbf{Z}} P_{z_\tau \rightarrow z_{\tau+1}} V(z_{\tau+1}) \right], \end{aligned} \quad (5)$$

where  $r_{z_\tau} = E(r_{\tau+1})$ ,  $\sum_{z_{\tau+1} \in \mathbf{Z}} P_{z_\tau \rightarrow z_{\tau+1}} V(z_{\tau+1}) = E(V(z_{\tau+1} | z=z_\tau))$ , and  $\pi(a|z) = P[a = a_\tau | z = z_\tau]$ .

The process of solving is to continuously search for the best  $\pi$ , eventually forming a deterministic policy. The state value function is  $v_\pi(z) = E_\pi[G_\tau | z=z_\tau] = E_\pi[r_{\tau+1} + \gamma v_\pi(z_{\tau+1}) | z=z_\tau]$ . The action value function is  $q_\pi(z, a) = E_\pi[G_t | z=z_\tau, a=a_\tau] = E_\pi[r_{\tau+1} + \gamma q_\pi(z_{\tau+1}, a_{\tau+1} | z=z_\tau, a=a_\tau)]$ . The value of a particular state can be represented by the values of all actions in that state, namely  $v_\pi(z) = \sum_{a \in \mathbf{A}} \pi(a|z) \cdot q_\pi(z, a)$ . Similarly, the value of a particular action can be represented by the values of the successor states of that state, that is  $q_\pi(z, a) = r_z^a + \gamma \sum_{z_{\tau+1} \in \mathbf{Z}} P_{z_\tau \rightarrow z_{\tau+1}}^a \cdot v_\pi(z_{\tau+1})$ . We can conclude that there exists an optimal policy  $\pi^*$ , such that  $\pi^* \geq \forall \pi$ , and all optimal policies can achieve the optimal state value function as well as the optimal action-value function. We can use the Q-learning approach to solve the Bellman optimality equation for the action value function, thereby obtaining the optimal policy, where the Bellman optimality equation for the action value function is  $q^*(z, a) = r_z^a + \gamma \sum_{z_{\tau+1} \in \mathbf{Z}} p_{z_\tau \rightarrow z_{\tau+1}}^a v^*(z_{\tau+1})$ . The Bellman optimality equation for the state value function is  $v^*(z) = \max_a (q^*(z, a))$ , and Substituting  $v^*(z)$  into the equation gives  $q^*(z, a) = r_z^a + \gamma \sum_{z_{\tau+1} \in \mathbf{Z}} p_{z_\tau \rightarrow z_{\tau+1}}^a \max_{a_{\tau+1}} (q^*(z_{\tau+1}, a_{\tau+1}))$ .

2) *Decision-Making*: To obtain an elegant solution, we approach the problem in two steps. First, we establish an appropriate threshold to preliminarily screen the services. Second, we apply reinforcement learning to further refine the service selection and determine the most suitable edge server for providing replication for the selected services. To simplify the subsequent selection process, we design a straightforward algorithm called PSS (Primary Services Selection) for the preliminary selection. The specific steps are outlined in Algorithm 3. For each service, we define a replication group  $\mathbf{f}(s_j)$ , which consists of edge servers that have cached service  $s_j$  at the current time, as shown in line 2. We introduce latency response gain to assess the benefit of adding a replication for each service. In line 3, we calculate  $\Delta_{m_k}^{s_j}$  for each service.

*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

---

### Algorithm 3 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  $\mathbf{US}_{m_k}, \mathbf{F}(\mathbf{US}_{m_k})$ .

```

---

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 a randomly selected edge server with a replication of  $s_j$  close to  $m_k$ . Alternatively, when no replications of  $s_j$  are nearby,  $T_{m_k, s_j} = d_{u_i}^c(s_j)$  represents the delay in retrieving  $s_j$  from the cloud.

Based on this, the request to extend the service replication will only be considered if the gain  $\Delta_{m_k}^{s_j}$  is positive. We define the benefit-cost ratio and establish an appropriate threshold  $H_0$  for primary selection. Services with a benefit-cost ratio  $H$  higher than  $H_0$  can proceed to further selection. We use  $\mathbf{US}_{m_k}$  to denote the set of services that meet this criterion.

*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  $H_{m_k}^{s_j}$  for services that satisfy  $\Delta_{m_k}^{s_j} > 0$ , and select those services for which  $H_{m_k}^{s_j} > H_0$  to be included in  $\mathbf{US}_{m_k}$ . In lines 8 to 9, we obtain the replication group set  $\mathbf{F}(\mathbf{US}_{m_k})$  corresponding to the services in  $\mathbf{US}_{m_k}$ .

For further selection, we propose a decision-making strategy called SUD (Service Updating Decision-making based on QoS) for selecting services and edge servers. We select services from  $\mathbf{US}_{m_k}$  to cache and choose an edge server for each service from  $\mathbf{F}(\mathbf{US}_{m_k})$  using reinforcement learning. The specific steps are outlined in Algorithm 4. In lines 1 to 8, we construct the Q-table, which includes both the state space and action space. In line 9, we initialize the values of all entries in the Q-table to zero.

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

In this approach, the entries in the Q-table represent the expected rewards associated with taking specific actions in particular states. For each step, we select which servers will provide replications for the services by choosing the action with the highest value in the current state or, with a certain probability, randomly selecting an action, as described in lines

**Algorithm 4** 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**  $\mathbf{US}'_{m_k}, \mathbf{F}'(\mathbf{US}_{m_k})$

---

12 to 15. After selecting an action, we calculate the next state and the reward for taking the action at the current state. The Q-table value is then updated, along with the state, as shown in lines 16 to 18. Through iterative training, we obtain the updated Q-table, which stores the values of various service and server selections. Finally, we obtain the decision  $\mathbf{US}'_{m_k}$  and  $\mathbf{F}'(\mathbf{US}_{m_k})$  regarding which services should have replications added and which servers should provide the replications by selecting the state and action corresponding to the maximum value in line 19.

The time complexity of Algorithm 4 can be analyzed as follows. In lines 1–3, the creation of the state space requires  $O(|\mathbf{US}_{m_k}|)$ , with a maximum value of  $O(|\mathbf{S}|)$ . In lines 4–8, the creation of the action space takes  $O(|\mathbf{US}_{m_k}| \cdot |\mathbf{M}|)$ . In line 9, initializing the Q-table takes  $O(|Z| \cdot |A|)$ . Lines 10–18 involve the learning process, which runs for  $MAX$  rounds, with each round involving up to  $|Z|$  state transitions, resulting in a time complexity of  $O(|MAX| \cdot |Z|)$ . Finally, line 19 updates the extension scheme, which takes  $O(1)$ . Therefore, the overall time complexity of the algorithm is:  $O(|\mathbf{S}| \cdot |\mathbf{M}| + |Z| \cdot |A| + |MAX| \cdot |Z|)$ .

### 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. We develop a caching replacement algorithm specifically designed for scenarios where available capacity is insufficient to accommodate the services designated for caching. To better describe the use

**Algorithm 5** 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**  $\hat{\mathbf{I}}$ ;

---

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 of the strategy are shown in Algorithm 5. The inputs to Algorithm 5, as described in Algorithm 4, include 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 already been cached on the edge server  $\mathbf{S}_{m_k}$ . In line 2, we check the remaining capacity  $R_{m_k}^{re}$  of server  $m_k$ . If the remaining capacity can support the caching of service  $s_j$ , we update  $\mathbf{S}_{m_k}$  directly. Otherwise, we calculate the replication density  $\rho_{s_j}^{m_k}$  for the services cached

on  $m_k$  in line 6. The candidate set for updating services,  $\mathbf{CS}_{m_k}$ , consists of those services whose density exceeds a defined threshold  $\rho_0^{m_k}$ , as set in Algorithm 3 (lines 3 to 8). In line 9, we calculate the visit weights for the services in  $\mathbf{CS}_{m_k}$ . We then remove the service with the minimum visit weight from  $\mathbf{CS}_{m_k}$  and from  $\mathbf{S}_{m_k}$  until the remaining capacity of  $m_k$  can accommodate  $s_j$ . The set  $\mathbf{CS}_{m_k}$  is updated accordingly in line 14. We then adjust the remaining capacity and, in line 16, update  $\mathbf{S}_{m_k}$  to include service  $s_j$ . These operations are repeated for each service in  $\mathbf{US}'_{m_k}$ . The final updated caching strategy is denoted as  $\hat{\mathbf{I}}$ .

The time complexity of Algorithm 5 is  $O(|\mathbf{US}'_{m_k}| \times |\mathbf{S}_{m_k}|)$ . The outermost loop, which processes each service in  $\mathbf{US}'_{m_k}$ , requires  $|\mathbf{US}'_{m_k}|$  iterations. Calculating the replication density for services in  $\mathbf{S}_{m_k}$  is performed  $|\mathbf{S}_{m_k}|$  times. Additionally, the operations for calculating the visit weight  $v_{s_l}$  and removing services are carried out at most  $|\mathbf{S}_{m_k}|$  times. Other operations are constant in time, and thus negligible. Hence, the overall time complexity is:  $O(|\mathbf{US}'_{m_k}| \times |\mathbf{S}_{m_k}|)$ .

## V. EXPERIMENT

### A. Basic Setting

We conduct extensive experiments to validate the effectiveness of our algorithms. All experiments were implemented in Python 3.8 and executed on a Windows 11 platform equipped with a 12th Gen Intel Core i9-12900KF CPU @ 3.19 GHz, NVIDIA RTX3080 GPU, and 32GB of memory. We used a dataset provided by China Telecom Shanghai Company [38], which contains information on 3,233 base station locations and corresponding user connections in June 2014. For the experiments, we randomly select subsets of 10, 20, and 30 base stations, each equipped with a server, to construct our server dataset. We then 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 added sequentially; 2) Group 2: 20 servers, 30 services, 70 initial users, and 10 new users added sequentially; 3) Group 3: 30 servers, 45 services, 110 initial users, and 10 new users added sequentially.

1) *Service Caching Experiment*: For the single-area scenario, we evaluate our proposed USC algorithm against several baseline methods. Specifically, we consider three greedy algorithms—Greedy based on the Number of users for services (GN), Greedy based on the Size of services (GS), and Greedy based on the Evaluation Indicator of services (GEI), as well as one random algorithm (RP) and a genetic algorithm [39], [40]. The GN algorithm prioritizes caching services that serve the largest number of users, whereas the GS algorithm favors caching services with smaller sizes. Similarly, the GEI algorithm selects services based on higher service evaluation indicators. In contrast, the RP algorithm caches services randomly without any specific criteria. The genetic algorithm utilizes the Non-dominated Sorting Genetic Algorithm II (NSGA-II) to determine the caching configuration on edge servers. To evaluate the performance of these algorithms, we adjust the weight settings for  $\mathbf{D}$  and  $\mathbf{C}$ . The

weights,  $\alpha$  and  $\beta$ , are varied within the range [0.1, 0.9] with an interval of 0.1 to explore different trade-offs between user QoS and system cost. For the multiple-area scenario, the feasible caching scheme obtained from the single-area scenario algorithms is used as the initial solution for our USC-TS algorithm. Subsequent comparisons of experimental results across different initial solutions assess the effectiveness of the USC-TS algorithm.

2) *Service Updating Experiment*: To evaluate the decision-making process for service updating, we compare our proposed SUD algorithm with two baseline approaches: Service Updating from the Cloud (SU-C) and Service Updating from the Edge or Cloud (SU-EC), using datasets at three different scales. The SU-C approach caches all new services and migrates service replications exclusively from the cloud. In contrast, the SU-EC approach caches all new services while migrating service replications from edge servers if the service is already deployed on any edge server; otherwise, replications are migrated from the cloud. For service updating replacement strategies, the evaluation compares the proposed SRD algorithm with two alternatives: the Least Recently Used (LRU) algorithm and a random replacement algorithm. Considering the impact of long-term dynamic behavior on the effectiveness of these strategies, performance is assessed by comparing the average results over five time slots.

### B. Experiment Results

A comprehensive evaluation of the proposed algorithms was conducted across various scales and preference configurations to assess their performance in service caching and updating at different stages. The experimental framework incorporates heterogeneous servers, each characterized by unique constraints in storage capacity, computational resources, and associated storage costs. These diverse resources are strategically allocated for caching services, thereby reflecting realistic conditions in edge computing environments.

1) *Service Caching Experiment*: The experimental results for the initial service caching stage are presented in Figures 3 through 6. As shown in Figure 3, the USC algorithm consistently outperforms other methods across three different dataset scales, with its superior performance becoming increasingly pronounced as the dataset scale expands. In the first scenario, involving 10 servers, 15 services, and 40 users, the objective values produced by the USC algorithm range from 10 to 50 across 10 distinct weight configurations. This narrow range reflects the limited scope of the scenario, where fewer servers and users lead to minimal variation between the lowest and highest objective values. Consequently, both individual solutions and comparisons between different algorithms exhibit less variation in this smaller scale scenario.

As the scenario grows in size (group 2), with 20 servers, 30 services, and 80 users, the range of objective values becomes a bit larger, from 10 to 70. Finally, in the largest scenario (group 3), with 30 servers, 45 services, and 120 users, the objective values range from 30 to 150. This broader range reflects the increased complexity of the scenario, resulting in greater variability in the objective values. The USC algorithm

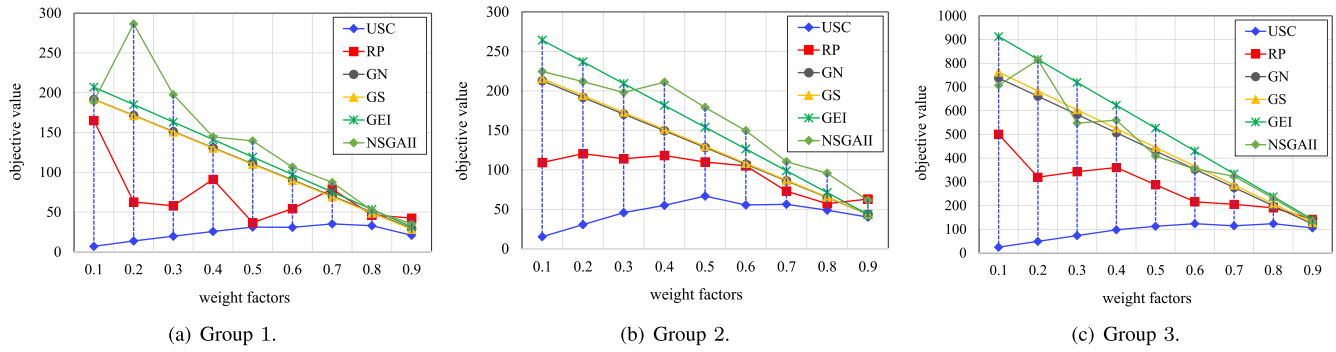


Fig. 3. Comparison of caching algorithms under different weight settings at the initial service caching stage.

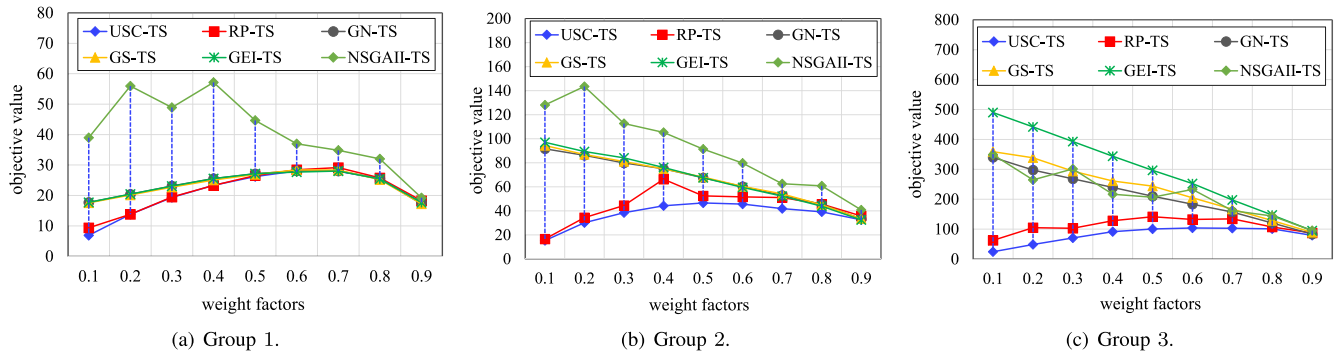


Fig. 4. Comparison of improved caching algorithms under different weight settings at the initial service caching stage.

distinguishes itself by balancing user latency and storage cost, whereas the greedy algorithms optimize only one aspect. Figures 3 and 4 indicate that although the objective values of all algorithms increase with the dataset scale, the USC algorithm exhibits the smallest increase, demonstrating greater robustness relative to the other basic caching algorithms. Similarly, the USC-TS algorithm shows comparable robustness when compared with other advanced caching schemes. The weights assigned to **D** and **C** represent the preference for user QoS and system cost, respectively, and different configurations yield varying performance. For example, in Figure 3(a), the objective value increases as the weight for **D** rises to 0.7 and then decreases. In Figure 3(b), the best performance is observed for a weight combination of (0.5, 0.5), while Figure 3(c) indicates that (0.6, 0.4) yields the highest objective value. Overall, the USC algorithm consistently delivers superior performance across different scales and weight settings.

In contrast, the three greedy algorithms (GN, GS, and GEI) exhibit a consistent trend as the weight for **D** increases, whereas the RP and genetic algorithms display irregular behavior. The objective value is calculated based on four variables:  $\alpha$ , **D**,  $\beta$ , and **C**; **D** and **C** are calculated based on the service placement results of algorithms. The service placement for the greedy algorithms is solely determined by their respective criteria and is not influenced by the QoS and cost preferences. Typically, the value of **D** is usually lower than that of **C**, thus, as the weight for **D** increases, the objective values for these algorithms tend to decrease. In contrast, the RP and genetic algorithms, which rely on randomness, exhibit more

variable objective values. In summary, the USC algorithm adjusts service placement based on the preferences for **D** and **C**, thereby achieving favorable objective values under various weight settings. Figure 4 illustrates the performance when multiple areas are considered, demonstrating improved results across all algorithms compared to single-area scenarios. Notably, even the GEI algorithm—which performs worst in the single-area scenario—shows significant optimization in the multi-area setting. In Group 3, the optimization effect nearly doubles, highlighting the benefits of considering multiple areas when caching services. The USC-TS algorithm, which uses the USC algorithm’s solution as its initial solution, outperforms all other methods across all groups regardless of scale or weight. However, the trend in objective values under different weights in Group 1 differs from those in the larger groups.

This difference is elucidated in Figure 6, which presents the delay and cost associated with caching schemes across the three scales. In Group 1, as shown in Figure 6(a), the values of cost **C** and delay **D** are very close, causing the objective value to first increase and then decrease as the weight increases. This behavior is primarily due to the request distribution in Group 1, where a small number of users account for a significant proportion of service requests. As a result, certain services—particularly under the GN algorithm—are not placed on edge servers, leading to high delay values **D** that approach the cost values **C**. Similarly, the GS algorithm fails to cache services that are both large and frequently requested, resulting in unserved users. The GEI algorithm

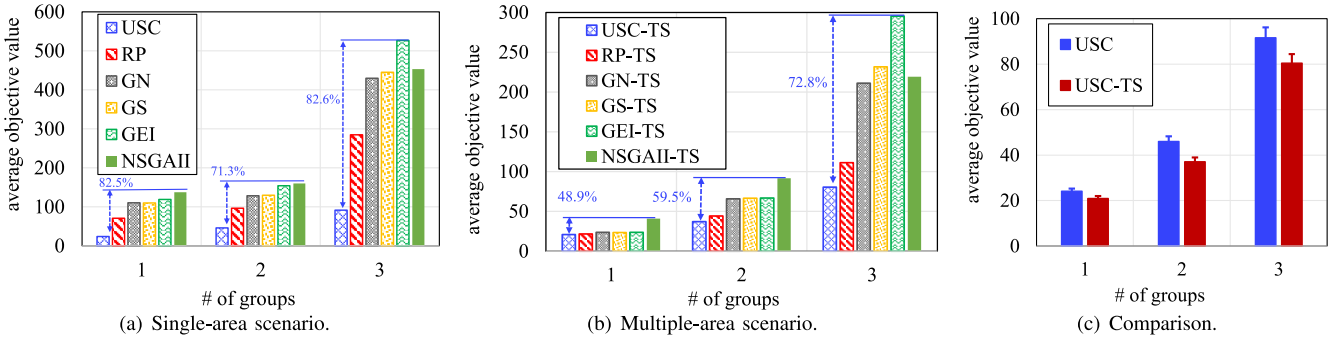


Fig. 5. Comparison of caching algorithms of single-area and multiple-area scenarios.

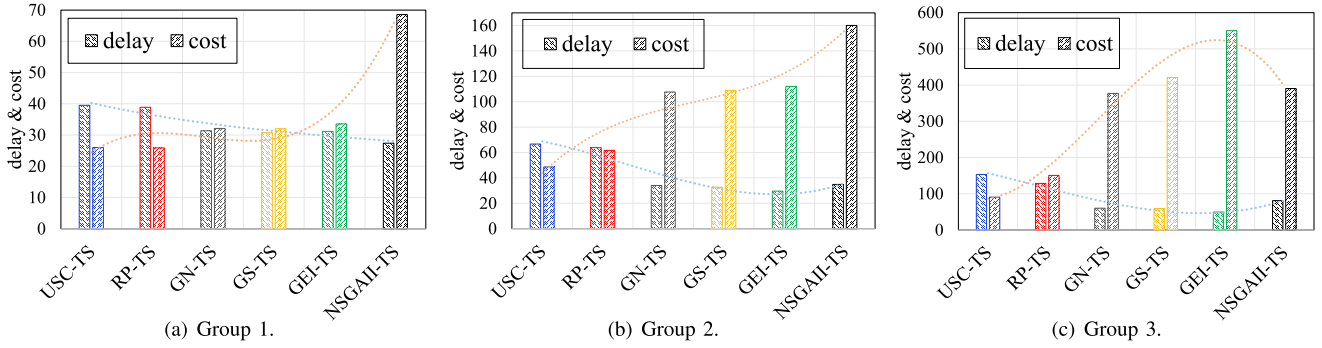


Fig. 6. Delay and cost of caching algorithms in the multiple-area scenario.

exhibits similar performance due to its reliance on comparable greedy strategies. In Groups 2 and 3, where the scale is larger, the proportion of such services decreases, leading to a more pronounced gap between  $\mathbf{D}$  and  $\mathbf{C}$  and a corresponding downward trend in the objective values for the greedy algorithms.

Figures 5 and 6 further illustrate the trade-offs between delay and cost for different caching algorithms. While Figure 5 presents the average results of the caching algorithms in single-area and multi-area scenarios across 10 weight configurations for the three dataset scales, Figure 6 provides a detailed breakdown of the delay and cost for each group. Figures 5(a) and 5(b) report the percentage reduction in the objective function value achieved by the USC algorithm relative to the worst-performing algorithm. These results clearly demonstrate that the superiority of the USC algorithm becomes increasingly apparent from Scale 1 to Scale 3 in single-area scenarios, with the USC-TS algorithm exhibiting similar improvements in multi-area scenarios. Figure 5(c) further confirms that the objective value is significantly reduced across all groups when employing the USC-TS algorithm.

2) *Service Updating Experiment*: 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 Table II. The results indicate that the SUD algorithm outperforms both the SUC and SUEC algorithms in all three groups. For instance, in Group 1, the SUD algorithm achieves an average objective value of  $63.16 \pm 25.04$ , whereas the SUC and SUEC algorithms yield  $138.35 \pm 74.14$  and  $130.86 \pm 76.6$ , respectively. These findings demonstrate that the SUD algorithm

TABLE II  
COMPARISON OF DECISION-MAKING ALGORITHMS

Group	SUD	SU-C	SU-EC
1	$63.16 \pm 25.04$	$138.35 \pm 74.14$	$130.86 \pm 76.6$
2	$115.02 \pm 62.61$	$307.19 \pm 177.82$	$288.83 \pm 181.16$
3	$226.02 \pm 93.03$	$594.66 \pm 39.95$	$577.71 \pm 346.34$

not only attains the lowest objective value but also exhibits the smallest variance, reflecting its stability and efficiency. Similar superior performance is observed in Groups 2 and 3. This advantage is attributed to the SUD algorithm's strategy of selectively migrating only certain services for caching, as opposed to the indiscriminate migration employed by the other approaches. Moreover, the SUD algorithm strategically selects the most appropriate servers to provide replications, rather than relying solely on cloud servers or random edge server selection.

In the service replacement stage, the proposed SRD algorithm is compared with the SR-LRU and SR-R algorithms on two dataset scales, as shown in Figure 7. For each dataset, performance is evaluated under two weight configurations:  $(\alpha = 0.4, \beta = 0.6)$  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  $(\alpha = 0.6, \beta = 0.4)$ . As illustrated in Figures 7(a) and 7(b), the SRD algorithm consistently outperforms the other methods in both weight settings for Groups 1 and 2. In Group 1, the SR-LRU and SR-R algorithms exhibit unstable performance under varying weight configurations, whereas the SRD algorithm consistently achieves the lowest objective value. Although the

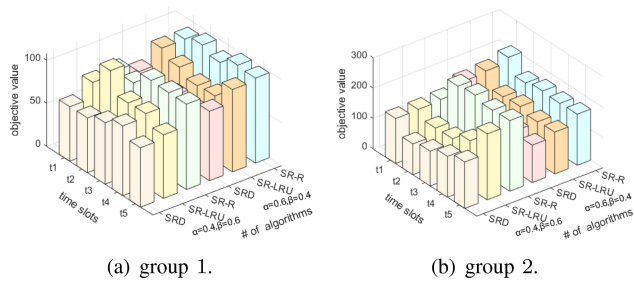


Fig. 7. Comparison of replacement algorithms at the service updating stage.

objective value for SRD is slightly higher under the  $(\alpha = 0.6, \beta = 0.4)$  configuration compared to  $(\alpha = 0.4, \beta = 0.6)$ , this trend is consistent across both groups. We further compare the algorithms over 5 time slots to assess their long-term performance. From Figures 7(a) and 7(b), it is evident that the SRD algorithm consistently achieves the best objective value in each time slot. The strength of SRD lies in its dual consideration of service access frequency and the distribution of service replications. When replications of the same service become overly dense, storage resources are used inefficiently, rendering further replication unnecessary.

In summary, the experimental results convincingly demonstrate the effectiveness and superiority of the proposed algorithms across various stages. The USC and USC-TS algorithms perform exceptionally well during the initial service caching stage, while the SUD and SRD algorithms offer significant advantages in the service updating stage, effectively reducing latency and cost.

## VI. CONCLUSION

This paper addresses the critical challenge of service caching in dynamic, resource-constrained MEC environments. We explore strategies to achieve dynamic service caching and updating in a cost-efficient manner while operating under the constraints of limited storage resources and ensuring Quality of Service (QoS) for users. The primary objective is to minimize user latency and system costs when dynamically caching and updating services within these environments. In the initial service caching stage, both single-area and multi-area scenarios are considered. For the single-area scenario, we design a service caching algorithm based on dynamic programming, while for the multi-area scenario, we develop an improved service caching algorithm using tabu search to optimize service provisioning at appropriate locations. To address the challenges posed by dynamic user behavior, we introduce a reinforcement learning-based algorithm for determining service extensions. In the service updating and replacement stage, a novel replacement algorithm is proposed that takes into account the distribution of service replications and service access frequency.

The effectiveness and performance of these algorithms are validated through extensive experiments, demonstrating significant improvements in latency reduction and cost efficiency. In summary, this paper provides an in-depth theoretical analysis and presents innovative solutions to the problem of dynamic

service caching and updating in edge computing, offering valuable insights for research and applications in related fields.

While this work focuses on dynamic service caching and updating under resource constraints, it does not consider user movement trajectories. In future work, we will explore caching prefetching methods based on user movement trajectories to further enhance QoS for users.

## REFERENCES

- [1] Z. Xu, L. Zhou, S. C.-K. Chau, W. Liang, Q. Xia, and P. Zhou, "Collaborate or separate? Distributed service caching in mobile edge clouds," in *Proc. INFOCOM IEEE Conf. Comput. Commun.*, 2020, pp. 2066–2075.
- [2] S. Chen, L. Jiao, F. Liu, and L. Wang, "EdgeDR: An online mechanism design for demand response in edge clouds," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 2, pp. 343–358, Feb. 2022.
- [3] S. Ko, S. J. Kim, H. Jung, and S. W. Choi, "Computation offloading and service caching for mobile edge computing under personalized service preference," *IEEE Trans. Wireless Commun.*, vol. 21, no. 8, pp. 6568–6583, Aug. 2022.
- [4] H. Li, M. Sun, F. Xia, X. Xu, and M. Bilal, "A survey of edge caching: Key issues and challenges," *Tsinghua Sci. Technol.*, vol. 29, no. 3, pp. 818–842, 2023.
- [5] P. Jin, X. Fei, Q. Zhang, F. Liu, and B. Li, "Latency-aware VNF chain deployment with efficient resource reuse at network edge," in *Proc. INFOCOM IEEE Conf. Comput. Commun.*, 2020, pp. 267–276.
- [6] B. Gao, Z. Zhou, F. Liu, F. Xu, and B. Li, "An online framework for joint network selection and service placement in mobile edge computing," *IEEE Trans. Mobile Comput.*, vol. 21, no. 11, pp. 3836–3851, Nov. 2022.
- [7] S. Bi, L. Huang, and Y.-J. A. Zhang, "Joint optimization of service caching placement and computation offloading in mobile edge computing systems," *IEEE Trans. Wireless Commun.*, vol. 19, no. 7, pp. 4947–4963, Jul. 2020.
- [8] H. Zhou, Z. Zhang, D. Li, and Z. Su, "Joint optimization of computing offloading and service caching in edge computing-based smart grid," *IEEE Trans. Cloud Comput.*, vol. 11, no. 2, pp. 1122–1132, Apr.–Jun. 2023.
- [9] G. Zhao, H. Xu, Y. Zhao, C. Qiao, and L. Huang, "Offloading dependent tasks in mobile edge computing with service caching," in *Proc. INFOCOM IEEE Conf. Comput. Commun.*, 2020, pp. 1997–2006.
- [10] F. A. Salaht, F. Desprez, and A. Lebre, "An overview of service placement problem in fog and edge computing," *ACM Comput. Surv.*, vol. 53, no. 3, pp. 1–35, 2020.
- [11] 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," *IEEE J. Sel. Areas Commun.*, vol. 41, no. 3, pp. 690–705, Mar. 2023.
- [12] H. Zhou, H. Wang, Z. Yu, G. Bin, M. Xiao, and J. Wu, "Federated distributed deep reinforcement learning for recommendation-enabled edge caching," *IEEE Trans. Services Comput.*, vol. 17, no. 6, pp. 3640–3656, Nov./Dec. 2024.
- [13] Q. Chen, Z. Zheng, C. Hu, D. Wang, and F. Liu, "On-edge multi-task transfer learning: Model and practice with data-driven task allocation," *IEEE Trans. Parallel Distributed Syst.*, vol. 31, no. 6, pp. 1357–1371, Jun. 2020.
- [14] S. Chen, L. Wang, and F. Liu, "Optimal admission control mechanism design for time-sensitive services in edge computing," in *Proc. IEEE INFOCOM Conf. Comput. Commun.*, 2022, pp. 1169–1178.
- [15] L. Pan, L. Wang, S. Chen, and F. Liu, "Retention-aware container caching for serverless edge computing," in *Proc. IEEE INFOCOM Conf. Comput. Commun.*, 2022, pp. 1069–1078.
- [16] J. Wu, L. Wang, Q. Jin, and F. Liu, "Graft: Efficient inference serving for hybrid deep learning with SLO guarantees via DNN re-alignment," *IEEE Trans. Parallel Distrib. Syst.*, vol. 35, no. 2, pp. 280–296, Feb. 2024.
- [17] J. Wu, L. Wang, Q. Pei, X. Cui, F. Liu, and T. Yang, "HiTDL: High-throughput deep learning inference at the hybrid mobile edge," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 12, pp. 4499–4514, Dec. 2022.
- [18] Q. Pei, L. Wang, D. Zhang, B. Yan, C. Yu, and F. Liu, "InferCool: Enhancing ai inference cooling through transparent, non-intrusive task reassignment," in *Proc. ACM Symp. Cloud Comput.*, 2024, pp. 487–504.

- [19] Z. Sun, X. Guan, J. Wang, F. Liu, and H. Cui, "New problems in distributed inference for DNN models on robotic IoT," in *Proc. Workshop Adv. Tools, Program. Lang., Platforms Implement. Eval. Algorithms Distrib. Syst.*, 2024, pp. 1–9.
- [20] L. Lei, L. You, G. Dai, T. X. Vu, D. Yuan, and S. Chatzinotas, "A deep learning approach for optimizing content delivering in cache-enabled HetNet," in *Proc. Int. Symp. Wireless Commun. Syst. (ISWCS)*, 2017, pp. 449–453.
- [21] R. Wang, M. Li, L. Peng, Y. Hu, M. M. Hassan, and A. Alelaiwi, "Cognitive multi-agent empowering mobile edge computing for resource caching and collaboration," *Future Gener. Comput. Syst.*, vol. 102, pp. 66–74, Jan. 2020.
- [22] G. Qu, Z. Lin, F. Liu, X. Chen, and K. Huang, "TrimCaching: Parameter-sharing AI model caching in wireless edge networks," 2024, *arXiv:2405.03990*.
- [23] S. Zhong, S. Guo, H. Yu, and Q. Wang, "Cooperative service caching and computation offloading in multi-access edge computing," *Comput. Netw.*, vol. 189, Apr. 2021, Art. no. 107916.
- [24] X. Huang, L. He, X. Chen, G. Liu, and F. Li, "A more refined mobile edge cache replacement scheme for adaptive video streaming with mutual cooperation in multi-MEC servers," in *Proc. IEEE Int. Conf. Multimedia Expo (ICME)*, 2020, pp. 1–6.
- [25] C. Li, M. Song, S. Du, X. Wang, M. Zhang, and Y. Luo, "Adaptive priority-based cache replacement and prediction-based cache prefetching in edge computing environment," *J. Netw. Comput. Appl.*, vol. 165, Sep. 2020, Art. no. 102715.
- [26] E. E. Ugwuanyi, S. Ghosh, M. Iqbal, T. Dagiukas, S. Mumtaz, and A. Al-Dulaimi, "Co-operative and hybrid replacement caching for multi-access mobile edge computing," in *Proc. Eur. Conf. Netw. Commun. (EuCNC)*, 2019, pp. 394–399.
- [27] 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," *IEEE Trans. Wireless Commun.*, vol. 22, no. 12, pp. 9595–9609, Dec. 2023.
- [28] P. Yuan, Y. Cai, Y. Liu, J. Zhang, Y. Wang, and X. Zhao, "ProRec: A unified content caching and replacement framework for mobile edge computing," *Wireless Netw.*, vol. 26, no. 4, pp. 2929–2941, 2020.
- [29] H. Wei, H. Luo, Y. Sun, and M. S. Obaidat, "Value-driven cache replacement strategy in mobile edge computing," in *Proc. GLOBECOM IEEE Global Commun. Conf.*, 2020, pp. 1–6.
- [30] Z. Li, C. Yang, X. Huang, W. Zeng, and S. Xie, "CoOR: Collaborative task offloading and service caching replacement for vehicular edge computing networks," *IEEE Trans. Veh. Technol.*, vol. 72, no. 7, pp. 9676–9681, Jul. 2023.
- [31] 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 Syst. J.*, vol. 16, no. 1, pp. 275–286, Mar. 2021.
- [32] Z. Wang, J. Hu, G. Min, Z. Zhao, and Z. Wang, "Agile cache replacement in edge computing via offline-online deep reinforcement learning," *IEEE Trans. Parallel Distrib. Syst.*, vol. 35, no. 4, pp. 663–674, Apr. 2024.
- [33] X.-Q. Pham, T.-D. Nguyen, V. Nguyen, and E.-N. Huh, "Joint service caching and task offloading in multi-access edge computing: A QoE-based utility optimization approach," *IEEE Commun. Lett.*, vol. 25, no. 3, pp. 965–969, Mar. 2021.
- [34] K. Kazari, F. Ashtiani, and M. Mirmohseni, "Cache update and delivery of dynamic contents: A stochastic game approach," *IEEE Trans. Mobile Comput.*, vol. 23, no. 4, pp. 3035–3047, Apr. 2024.
- [35] G. Yu, Y. He, J. Wu, Z. Chen, and J. Pan, "Mobility-aware proactive edge caching for large files in the Internet of Vehicles," *IEEE Internet Things J.*, vol. 10, no. 13, pp. 11293–11305, Jul. 2023.
- [36] P. Li, Y. Zhang, H. Zhang, W. Wang, K. Xu, and Z. Zhang, "A delayed eviction caching replacement strategy with unified standard for edge servers," *Comput. Netw.*, vol. 230, Jul. 2023, Art. no. 109794.
- [37] C. Zhang, H. Tan, G. Li, Z. Han, S. H. C. Jiang, and X. Y. Li, "Online file caching in latency-sensitive systems with delayed hits and bypassing," in *Proc. INFOCOM IEEE Conf. Comput. Commun.*, 2022, pp. 1059–1068.
- [38] Y. Li, A. Zhou, X. Ma, and S. Wang, "Profit-aware edge server placement," *IEEE Internet Things J.*, vol. 9, no. 1, pp. 55–67, Jan. 2021.
- [39] A. M. Maia, Y. Ghamri-Doudane, D. Vieira, and M. F. de Castro, "A multi-objective service placement and load distribution in edge computing," in *Proc. IEEE Global Commun. Conf. (GLOBECOM)*, 2019, pp. 1–7.
- [40] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Trans. Evol. Comput.*, vol. 6, no. 2, pp. 182–197, Apr. 2002.



**Shuaibing Lu** (Member, IEEE) received the Ph.D. degree in computer science and technology from Jilin University, Changchun. She is currently a Lecturer with the College of Computer Science, Beijing University of Technology. She was supported by the China Scholarship Council as a Visiting Scholar supervised by Prof. J. Wu with the Department of Computer and Information Sciences, Temple University. Her current research focuses on distributed computing and edge computing.



**Xin Jin** received the B.S. degree in electronic science and technology from the Hebei University of Technology. She is currently pursuing the M.S. degree with the College of Computer Science, Beijing University of Technology. Her research interests include cloud computing and edge computing.



**Jie Wu** (Fellow, IEEE) is the Director of the Center for Networked Computing and a Laura H. Carnell Professor with Temple University. He also serves as the Director of International Affairs with the College of Science and Technology. He served as the Chair with the Department of Computer and Information Sciences from Summer 2009 to Summer 2016 and an Associate Vice Provost for International Affairs from Fall 2015 to Summer 2017. Prior to joining Temple University, he was a Program Director of the National Science Foundation and was a Distinguished Professor with Florida Atlantic University. He regularly publishes in scholarly journals, conference proceedings, and books. His current research interests include mobile computing and wireless networks, routing protocols, cloud and green computing, network trust and security, and social network applications. He is the recipient of the 2011 China Computer Federation (CCF) Overseas Outstanding Achievement Award. He serves on several editorial boards, including IEEE TRANSACTIONS ON SERVICE COMPUTING and the *Journal of Parallel and Distributed Computing*. He was the General Co-Chair of IEEE MASS 2006, IEEE IPDPS 2008, IEEE ICDCS 2013, ACM MobiHoc 2014, ICPP 2016, and IEEE CNS 2016, as well as the Program Co-Chair for IEEE INFOCOM 2011 and CCF CNCC 2013. He was an IEEE Computer Society Distinguished Visitor, an ACM Distinguished Speaker, and the Chair of the IEEE Technical Committee on Distributed Processing. He is a CCF Distinguished Speaker.



**Shuyang Zhou** is currently a Junior with Beijing Jiaotong University, majoring in software engineering. His current research interests are focused on edge computing and cloud computing.



**Jackson Yang** is currently pursuing the undergraduate degree with the School of Software, Beijing Jiaotong University, specializing in software engineering. He has participated in several projects focusing on the development of VR systems for psychological treatments, edge computing, and cloud computing.



**Ran Yan** received the B.Sc. degree in network engineering from the Beijing Information Science and Technology University. She is currently pursuing the M.Sc. degree with College of Computer Science, Beijing University of Technology. Her research interests include cloud computing and edge computing.



**Zhi Cai** (Member, IEEE) received the M.Sc. degree from the School of Computer Science, University of Manchester in 2007, and the Ph.D. degree from the Department of Computing and Mathematics, Manchester Metropolitan University, U.K., in 2011. He is an Associate Professor with the College of Computer Science, Beijing University of Technology, China. His research interests include information retrieval, ranking in relational databases, keyword search, and intelligent transportation systems.



**Haiming Liu** received the B.S. degree in computer science and technology, the M.S. degree in computer software and theory, and the Ph.D. degree in computer science and technology (bioinformatics) from Jilin University, Changchun, in 2012 and 2015, respectively. He is currently a Lecturer with the School of Software Engineering, Beijing Jiaotong University. His current research focuses on the edge computing, data mining, and bioinformatics. He is a member of the Chinese Association for Artificial Intelligence.