Practical Efficient Deployment and Updating for Microservice with Dependencies in Multi-Access Edge Computing

Shuaibing Lu, *Member, IEEE,* Ran Yan, Jie Wu, *Fellow*, IEEE, Zhi Cai, Jackson Yang, Shuyang Zhou, Haiming Liu, and Juan Fang, *Member, IEEE*

Abstract—As mobile edge computing technology advances rapidly, latency-sensitive and resource-intensive applications are being offloaded to edge servers to enhance Quality of Service (QoS) for users. Traditional monolithic architectures, however, struggle to meet the escalating service and traffic requirements of distributed users due to their inherent inflexibility. In response to these challenges, microservices architecture, characterized by scalability and flexibility, has been adopted for dynamic deployment at the network edge. However, the deployment of these lightweight, dependency-rich components in a way that minimally impacts the makespan and maximizes quality of service is complex. Current studies often overlook the deployment of microservices with specific dependencies within constrained environments of edge server clusters and communication links. This paper introduces practical and effective strategies for the deployment and updating of microservices, tailored to various application contexts. Initially, two scenarios are analyzed: one constrained by bandwidth with unlimited storage, and the other by storage with unlimited bandwidth. For each scenario, optimal solutions are developed using a novel enhanced graph construction method. The study progresses to a more intricate scenario involving comprehensive constraints on storage, computation, and communication resources. An optimized deployment method is proposed, utilizing main path embedding followed by an innovative simulated annealing algorithm for iterative refinement. This method is validated by demonstrating that the main path coincides with the critical path. Furthermore, the dynamic reallocation of edge resources is explored through a critical path-based updating algorithm that optimizes microservice locations to reduce overall makespan. Extensive experiments demonstrate that our strategies outperform existing representative benchmark approaches in terms of overall performance and microservice deployment efficiency.

Index Terms—microservice deployment, updating, dependency, high-efficiency, multi-access edge computing.

1 Introduction

With the widespread adoption of mobile devices and the continuous emergence of mobile applications, traditional cloud computing faces a range of challenges, such as high latency, network congestion, and extensive data transmission [1]. Multi-access edge computing, as a flexible and scalable computing platform, pushes computation and data processing to the network edge, closer to users and devices [2]. This effectively reduces the distance of data transmission on the network, significantly reducing latency and improving responsiveness. The limitations of traditional monolithic applications in terms of scalability and flexibility led to the emergence of microservice architecture, a lightweight and highly flexible architectural pattern [3]. By decomposing complex monolithic applications into small, autonomous service units, modularity, scalability, and maintainability are improved. This architecture is widely employed in constructing distributed systems and cloud-native applications. Containerization technology encapsulates microservices and their dependencies

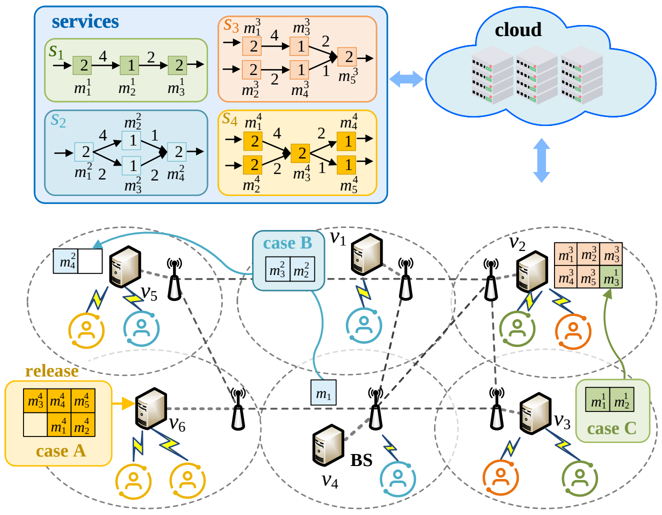


Fig. 1. An illustrating example. v_1 to v_6 denote edge servers, and S_1 to S_4 represent services composed of multiple dependent microservices. The numbers inside nodes indicate processing requirements, and the numbers on arrows denote data-flow sizes.

within lightweight, isolated containers [4], which enables decentralized deployment of services, allowing users to access the services they need without relying on traditional centralized cloud infrastructure. By leveraging container orchestration platforms such as Docker and Kubernetes, deploying microservices on edge servers is feasible.

Nonetheless, the effective deployment of microservices across extensive, distributed environments—including

- *Shuaibing Lu and Ran Yan are with the College of Computer Science, Beijing University of Technology, Beijing, China, 100124. E-mail: lushuaibing@bjut.edu.cn.*
- *Jie Wu is with the China Telecom Cloud Computing Research Institute, Beijing, 100088, China, and with the Department of Computer and Information Sciences, Temple University, 1925 N. 12th St., Philadelphia, PA 19122, USA. E-mail: jiewu@temple.edu.*
- *Haiming Liu is with the School of Software Engineering, Beijing Jiaotong University, Beijing, China, 100091. E-mail: liuhaiming@bjtu.edu.cn.*

both cloud frameworks and edge devices—continues to pose significant challenges. The inherent complexity of managing interdependent microservices intensifies these challenges, demanding nuanced solutions. Central to this discourse are three pivotal concerns:(i) How can complex dependencies between microservices be effectively dealt with to improve overall efficiency? (ii) How to balance the trade-off between processing and transmission time for optimal deployment without overwhelming resource constraints? (iii) How to update the locations of microservices to reduce the overall makespan? Addressing these issues necessitates sophisticated strategies for managing dependencies, optimizing deployment tactics, and maintaining the system's overall performance and stability. This paper focuses on developing optimized deployment strategies for microservices, particularly those with complex dependencies, within multi-access edge computing environments, aiming to reduce the overall makespan across varied scenarios.

1.1 Motivation and challenge

In a complex and real-time critical autonomous driving system, several key services work together, such as sensor data processing, path planning, vehicle control, and perception of the environment. These services are divided into several microservices, each of which focuses on specific functions and has complex dependencies. For example, the sensor data processing service collects, caches, and processes data from sensors such as cameras, radars, and LiDAR. The environment perception service integrates and analyzes sensor data to identify and track objects in the environment, e.g., vehicles, pedestrians, and obstacles. The path planning service dynamically calculates the optimal route for the current situation. The vehicle control service generates control commands in real time and controls steering, acceleration, and braking. We use the following example to illustrate these challenges.

We assume four representative services, denoted as S_1, S_2, S_3, and S_4. Each service consists of multiple microservices with dependencies, and the resulting workflow can be modeled as a directed acyclic graph. Figure 1 provides an illustrative example to demonstrate the challenges of microservice deployment and updating under such dependency constraints. We use v_k denotes the k-th edge server with computing capability $c(v_k)$, and m_i^h represents the i-th microservice of service S_h, where the numeric label inside the node indicates its required processing capacity $q_{m_i^h}$. The directed arrows indicate dependency relationships between microservices, and the numeric labels on these arrows indicate the size of the corresponding data flows $r_{m_i^h \to m_j^h}$. (i) Finding an effective combination that can reduce the makespan is non-trivial since merging or splitting dependent microservices introduces complexity. Taking S_3 as an example, one extreme solution is to deploy all the microservices in S_3 on v_2. This strategy is shown as case A, which will bring no transfer time of S_3. However, the edge servers are heterogeneous with different storage and computing capabilities, which might result in an extremely high computation time when the computing capacity is very poor. Another extreme solution is to deploy the microservices separately on edge servers with weak storage but powerful computing capacities, such as v_1, v_4, and v_5, as shown in case B. This strategy would increase the communication time and have a negative impact on the makespan. Therefore, it is essential to consider the connections between microservices, which can be either merged or split within the service. (ii) Since the distance and communication bandwidth between every pair of edge servers are different, determining which path to transmit data generated by the combination of microservices is non-trivial. Taking case C as an example, we choose to deploy the microservices in S_1 on servers v_2 and v_3 with the path that has the largest communication capacity. However, the computing capacities of v_2 and v_3 are lower than those of other servers, which in turn increases the makespan. We are faced with the trade-off of whether to prioritize servers with shorter distances but higher connection bandwidth to reduce communication time, or to place microservices on servers with higher computing power to improve performance. Therefore, the problem of how to achieve an efficient microservice deployment by jointly considering the computing and bandwidth capacities under the storage constraint is a challenge. (iii) Due to the dynamic release of services, finding an updating strategy that can reduce the makespan is non-trivial. Take S_4 as an example, when S_4 has completed its task and requests the release of resources, server v_6 transitions to an idle state. We assume that the completion time of S_2 is the longest in the system. One updating strategy is to migrate all services of S_2 to v_6 to reduce communication time. However, when v_6 computing capacity is very poor, this strategy may result in extremely high computation time, thus increasing the makespan. Another updating strategy is to migrate some services from S_2 to v_6, but due to the dependencies among microservices, selecting a combination to migrate introduces complexity to the updating problem. For instance, migrating m_2^1 and m_2^3 to v_6 would involve three communication instances, increasing communication time. Another updating strategy is to refrain from migration or migrate services with lower completion times. However, such updating strategies may not necessarily reduce the makespan and, in certain situations, could potentially increase the makespan. Therefore, determining how to merge or migrate microservices to decrease makespan also poses a significant challenge.

1.2 Contributions and paper organization

To address the aforementioned challenges, we investigate the efficient deployment of interdependent microservices with the goal of minimizing the makespan while improving system performance and resource utilization. The main idea of our approach is to formulate the microservice placement problem within a unified optimization framework that explicitly incorporates dependency-aware resource allocation and enables efficient, high-quality deployment decisions. The major contributions of this paper are summarized as follows:

- The microservice deployment problem with dependencies is examined, aiming to minimize the makespan of multiple services under storage, computing, and communication constraints in multi-access edge computing. The complexity of this problem is theoretically demonstrated by proving it to be NP-hard

- Four strategies for deploying microservices are proposed, providing flexibility and adaptability for different application scenarios. Initially, two straightforward scenarios are considered: one with unlimited storage resources under the bandwidth constraint, and the other with unlimited bandwidth resources under the storage constraint. A novel enhanced graph construction method is introduced, and two optimal solutions are designed for each of these scenarios.
- Resource constraints on storage, computing, and communication necessitate a tailored approach to microservice deployment. Accordingly, this study introduces a feasible solution through an effective embedding method that leverages novel definitions of the main path and the preferred server. These definitions are derived from the topology features of the services and the edge environment. Furthermore, the study proposes an updated deployment method using an enhanced simulated annealing strategy. The complexity of this scenario is rigorously analyzed to validate the proposed approach.
- The discussion extends to a more complex and realistic scenario in which all services deployed on edge servers dynamically release resources after completing their tasks. An update algorithm, based on the critical path, adjusts the locations of existing microservices to ensure rapid response to service requests and to minimize the overall makespan.
- Extensive experiments were conducted to evaluate the performance of our strategies against several benchmarks using the dataset from China Telecom Shanghai Company, which includes the geographic information of 3,233 base stations. The results indicate that the proposed methods not only enhance performance but also reduce time complexity across various scenarios.

The remainder of this paper is organized as follows. Section 2 surveys related work. Section 3 introduces the models and presents the problem. Section 4 explores four distinct dependent microservice deployment and updating strategies for different scenarios. Section 5 presents the experimental results. Finally, Section 6 provides a summary of the entire paper.

2 Related work

Multi-access edge computing has recently emerged as a significant research domain, with microservices architecture being extensively utilized in distributed and cloud-native systems. Research primarily focuses on the deployment and dynamic updating of microservices.

2.1 Microservice deployment

In mobile edge computing, improper deployment of microservices can lead to increased transmission delays and server overloads. Innovative solutions include Ding et al. [5], who improved the genetic algorithm for microservice placement, and Tang et al. [6], who minimized the resource consumption cost through adaptive deployment optimization. Wang et al. [7] and Samanta et al. [8] reducing the overall latency. Samanta et al. [8] proposed algorithms to reduce latency and maximize resource utilization, respectively. Moreover, studies like those by Gu et al. [9], [11] and Kumar et al. [12] have enhanced throughput and optimized resource allocation using heuristic and algorithmic approaches.

Significant efforts also address the deployment complexities of interdependent microservices. For example, Zhao et al. [13] and Niu et al. [14] focused on optimizing deployment configurations and minimizing response times through various algorithmic strategies. Guerrero et al. [15] and Pallewatta et al. [16], [17], [18] employed optimization techniques to enhance microservice efficiency and reduce makespan. Other notable contributions by Liao et al. [19], Wang et al. [23], [24], and Qi et al. [25] have prioritized task allocation, context extraction, and collaborative delivery strategies to improve edge computing performance.

2.2 Microservice updating

Dynamic updating of microservices in edge computing environments has been addressed by several researchers. Li et al. [26] optimized network overhead using a heuristic graph mapping algorithm, while Singh et al. [27] focused on minimizing downtime during microservice updates. Zambianco et al. [28] and Sampaio et al. [29] proposed dynamic re-orchestration and runtime adaptation mechanisms to migrate and reconfigure microservices based on resource utilization. Hossen et al. [30] introduced a feedback-based auto-scaling method that adjusts resource allocation dynamically. While existing works provide foundational insights, many overlook the impacts of server heterogeneity and the dynamic dependencies of microservices on resource allocation. Our work extends these discussions by focusing on real-time resource updates to optimize service delivery and resource utilization in edge computing scenarios.

3 Problem Formulation

3.1 System model

This paper considers a three-layer network architecture as depicted in Figure 1, comprising the cloud data center, edge servers, and end users. Given a substrate topology of an edge network which is modeled as a weighted undirected graph $\mathbf{G}(V, L)$, where $V = \{v_k\}$ and $L = \{l_{(v_k,v_q)}\}$ represent the sets of edge servers and links, respectively. Here, we use v_k to denote the k-th edge server, and $l_{(v_k,v_q)}$ represents the communication link between servers v_k and v_q. The computing capability of edge server v_k is represented as $c_{(v_k)}$, measured in gflop/s, and the communication capacity of $l_{(v_k,v_q)}$ is denoted by $b_{(v_{k,q})}$, measured in GB/s. This paper combines Docker container technology with a microservices architecture to achieve decentralized service deployment. We encapsulate each microservice within a container, allowing users to access the required services without relying on traditional centralized cloud infrastructure. Therefore, we quantify the capacity of each server in terms of slots. Each edge server has a storage capacity, denoted by $\phi_{(v_k)}$, representing the maximum number of microservices it can accommodate. In addition, we assume that the services required by the users have been originally provisioned in the cloud data center [20], which is represented by set $\mathbf{S} = \{S_h\}$. Here, we use S_h to denote the h-th service that consists of a set of microservices $M_h = \{m_i^h\}$ and directed links $E_h = \{e^h_{m_i \to m_j}\}$, i.e., $S_h = \{M_h, E_h\}$. Let m_i^h represent the i-th microservice of S_h. The required processing capability of m_i^h is denoted as $q_{m_i^h}$, measured in $gflops$. Let $e^h_{m_i \to m_j}$ represent the dependency between m_i^h and m_j^h. We use $r^h_{m_i \to m_j}$ to denote the corresponding data flow size between microservices m_i^h and m_j^h, representing the weight of directed edge $e^h_{m_i \to m_j}$, measured in GB. The symbols used in this paper are summarized in Table 1.

TABLE 1
Summary of Symbols in the System Model

Symbol	Definition
$\mathbf{S}$	Set of services, where $\mathbf{S} = \{S_h\}$.
S_h	The h-th service, where $S_h = \{M_h, E_h\}$.
M_h	Set of microservices of S_h, where $M_h = \{m_i^h\}$.
E_h	Set of directed links between microservices of S_h, i.e., $E_h = \{e^h_{m_i \to m_j}\}$.
$\mathbf{G}$	Topology of the edge network, where $\mathbf{G} = \{V, L\}$.
V	Set of edge servers, where $V = \{v_k\}$.
L	Set of links between edge servers, $L = \{l_{(v_k,v_q)}\}$.
P	Set of simple paths in graph $\mathbf{G}$, where $P = \{p_k\}$.
w_{p_i}	Weight of path p_i.
$q_{m_i^h}$	Required processing capability for microservices m_i^h, measured in $gflops$.
$r^h_{m_i \to m_j}$	Data flow size between microservices m_i^h and m_j^h, measured in GB.
$b_{(v_k,v_q)}$	Communication capability of link $l_{(v_k,v_q)}$, measured in GB/s.
$c_{(v_k)}$	Computing capability of edge server v_k.
$\phi_{(v_k)}$	Storage capability of edge server v_k.
$d_c(m_i^h)$	Processing time of m_i^h on an edge server.
$d_l(e^h_{m_i \to m_j})$	Transmission latency between m_i^h and m_j^h.
$f(m_i^h)$	Completion time of microservice m_i^h.
$x(i,k)$	Boolean variable indicating whether m_i^h deploys on edge server v_k.
$y(i,j)$	Boolean variable indicating whether m_i^h and m_j^h with dependency are co-located.

3.2 Computation and Communication Models

In our work, the completion time of a service includes both the processing time of the microservices and the time required for data flow communication. We use $d_c(m_i^h)$ to represent the processing time of microservice m_i^h on an edge server, which is formulated as follows.

$$d_c(m_i^h) = x(i,k) \cdot q_{m_i^h} / c_{(v_k)} \tag{1}$$

We use $x(i,k)$ to indicate whether microservice m_i^h is deployed on edge server v_k. If m_i^h is deployed on v_k, then $x(i,k) = 1$; otherwise, $x(i,k) = 0$. Here, $q_{m_i^h}$ represents the processing workload required by m_i^h, and $c_{(v_k)}$ denotes the computing capability of edge server v_k. When inter-dependent microservices are deployed on different servers, the data transfer between them incurs communication time. Here, we use $d_l(e^h_{m_i \to m_j})$ to represent the transferring time of two dependent microservices, where m_i^h is the predecessor of m_j^h. The communication time is given by:

$$d_l(e^h_{m_i \to m_j}) = y(i,j) \cdot r^h_{m_i \to m_j} / b_{(v_k,v_q)} \tag{2}$$

We use $y(i,j)$ to indicate whether microservices m_i^h and m_j^h are deployed on the same edge server. When two dependent microservices are deployed on different edge servers, data needs to be transmitted through links of communication, and $y(i,j) = 1$. On the contrary, if two dependent microservices are deployed on the same edge server, $y(i,j) = 0$, which means that server-to-server data transfers are seamless and result in $d_l(e^h_{m_i \to m_j}) = 0$. And the $r^h_{m_i \to m_j}$ is data flow size between microservices m_i^h and m_j^h, $b_{(v_k,v_q)}$ is communication capability of link $l_{(v_k,v_q)}$.

3.3 Problem Formulation

In this paper, we focus on finding an efficient strategy that minimizes the makespan of services in set $\mathbf{S}$ under the constraint, which is determined by the part with the longest completion time. We use $f(m_j^h)$ to denote the completion time of microservice m_j^h, which has a predecessor m_i^h, i.e., $m_i^h \to m_j^h$. It depends on the completion time $f(m_i^h)$ of the predecessor m_i^h, the computation time $d_c(m_j^h)$ of m_j^h, and the communication time $d_l(e^h_{m_i \to m_j})$ for data transferred from predecessor m_i^h to m_j^h. However, it is worth noting that there may be multiple predecessor microservices in a service with complex dependencies. Thus, the value of $f(m_j^h)$ is determined by the path of precedence with the maximum completion time. It is calculated as:

$$f(m_j^h) = \max_{\forall i,j}\{f(m_i^h) + d_c(m_j^h) + d_l(e^h_{m_i \to m_j})\}. \tag{3}$$

Here, we define T_h as the makespan of service S_h, which depends on the maximum completion time of all microservices in S_h, where

$$T_h = \max_{\forall j, m_j^h \in S_h}\{f(m_j^h)\}, \tag{4}$$

and the makespan of services in set $\mathbf{S}$ depends on the maximum completion time of all services S_h, which is given by

$$\mathbf{T} = \max_{S_h \in S}\{T_h\}. \tag{5}$$

Therefore, the problem formulation is shown as follows:

$$\mathbf{P1}: \text{minimize} \quad \mathbf{T} \tag{6}$$

$$\text{s.t.} \sum\nolimits_{k=1}^{|V|} x(i,k) = 1, \quad \forall i \tag{7}$$

$$\sum\nolimits_{i=1}^{|M_h|} x(i,k) \le \phi_{(v_k)}, \quad \forall k \tag{8}$$

$$b_{(v_k,v_q)} \le \tau \tag{9}$$

$$x(i,k) \in \{0,1\}, \quad y(i,j) \in \{0,1\}, \quad \forall i, \forall j, \forall k. \tag{10}$$

P1 is the objective function that minimizes the makespan of services, and equations (6) to (10) are the constraints. Equation (7) signifies that each microservice can only be allocated to a single edge server. Equation (8) states that the number of microservices processed on an edge server cannot exceed its storage capacity. Equation (9) represents the constraint imposed by the communication bandwidth, where τ is the threshold determined by the microservices and servers. The proof of the bound is provided in Lemma 1 in the appendix. Equation (10) specifies the decision of microservice m_i^h that whether deployed on edge server v_k, where $x(i,k) \in \{0,1\}$, and the status that whether m_i^h and m_j^h are co-located on edge server v_k, where $y(i,j) \in \{0,1\}$.

To address the intricacies of microservice deployment in multi-access edge computing, we define the Optimal Microservice Deployment with Dependencies (OMDD) problem as follows.

Definition 1 (Optimal Microservice Deployment with Dependencies (OMDD) problem). Given the distribution of microservice **S** and the edge network **G**, the OMDD problem comprises how to find a strategy for microservices in **S** to minimize **P1** under the constraints (7)-(10).

The OMDD problem is NP-hard, as demonstrated in the detailed proof provided in the appendix.

4 Algorithm Design

4.1 Enhanced Graph Construction

This subsection addresses the simultaneous deployment requests of multiple services. A novel enhanced graph construction method is introduced to calculate the makespan, as specified in equation (5), facilitating parallel deployment across the service set **S**. The enhanced graph, denoted as $\hat{\mathbb{I}}$, incorporates a virtual source m_s and a virtual destination m_d, effectively interlinking all services involved. This configuration serves to streamline the deployment process by simplifying the connection of service components within the network.We suppose that the required processing capacities of m_s and m_d are all 0, where $q_{m_s} = 0$ and $q_{m_d} = 0$. To be precise, we construct $H_h = \{m^h_\omega|_{m^h_\omega \in M_h}\}$ as the set of starting nodes in service S_h, and $\mathbf{H} = \{H_h\}$ represents the set of starting nodes of all services. Then we add directed edges $e_{m_s \to m^h_\omega}$ which connect the virtual source m_s and all starting nodes in **H** of all services $\forall S_h \in$ **S**. Then we construct $D_h = \{m^h_\varpi|_{m^h_\varpi \in M_h}\}$ as the set of ending nodes in service S_h, and $\mathbf{D} = \{D_h\}$ represents the set of ending nodes of all services. We add directed edges $e_{m^h_\varpi \to m_d}$ connecting all ending nodes m_ϖ in **D** to the virtual destination m_d. We then give values to the directed edges which represent the required data flow size, where $r_{m_x \to m^h_\omega} = 0$ and $r_{m^h_\varpi \to m_d} = 0$. Since the newly added nodes and edges do not alter service dependencies or affect the completion times of microservices, the makespan of the enhanced graph $\hat{\mathbb{I}}$ is equivalent to the set of services **S**.

4.2 Scenario 1: OMDD with no storage constraint

This study initially examines a scenario designed to minimize **P1** under conditions where storage constraints are absent. This corresponds to relaxing the conditions on $\phi_{(v_k)}$, as defined in equation (8), such that $\phi_{v_k} \geq \sum_{h=1}^{|S|} |M_h|$ for each server k and for all services h. This scenario is particularly relevant in environments where edge servers possess ample storage capabilities but face limitations due to restricted network bandwidth, potentially due to network congestion or inadequate infrastructure. Consequently, the optimization challenge shifts to balancing computational and communication resources effectively, which is formulated as follows.

$$\mathbf{P2}: \text{minimize} \quad \mathbf{T} \tag{11}$$

$$\text{s.t.}(7), (9) - (10) \tag{12}$$

$$\sum\nolimits_{i=1}^{n} x(i,k) \leq \phi_{(v_k)}, \phi_{v_k} \geq \sum\nolimits_{h=1}^{|S|} |M_h|, \quad \forall k \tag{13}$$

Based on the interaction, we propose a greedy-based solution of Algorithm 1, which has been proven to be optimal in Theorem 1. We use the enhanced graph $\hat{\mathbb{I}}$, and the edge network **G** as the input. The output is the deployment strategy **X** of microservices and the makespan **T**. First, for each server in the set V, we calculate the sum of ϕ_{v_k} in lines 1 to 2. Then we calculate the total number of microservices in lines 3 to 4. In line 5, we determine whether the total storage resources sum_ϕ can accommodate all microservices sum_m. If the edge network **G** can accommodate all microservices, where $sum_\phi \geq sum_m$, we sort V in descending order by the computing capacity $c_{(v_k)}$ of the server to find the server with the highest computing capacity in line 6. In lines 7 to 10, we then begin with the provision of the microservices.

Algorithm 1 OMDD with unlimited-storage (OMDD-US)

```
Require: $\hat{\mathbb{I}}$, G.
Ensure: X, T.
 1: for $v_k \leftarrow 1$ to $|V|$ do
 2:     $sum_\phi \leftarrow sum_\phi + \phi_{v_k}$;
 3: for $m_i \leftarrow 1$ to $\hat{\mathbb{I}}$ do
 4:     $sum_m \leftarrow sum_m + 1$;
 5: if $sum_\phi \geq sum_m$ then
 6:     $V \leftarrow$ Update with order by $v_k = \arg\max\{c_{(v_k)}\}$;
 7:     for each $m_i$ in $\hat{\mathbb{I}}$ do
 8:         Place microservice $m_i$ on edge server $v_0$.
 9:         Update the deployment list X.
10:         $\mathbf{T} \leftarrow \mathbf{T} + q_{m_i}/c_{(v_1)}$
11: Return X, T.
```

For each m_i in $\hat{\mathbb{I}}$, we deploy the m_i in line 8 to the server v_0. We then update the deployment list **X** and calculate the **T** in lines 9 to 10. Finally, we return the deployment strategy **X** of microservices and the makespan **T** in line 11.

Theorem 1. OMDD-US is an optimal solution for **P2**.

Proof: We prove this theorem by contradiction. We assume that the completion time for placing microservices separately denoted as $\mathbf{T}_s$ is lower than that of merging them as a whole $\mathbf{T}_t$, i.e., $\mathbf{T}_s < \mathbf{T}_t$. Suppose there are two edge servers v_1 and v_2, where $c_{(v_1)} \geq c_{(v_2)}$. We first consider the simplest case of a chain-like microservice graph with only two microservices m_x and m_y. When microservices are deployed separately, m_x is located on edge server v_1, and m_y is located on edge server v_2. Given this, the completion time $\mathbf{T}_s$ for separate deployment is: $\mathbf{T}_s = q_{m_x}/c_{(v_1)} + q_{m_y}/c_{(v_2)} + r_{m_x \to m_y}/b_{v_1,v_2}$. Then, we consider the case consisting of placing all microservices on the same edge server v_1, which has $\mathbf{T}_t = q_{m_x}/c_{(v_1)} + q_{m_y}/c_{(v_1)}$. Since we suppose $c_{(v_1)} \geq c_{(v_2)}$, it follows that $q_{m_y}/c_{(v_2)} \geq q_{m_y}/c_{(v_1)}$. We calculate the difference between $\mathbf{T}_s$ and $\mathbf{T}_t$, where $\mathbf{T}_s - \mathbf{T}_t = (q_{m_y}/c_{(v_2)} - q_{m_y}/c_{(v_1)})) + r_{m_x \to m_y}/b_{v_1,v_2}$. Due to the fact that $r_{m_x \to m_y}/b_{v_1,v_2} \geq 0$, we have $\mathbf{T}_s - \mathbf{T}_t \geq 0$, i.e., $\mathbf{T}_s \geq \mathbf{T}_t$, which contradicts our assumption. Then, we consider a more realistic case of a DAG-based microservice graph with only complex dependencies. We assume that there exists a path $m_x \to m_y \to m_z$ with the maximum required processing capability of services **S**, and the makespan $\mathbf{T}_t = q_{m_x}/c_{(v_1)} + q_{m_y}/c_{(v_1)} + q_{m_z}/c_{(v_1)}$ for the case that placing all microservices on the same edge server v_1. Assume that these microservices are deployed separately, where m_x is deployed on the server v_1, and m_y and m_z are deployed on the server v_2. The completion time $\mathbf{T}_s$ will be $\mathbf{T}_s = q_{m_x}/c_{(v_1)} + q_{m_y}/c_{(v_2)} + q_{m_y}/c_{(v_2)} + r_{m_x \to m_y}/b_{v_1,v_2}$. Additionally, we calculate the difference between $\mathbf{T}_s$ and $\mathbf{T}_t$, where $\mathbf{T}_s - \mathbf{T}_t = ((q_{m_y} + q_{m_z})/c_{(v_2)} - (q_{m_y} + q_{m_z})/c_{(v_1)}) + r_{m_x \to m_y}/b_{v_1,v_2}$. Since we suppose $c_{(v_1)} \geq c_{(v_2)}$, it follows that $(q_{m_y} + q_{m_z})/c_{(v_2)} \geq (q_{m_y} + q_{m_z})/c_{(v_1)}$. Furthermore, due to the fact that $r_{m_x \to m_y}/b_{v_1,v_2} \geq 0$, we are able to deduce that $\mathbf{T}_s - \mathbf{T}_t \geq 0$, i.e., $\mathbf{T}_s \geq \mathbf{T}_t$, which contradicts our assumption. Therefore, we can obtain that OMDD-US can minimize **P1** under the constraints (7), (9)-(11). ■

Algorithm 2 OMDD with unlimited-bandwidth (OMDD-UB)

```
Require: $\hat{\mathbb{I}}$, G.
Ensure: X, T.
 1: Same as Algorithm 1 in lines 1-4;
 2: if $sum_\phi \geq sum_m$ then
 3:     $V \leftarrow$ Update with order by $v_k = \arg\max\{c_{(v_k)}\}$;
 4:     $\hat{\mathbb{I}} \leftarrow$ Update with order by $\hat{\mathbb{I}} = \arg\max\{q_{m_i}\}$;
 5:     for each $m_i$ in $\hat{\mathbb{I}}$ do
 6:         for $v_k \in V$ do
 7:             if $\phi_{v_k} > 0$ then
 8:                 Place $m_i$ on edge server $v_k$;
 9:                 $\phi_{v_k} = \phi_{v_k} - 1$;
10:                 Update the deployment list X;
11:                 $\mathbf{T} \leftarrow \mathbf{T} + q_{m_i}/c_{(v_i)}$;
12:             else
13:                 Update $V = V/v_k$ and go back to line 5;
14: Return X, T
```

4.3 Scenario 2: OMDD with no communication constraint

This subsection explores a scenario aimed at minimizing **P1**, characterized by the absence of bandwidth constraints and the removal of equation (9), stipulating $b_{(v_k,v_q)} \geq \tau$. This setup typifies environments where bandwidth is plentiful while storage resources are constrained, often due to financial limitations or spatial restrictions. Such conditions prevail in edge computing contexts where rapid network connections are ubiquitous, but storage capacities remain limited. This scenario is delineated as follows.

$$\mathbf{P3}: \text{minimize} \quad \mathbf{T} \tag{14}$$
$$\text{s.t.}(7)-(8),(10) \tag{15}$$
$$b_{(v_k,v_q)} \geq \tau. \tag{16}$$

Thus, the initial optimization problem has changed to become how to balance the computing and storage resources. On the basis of the interaction, we propose a novel method of Algorithm 2, which we prove to be optimal. We use the enhanced graph $\hat{\mathbb{I}}$, and the edge cloud environment **G** as the input. The output is the microservices deployment strategy **X** and the makespan **T**. First, we determine whether the total storage of all edge servers can accommodate all microservices in the enhanced graph $\hat{\mathbb{I}}$ in the same way as Algirhtm 1 in lines 1 to 4. If **G** can accommodate all microservices, where $sum_\phi \geq sum_m$, we sort servers in descending order of computing capability $c_{(v_k)}$ in line 3, and sort microservices in descending order of required computing capability q_{m_i} in line 4. Then, we deploy the microservices sequentially in lines 5 to 13. For each v_k in the sorted V, we start the deployment by checking whether v_k has available storage resources in line 7. If the current remaining resources $\phi_{v_k} > 0$, we place m_i^h on the server v_k in line 8 and update ϕ_{v_k}. We then update the deployment list **X** and calculate the makespan **T** in lines 10 to 11. If v_k has no further storage resources, we update set V by removing v_k, i.e., $V = \{V/v_k\}$, and then we go back to line 5. Finally, line 14 returns the deployment strategy **X** and the makespan **T**.

Theorem 2. OMDD-UB is an optimal solution to **P1** under the constraints (7)-(8), (10).

Proof: This theorem is demonstrated through a proof by contradiction. Consider the hypothesis that deploying microservices with more substantial computational demands on a server with lower computational capacity results in a shorter completion time than deploying them on a server with greater computational capacity. Let us denote the makespan on the lower-capacity server as $\mathbf{T}'$ and on the higher-capacity server as $\mathbf{T}''$, with the assumption $\mathbf{T}' < \mathbf{T}''$. Suppose there are two edge servers v_1 and v_2, where $c_{(v_1)} < c_{(v_2)}$. We consider the case of service with two microservices m_x and m_y, where $q_{m_y} > q_{m_x}$. When placing a microservice with higher required computational resources on a server with lower computing capability, m_x is located on edge server v_1, and m_y on edge server v_2.

The completion time $\mathbf{T}'$ for separate deployment is $\mathbf{T}' = q_{m_x}/c_{(v_1)} + q_{m_y}/c_{(v_2)} + r_{m_x\to m_y}/b_{v_1,v_2}$. Since the bandwidth is unlimited, the communication time can be neglected. Therefore, $\mathbf{T}' = q_{m_x}/c_{(v_1)} + q_{m_y}/c_{(v_2)}$. Then consider the case where microservices with higher required computational capacity are placed on a server with higher computing capacity, which has $\mathbf{T}'' = q_{m_x}/c_{(v_2)} + q_{m_y}/c_{(v_1)}$. We calculate the difference between $\mathbf{T}'$ and $\mathbf{T}''$, where $\mathbf{T}' - \mathbf{T}'' = (q_{m_x}/c_{(v_1)}) + (q_{m_y}/c_{(v_2)}) - (q_{m_x}/c_{(v_2)}) - (q_{m_y}/c_{(v_1)}) = (c_{(v_2)}q_{m_x} + c_{(v_1)}q_{m_y} - c_{(v_1)}q_{m_x} - c_{(v_2)}q_{m_y})c_{(v_1)}c_{(v_2)} = (c_{(v_2)} - c_{(v_1)})(q_{m_x} - q_{m_y})/c_{(v_1)}c_{(v_2)}$. Since we suppose $c_{(v_2)} > c_{(v_1)}$ and $q_{m_x} > q_{m_y}$, we have $\mathbf{T}' - \mathbf{T}'' > 0$, i.e., $\mathbf{T}' > \mathbf{T}''$, which contradicts our assumption. Therefore, we obtain that OMDD-UB minimizes **P1** under the constraints (7)-(8), (10). ■

4.4 Scenario 3: OMDD with constraints (7)-(10)

In this subsection, we investigate a more complicated scenario with all resource constraints on storage, computation, and communication, representing OMDD with constraints (7)-(10). Each of these physical resource constraints could potentially serve as a bottleneck for microservices with dependencies in the extended graph $\hat{\mathbb{I}}$. In order to solve the problem, we present a preliminary method based on embedding the main path and borrowing ideas from the critical path approach to optimize the makespan by balancing multiple resource constraints in microservice provisioning. However, the completion time of each microservice varies depending on the deployment strategy, so it is impossible to obtain the critical path directly without determining the locations. In order to reduce the complexity of the problem, we introduce a novel definition of the main path for microservices as follows.

Definition 2 (main path). The main path p_i refers to the path with the maximum weight $\arg\max\{w_{(p_i)}\}$ of $\hat{\mathbb{I}}$.

Here, we use $P = \{p_i\}$ to denote the set of all simple paths of $\hat{\mathbb{I}}$, and we use p_i to denote a simple path consisting of a series of microservices denoted as $S^{(p_i)} = \{m_1, m_2, \ldots, m_n\}$. We treat the computation demand q_{m_i} of each microservice as the node weight and the data flow size $r_{m_i\to m_j}$ between microservices as the edge weight. Thus, for any path p_i, the weight $w_{(p_i)}$ is calculated as:

$$w_{(p_i)} = \sum\nolimits_{i=1}^{n} q_{m_i} + \sum\nolimits_{i=1}^{n-1} r_{m_i\to m_{i+1}}, \quad m_i \in S^{(p_i)}. \tag{17}$$

We prove that the main path is equal to the critical path if the computational capacities of the servers are equal and the bandwidths between the servers are equal.

Theorem 3. The main path will become the critical path when server computation capacities and inter-server bandwidths are equal, where $\{c_{v_i} = b_{(v_k,v_q)} | \forall v_i \in V, \forall l_{(v_k,v_q)} \in L\}$.

Proof: We prove this theorem by contradiction. First, we assume the existence of a path p_i composed of a series of microservices denoted as $\{m_1, m_2, \dots, m_n\}$. The total weight of microservices in p_i is given by $w_{(p_i)} = \sum_{i=1}^{n} q_{(m_i)} + \sum_{i=1}^{n-1} r_{m_i \to m_{i+1}}$. Then, we suppose there exists another path p_a with additional microservices, composed of $\{m_a, m_b, \dots, m_m\}$. The total weight of microservices in p_a is $w_{(p_a)} = \sum_{j=a}^{m} q_{(m_j)} + \sum_{j=1}^{m-1} r_{m_j \to m_{j+1}}$, with $w_{(p_a)} < w_{(p_i)}$. We assuming p_a is the critical path, i.e. $t_{(p_a)} > t_{(p_i)}$. Due to limited storage resources, we assume that microservices on path p_i cannot be placed on the same edge server. As servers have equal computation capacity and inter-server bandwidth, we have $t_{(p_i)} = \sum_{i=1}^{n} q_{(m_i)}/c_{(v_k)} + \sum_{i=1}^{n-1} r_{m_i \to m_{i_1}}/b_{(v_k,v_q)}$ and $t_{(p_a)} = \sum_{j=a}^{m} q_{(m_j)}/c_{(v_k)} + \sum_{j=1}^{m-1} r_{m_j \to m_{j+1}}/b_{(v_k,v_q)}$. Simplifying, $t_{(p_a)} - t_{(p_i)} = \sum_{i=1}^{n} q_{(m_i)}/c_{(v_k)} + \sum_{i=1}^{n-1} r_{m_i \to m_{i+1}}/b_{(v_k,v_q)} - \sum_{j=a}^{m} q_{(m_j)}/c_{(v_k)} - \sum_{j=1}^{m-1} r_{m_j \to m_{j+1}}/b_{(v_k,v_q)}$. Since we suppose $c_{(v_k)} = b_{(v_k,v_q)}$, it follows that $t_{(p_a)} - t_{(p_i)} = w_{(p_a)} - w_{(p_i)} < 0$. So, we have $t_{(p_a)} - t_{(p_i)} < 0$, i.e., $t_{(p_a)} < t_{(p_i)}$, which contradicts our assumption. Therefore, the main path will become the critical path when server computation capacities and inter-server bandwidths are equal. ■

Theorem 4 brings us to the conclusion that the main path occasionally becomes the bottleneck of the overall makespan, so it then also becomes the critical path. So if the critical path cannot be found because the recently arrived microservices have not been deployed, we can reduce the overall makespan by searching for the main path. We use $t_{(p_i)}$ to represent the completion time of path p_i. The formula to calculate $t_{(p_i)}$ is as follows:

$$t_{(p_i)} = d_c(1) + \sum_{j=2}^{n} \left(d_c(j) + d_l(j-1, j) \right). \qquad (18)$$

The makespan $\mathbf{T}$ can be transformed into the maximum value of all path completion times $t_{(p_i)}$, where

$$\mathbf{T} = \max\{f_i(j)\} \equiv \max\left(t_{(p_i)}\right). \qquad (19)$$

On the basis of this, we propose a preliminary deployment strategy consisting of a main path embedding method in Algorithm 3. We use the enhanced graph $\hat{\mathbb{I}}$ and the edge network $\mathbf{G}$ as the input. The output is the preliminary microservices deployment strategy $\mathbf{X}_0$ and the makespan $\mathbf{T}_0$. First, we determine whether the total storage of all edge servers can accommodate all microservices in an enhanced graph $\hat{\mathbb{I}}$ in the same way as Algorithm 1 does in lines 1 to 4.In order to optimize both computation and transferring time, we propose a new definition of the preferred server as follows.

Definition 3 (preferred server). Let v° indicate the preferred server of V, where $v^\circ = \max_{\xi(v_k)}\{v_k|_{v_k \in V}\}$. Here, $\xi(v_k)$ is the priority value of v_k with the sum of the computing capacity and the maximum bandwidth connected in $\mathbf{G}$.

Here, we define function $\xi(v_k)$ to calculate the priority value in order to find the preferred server in lines 3 to 4, aiming to obtain a server with better processing capacity and bandwidth, and jointly optimize the processing time and transferring time. For each v_k in set V, we calculate the sum of computing capability $c_{(v_k)}$ and the maximum bandwidth connected by v_k. We choose the server with the largest $\xi(v_k)$ value as a preferred server in line 5. We use the depth-first search to find all simple paths of set $P = \{p_i\}$ in line 6. Then, we deploy the microservices in lines 7 to 17. Lines 7 to 8 use equation (11) to calculate the weight of each path, and then find the main path p_i. Then, we need to determine whether the v° storage is sufficient to accommodate all the microservices of p_i, where $v^\circ \le p_i$. We place all microservices on the preferred server when the v° storage is sufficient in line 10. Otherwise, we divide the path based on the maximum cut c_{p_i} in line 12 which is defined as follows.

Definition 4 (maximum cut). Let c_{p_i} indicate the maximum cut of path p_i in $\hat{\mathbb{I}}$ which constructs by $|\phi_{v^\circ}|$ microservices with the largest weights combination.

Here, $|\phi_{v^\circ}|$ represents the server storage of the preferred server v°. Then, we deploy the maximum cut c_{p_i} on v° in

Algorithm 3 OMDD based on Main Path Embedding (OMDD-MPE)

```
Require: $\hat{\mathbb{I}}$, $\mathbf{G}$.
Ensure: $\mathbf{X}_0$, $\mathbf{T}_0$.
 1: Same as Algorithm 1 in lines 1-4;
 2: if $sum_\phi \ge sum_m$ then
 3:     for $v_k \leftarrow 1$ to $|V|$ do
 4:         $\xi(v_k) = c_{(v_k)} + \max\{b_{(v_k,v_q)}\}$;
 5:     Choose the preferred server $v^\circ$;
 6:     Construct set $P$ by depth-first search;
 7:     for $p_i$ in $P$ do
 8:         Find the main path $p_i$ by Equation (11);
 9:         if $\phi_{v^\circ} > |p_i|$ then
10:             Place $p_i$ on $v^\circ$.
11:         else
12:             Construct $c_{p_i}$ with $|\phi_{v^\circ}|$ microservices;
13:             Place $c_{p_i}$ on $v^\circ$;
14:             Update $p_i = p_i - c_{p_i}$ and go back to line 9;
15:         Update the deployment list $\mathbf{X}_0$;
16:         Update $t_{p_i}$ by equation (12);
17:     Update $\mathbf{T}_0$ by equation (13);
18: Return $\mathbf{X}_0$, $\mathbf{T}_0$;
```

Algorithm 4 OMDD based on Improved Simulated Annealing (OMDD-ISA)

```
Require: $\hat{\mathbb{I}}$, $\mathbf{G}$, $\mathbf{X}_0$, $\mathbf{T}_0$.
Ensure: $\mathbf{X}$, $\mathbf{T}$.
 1: Initialize $\mathbf{X} \leftarrow \mathbf{X}_0$, $\mathbf{T} \leftarrow \mathbf{T}_0$, $r$, $t$, $k$        ▷ $r$ controls
    the speed of cooling, $t$ is the temperature of the system,
    $k$ is the number of iterations.
 2: for $i \leftarrow 1$ to $k$ do
 3:     Exchange the deployment positions of microservices
    in $\mathbf{X}$ and generate an updating deployment $\hat{\mathbf{X}}$;
 4:     Calculate makespan $\hat{\mathbf{T}}$ of $\hat{\mathbf{X}}$.
 5:     if $\hat{\mathbf{T}} < \mathbf{T}$ then
 6:         $\mathbf{X} \leftarrow \hat{\mathbf{X}}$;
 7:     else
 8:         Calculate $\rho = e^{(\mathbf{T}-\hat{\mathbf{T}})/t}$;
 9:         if $\varepsilon < \rho$ then
10:             $\mathbf{X} \leftarrow \hat{\mathbf{X}}$;
11:     $t \leftarrow t \times r$;
12: Return $\mathbf{X}$, $\mathbf{T}$
```

Algorithm 5 Updating based on Simulated Annealing with Critical Path (U-SAC)

```
Require: $S_r$, $\mathbf{X}$, $\hat{\mathbb{I}}$.
Ensure: $\mathbf{X}_U$, $\mathbf{T}_U$.
 1: Service $S_r$ requests resources release;
 2: Release the resources occupied by $S_r$;
 3: Update enhanced graph $\hat{\mathbb{I}}$;
 4: Calculate critical path $p_i'$ with the maximum compilation time $\arg\max\{t_{(p_i)}\}$ of $\hat{\mathbb{I}}$.
 5: Record servers with resource changes by constructing set $V_u = \{\hat{v}_k|_{\{\phi_{(\hat{v}_k)} \neq \phi_{(v_k)}\}}\}$;
 6: Record servers of critical path by constructing set $V_o = \{\tilde{v}_k|_{m_i^h \in p_i'}\}$;
 7: if any $\hat{v}_k \in V_u$ for $\tilde{v}_k \in V_o$ then
 8:     Same as Algorithm 4 in lines 1-11;
 9: else
10:     Not updated;
11:     Return $\mathbf{X}_U$, $\mathbf{T}$.
```

line 13. We update path p_i with $p_i = p_i - c_{p_i}$ to continue to complete the deployment of the remaining services and go back to line 9. After that, we update the deployment list $\mathbf{X}_0$, and we use equations (12) and (13) to calculate the makespan $\mathbf{T}_0$ in lines 15 to 17. Finally, the microservices deployment strategy $\mathbf{X}_0$ and the makespan $\mathbf{T}_0$ are returned in line 18. Although we proved in Theorem 4 that the main path can become the critical path under certain conditions, the actual scenario might be more complicated. This means that the main path is not always identical to the critical path. Therefore, Algorithm 3 may not always provide the optimal solution and may lead to suboptimal deployment results in certain cases. To further improve the quality of our solution, we have therefore taken additional measures to optimize the performance of Algorithm 3.

We utilize the preliminary deployment solution $\mathbf{X}_0$ obtained from Algorithm 3 as a starting point and introduce a novel strategy based on the improved simulated annealing algorithm for iterative optimization. However, due to the limitation of server storage resources, the traditional simulated annealing algorithm might exceed the capacity constraint when searching for neighbor solutions. To address this issue, we have refined the algorithm. The specific steps are presented in Algorithm 4. In line 1, we take the preliminary deployment strategy $\mathbf{X}_0$ and makespan $\mathbf{T}_0$ obtained in Algorithm 3 as the required values of Algorithm 4, and we set the values of hyperparameters. Lines 3 to 4 randomly select the deployment positions of two microservices from the current solution for exchanging and generating an updating solution. We then determine an updated makespan $\hat{\mathbf{T}}$. Lines 5 to 10 determine whether the new solution is accepted based on the Metropolis criterion. If the makespan of the updating strategy $\hat{\mathbf{T}}$ is lower than the preliminary makespan $\mathbf{T}$, we accept the updating strategy $\hat{\mathbf{X}}$ in line 6. Otherwise, we calculate the probability ρ of the new strategy in line 8. After that, we use ε to represent the judging condition for accepting ρ which is a random probability between 0 and 1, where $\varepsilon \in [0, 1]$. The new approach is acceptable if the probability ρ is above ε. Otherwise, reserve strategy $\mathbf{X}$. After that, we cool down the t at a rate r range from [0.9,1) in line 11. This optimization strategy enables us to progressively modify the distribution of microservices depending on the preliminary solution to better respond to the practical environment and resource constraints. Through multiple iterations, the simulated annealing algorithm progressively converges towards improved solutions, thereby enhancing the quality and effectiveness of the deployment strategy.

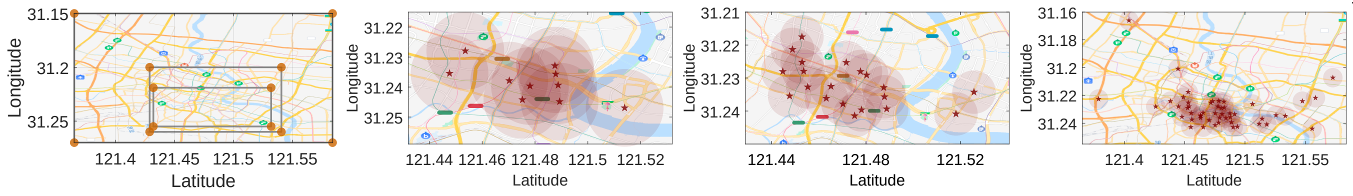
Theorem 4. The complexity of Algorithm 4 is $O(\kappa \times |V| \cdot |V-1|!)$, where κ is the maximum number of iterations.

Proof: In Algorithm 4, each iteration involves a series of operations, including generating new solutions, calculating the objective function value, and deciding whether to accept the new solution. The time complexity of these operations is fixed and denoted as $O(1)$. In addition, since the number of maximum iterations κ determines the runtime of the algorithm, the overall time complexity of the algorithm is directly proportional to the maximum number of iterations κ, i.e., $O(\kappa)$. During each iteration, the algorithm performs calculations to determine whether new solutions are accepted. We need to traverse all possible paths and perform calculations from the source to the destination. Since the number of paths depends on the topology of the graph, in the worst case it can reach the factorial size of the number of nodes, which is denoted by $O(|V| \cdot |V-1|!)$. Thus, the time complexity of Algorithm 4 can be expressed as $O(\kappa \times |V| \cdot |V-1|!)$, where κ is the maximum number of iterations, and $|V|$ is the number of edge servers in the enhanced graph $\mathbf{G}$. ■

4.5 Scenario 4: OMDD with Updating

In this subsection, we explore a more realistic and continuous scenario where all services are deployed under constraints (7)-(10) and dynamically release resources after their operation cycle is completed. Our focus is on real-time resource updating during the dynamic operation of microservices to ensure that resources are optimally utilized and available for new services when needed. Therefore, we design a flexible update algorithm that responds promptly to release requests and adjusts the locations of microservices to reduce the overall makespan by migrating and merging microservices.

The specific steps are outlined in Algorithm 5. We use the enhanced graph $\hat{\mathbb{I}}$, the edge environment $\mathbf{G}$, and the service S_r that released resources as the input. The output is the updating strategy $\mathbf{X}_u$ and the makespan $\mathbf{T}$ of the microservices. When the service S_r requests the release of resources, we first release the resources occupied by S_r and then update the extended graph $\hat{\mathbb{I}}$ in lines 1 to 3. After that, we record servers with resource changes in set V_u in line 4. For the updated $\hat{\mathbb{I}}$, we calculate the critical path p_i' with the maximum compilation time by equation (18), and record set of servers V_o of p_i'. If a server in V_u is also in V_o, which means that there are excess resources on the server where the critical path is located, we can reduce the makespan by merging the critical paths. The proof is shown in Theorem 6. Then we use the improved algorithm for simulated annealing, which corresponds to lines 1 to 11 of Algorithm 4. Specifically, we randomly select two microservices on servers in the set V_o and perform a location exchange for an updating solution. On that basis, we determine an updating makespan $\hat{\mathbf{T}}$ and determine whether the new solution is accepted based on the Metropolis criterion in line 8. If there is no change in resources on the server where the critical



(a) Server Distribution. (b) Server locations (# of 10). (c) Server locations (# of 20). (d) Server locations (# of 50).

Fig. 2. Locations of edge servers.

path resides, microservice updates will not be carried out. Finally, the deployment strategy $\mathbf{X}$ and the makespan $\mathbf{T}$ are returned in line 11.

Theorem 5. Updating the locations of microservices that construct the critical path is a necessary and sufficient condition for reducing makespan $\mathbf{T}$.

Proof: First, we prove that updating the locations of microservices that construct the critical path is a necessary condition for reducing the makespan. We use proof by contradiction to demonstrate this. We assume that updating the locations of microservices constructs a simple path that non-critical path will reduce the makespan. We define the updating makespan as $\mathbf{T}'$, i.e., $\mathbf{T}' < \mathbf{T}$. We assume that there are two paths, p_1 and p_2, in the augmented graph $\hat{\mathbb{I}}$, where $t_{(p_1)} > t_{(p_2)}$. According to equation (19), $\mathbf{T} = \max(t_{(p_i)})$, so $\mathbf{T} = t_{(p_1)}$. Suppose we update the locations of microservices on path p_2, resulting in an updating time $t'_{(p_2)}$ where $t'_{(p_2)} < t_{(p_2)}$. Since $t_{(p_2)} < t_{(p_1)}$, we have $t'_{(p_2)} < t_{(p_1)}$. Due to equation (19), $\mathbf{T}' = \max(t_{(p_i)})$, and the updating makespan is $\mathbf{T}' = t_{(p_1)}$. $\mathbf{T}' = \mathbf{T}$, which contradicts our assumption. Therefore, we can conclude that updating the locations of microservices that construct the critical path is a necessary condition for reducing the makespan.

Then, we prove that updating the locations of microservices that construct the critical path is a sufficient condition, i.e., when the makespan is reduced, the locations of microservices on the critical path have been reduced. We use proof by contradiction to demonstrate this. We assume that when the makespan is reduced, the reduced path is a simple path which is the non-critical path. Suppose there are multiple paths in the enhanced graph $\hat{\mathbb{I}}$ before updating, where p_1 has the longest finish time. According to equation (19), $\mathbf{T} = \max(t_{(p_i)})$, so $\mathbf{T} = t_{(p_1)}$. After updating $\mathbf{T}' < \mathbf{T}$, $\mathbf{T}' = \max(t_{(p_i)})$, the reduced critical path in the graph $\hat{\mathbb{I}}$ is $t_{(p_i)}$ and $t_{(p_i)} < t_{(p_1)}$. However, since $\mathbf{T} = \max(t_{(p_i)})$ and $t_{(p_i)} < t_{(p_1)}$, the reduced makespan should be $\mathbf{T}' = t_{(p_1)} = \mathbf{T}$. This is a contradiction, so our assumption is false. Therefore, we can conclude that updating the locations of microservices that construct the critical path is a sufficient condition for reducing the makespan. ■

5 Experiment

5.1 Basic Setting

We conducted extensive experiments to validate the effectiveness of our algorithms under various scenarios. All experiments were conducted using Python 3.7 on Windows 10 with an Intel(R) Xeon(R) Silver 4210R CPU @ 2.40GHz, NVIDIA RTX5000 GPU, and 32GB memory. We utilized a dataset obtained from China Telecom Shanghai Company [21], containing information about 3,233 base station locations and their corresponding user connections in June 2014. We randomly selected subsets of locations containing 10, 20, and 50 base stations, and each base station was equipped with a server, forming set V, as shown in Figure 2. Subsequently, we generate varying numbers of services under each scale, with each service abstracted as a DAG composed of 4-5 microservices. The total numbers of microservices for three different scale scenarios are 25, 50, and 120, respectively. We configure the required processing capacity and internal data flow for these microservices.

5.1.1 Scenarios

We made modifications to the edge network in accordance with various scenarios.

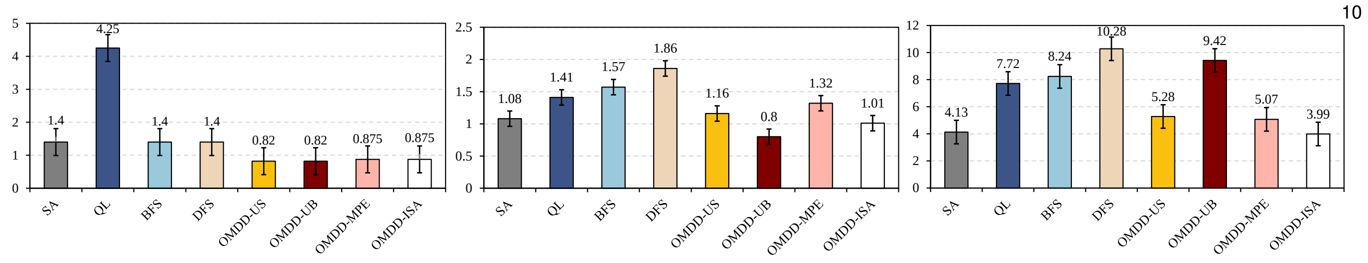
- **Scenario 1**: We set computing capacities of edge servers to range from $[5, 20]$ $gflops$, and storage resources range from $[120, 150]$ units. We set the inter-server bandwidth to range from $[20, 80]$ GB/s. Additionally, the required computing capacities of microservices range from $[1, 3]$ $gflops$, and the inter-microservice data flow sizes range from $[1, 80]$ GB/s.
- **Scenario 2**: We set computing capacities of servers to range from $[5, 20]$ $gflops$, and storage resources range from $[1, 5]$ units. We set the inter-server bandwidth to range from $[100, 500]$ GB/s. The setting of required computing capacities for microservices is the same as scenario 1, and the inter-microservice data flow sizes are changed to $[0.1, 0.2]$ GB/s.
- **Scenario 3 and Scenario 4**: We set storage resources to range from $[1, 5]$ units. The computing capacities, inter-server bandwidth ranges, required computing capacities, and inter-microservice data flow size ranges are similar to scenario 1.

5.1.2 Baselines

We introduce four baselines to compare with our proposed OMDD-ISA (ISA) algorithm for scenarios 1, 2, and 3.

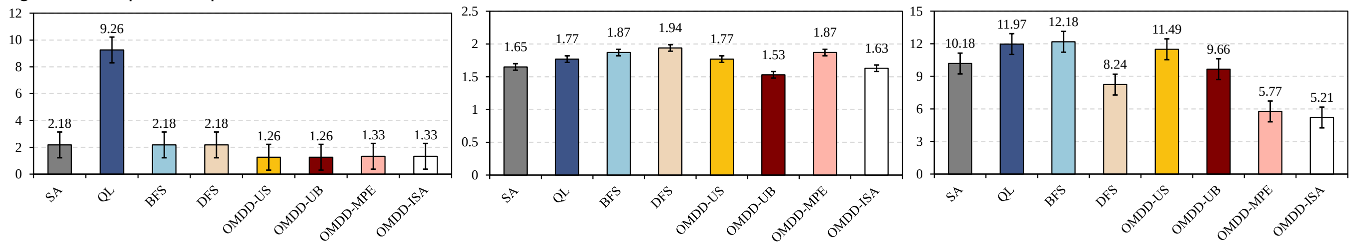
- **Simulated Annealing-only (SA)**: Traditional annealing algorithm, generating random initial values.
- **Q-Learning (QL)**: States are composed of the allocation status of a series of services. Each service can be assigned to different servers or remain unassigned. In the initial state, all services are in the cloud and the action space contains $\sum_{h=1}^{|S|} |M_h| \cdot |V|$ actions. Deployment decisions are guided by the Q table, with positive rewards for successful deployments and negative rewards for unsuccessful ones.
- **BFS**: Prioritizes microservices with minimal computational demands and no dependencies, deploying them sequentially to optimize resource utilization on available servers [10].
- **DFS**: Targets microservices within the largest data streams for early deployment, arranging them to minimize data transmission times between interdependent services [10].

For scenario 4, we introduced six baselines in comparison with our proposed U-SAC algorithm:



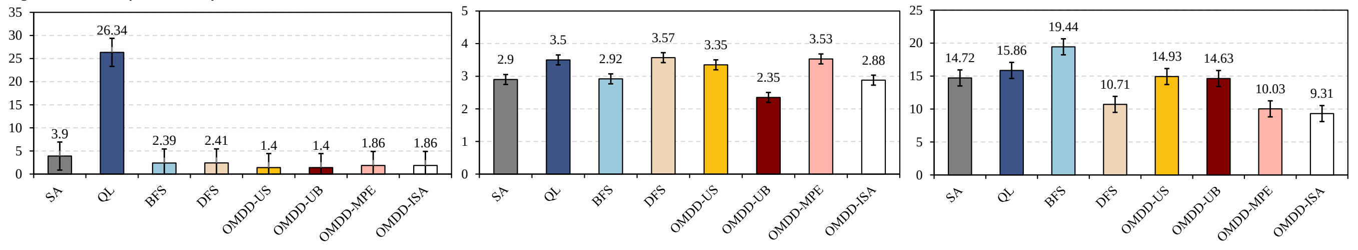
(a) Scenario 1. (b) Scenario 2. (c) Scenario 3.

Fig. 3. Makespan comparison with 10 servers and 25 microservices.



(a) Scenario 1. (b) Scenario 2. (c) Scenario 3.

Fig. 4. Makespan comparison with 20 servers and 50 microservices.



(a) Scenario 1. (b) Scenario 2. (c) Scenario 3.

Fig. 5. Makespan comparison with 50 servers and 120 microservices.

TABLE 2
The total makespan under real-world microservice benchmark.

methods		SA	QL	BFS	DFS	OMDD-US	OMDD-UB	OMDD-MPE	OMDD-ISA
makespans	Lakeside	5.14	6.65	5.94	7.9	6.84	7.3	5.45	4.01
	FTGO	4.73	5.64	8.52	7.84	5.3	5.84	5.59	3.77

- **ISA**: No updates are performed after the release of services (MODD-ISA).
- **Updating-Simulated Annealing with random path (U-SAR)**: Randomly select a non-critical path for a microservices route, determine the location of the server along that path, and check for resource changes. If any of these servers experience resource changes, apply simulated annealing to resolve.
- **Updating-by Capacity (U-CA)**: Identify servers with resource changes after service release, select the server with the most remaining storage resources as the target server, and merge the paths of microservices on the target server.
- **Updating-by Compute Power (U-CP)**: Identify servers with resource changes after the service release, select the server with the strongest computing capacity as the target server, and merge the paths of the microservices on the target server.
- **Updating-by Capacity with critical path (U-CAC)**: Locate the critical path, select the server with the maximum remaining storage resources among those on the critical path as the target server, and merge the critical path.
- **Updating-by Compute Power with critical path (U-CPC)**: Identify the critical path, select the server with the highest computing capability among those on the critical path as the target server, and merge the critical path.

5.2 Experiment Results

5.2.1 Microservice Deployment

We evaluated multiple algorithms across different experimental scenarios to compare their performance on microservice applications of varying scales and scenarios. As shown in Figures 3, 4, and 5, the x-axis lists all evaluated algorithms, while the y-axis represents the overall makespan, defined as the end-to-end execution time required to complete all microservices in the workflow. Based on these results, we draw the following conclusions:

(i). The OMDD-US method consistently achieves the lowest makespan in Scenario 1. As shown in Figures 3(a), 4(a), and 5(a), the makespan for the OMDD-US method is 0.82 seconds with 10 servers and 25 microservices, 1.26 seconds for 20 servers and 50 microservices, and 1.4 seconds for 50 servers and 120 microservices. This performance surpasses all other methods. Notably, the makespan of UB is consistently the same as that of the US because both methods use a greedy approach to place microservices on high-capacity servers. The difference is that UB sorts microservices based on $q_{m_i^h}$ before deployment, making it more complex in large-scale scenarios. We can see in Figures 3(a) and 4(a) that the makespan for DFS and BFS are consistent, while in Figure 5(a), there are slight differences. This is because when the server capacity is large enough, the order of microservice deployment can be ignored, but when the server capacity is insufficient to accommodate all microservices, the deployment order will affect the completion time. Meanwhile, MPE and ISA per-

TABLE 3
The released services are the ones that are on the critical path.

methods		**ISA**	**U-SAC**	**U-SAR**	**U-CA**	**U-CP**	**U-CAC**	**U-CAP**
makespans	$S_1(10)$	3.61	**3.26**	3.61	3.51	3.51	6.11	3.61
service (# of servers)	$S_3(20)$	4.81	**4.81**	4.81	4.81	4.81	4.81	4.81

form relatively modestly in Scenario 1, with slightly higher makespans compared to other methods. This is because MPE takes into account both computation and communication resources. In Scenario 1, where storage resources are unrestricted, placing all services on a single server does not generate communication time, making communication resources less critical. The uniform performance of MPE and ISA arises from the swap-based optimization strategy under capacity constraints when using simulated annealing, which is not applicable in the capacity-unlimited Scenario 1. Furthermore, the SA method exhibits relatively poorer performance in Scenario 1 due to its tendency to get trapped in a local optimum when storage resources are unlimited and the problem scale is substantial. Finally, the QL method shows relatively higher makespans across all problem scales in Scenario 1. This is attributed to the complexity of its state space, resulting in longer computation times and the inability to find the optimal solution within limited iterations.

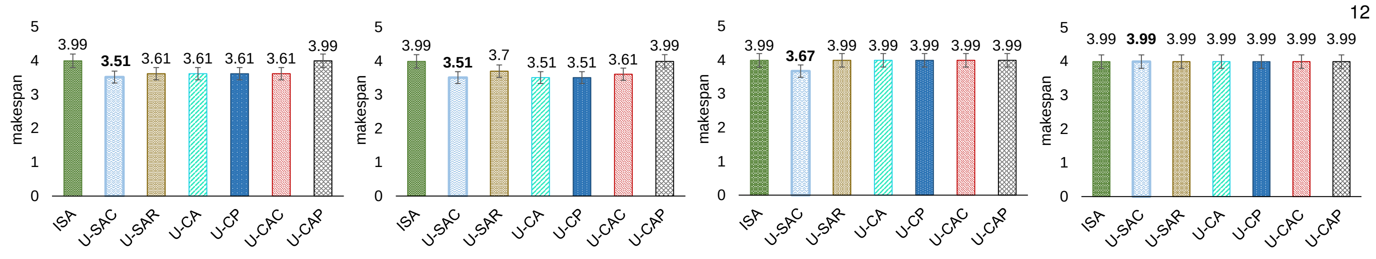
(ii). The OMDD-UB method consistently minimizes the makespan in Scenario 2 as depicted in Figures 3(b), 4(b), and 5(b). For example, with 10 servers and 25 microservices, the makespan of UB is 0.8. Similarly, with 20 servers and 50 microservices, the makespan is 1.53, and with 50 servers and 120 microservices, the makespan is 2.35, surpassing other methods in each instance. The OMDD-US method performs relatively well in Scenario 2, though with a slightly higher makespan than UB. This is because the US method does not fully utilize the computing capacity of the edge servers. Some microservices that require fewer computing resources are deployed on powerful servers, resulting in wasted resources and a longer runtime. MPE performs less favorably in Scenario 2 as it considers both computation and communication resources, where communication resources are not the primary bottleneck. DFS and BFS perform poorly overall in Scenario 2, but BFS consistently performs better than DFS. This is because DFS uses the method described in [10] to place the paths with the larger edge weights first. In scenario 2, however, the bandwidth of the link is large enough to neglect the transmission time, so that the data transfer between the microservices does not have to be taken into account. Therefore, BFS consistently outperforms DFS in this situation. The SA and QL methods demonstrate moderate performance in Scenario 2, with relatively high makespans across different problem scales, possibly due to their stochastic nature causing significant fluctuations. In addition, QL performs well in small-scale problems but exhibits higher makespans in larger-scale problems, likely due to the complexity of its state space, resulting in extended computation times and an inability to find global optima within a limited number of iterations. In Scenario 2, ISA runs effectively with a lower makespan. The main reason it falls short of UB is due to its stochastic character, which can make it impossible to find the optimal solutions.

(iii). The ISA method consistently minimizes the makespan in Scenario 3 as shown in Figures 3(c), 4(c), and 5(c). Specifically, with 10 servers and 25 microservices, ISA achieves a makespan of 3.99. With 20 servers and 50 microservices, the makespan is 5.21, and with 50 servers and 120 microservices, it is 9.31, outperforming all other methods. Moreover, the performance difference increases by 20%-40% as the problem gets larger. When we compare the performance of the algorithms in different scenarios, we also notice the following trends: the difference between SA and ISA is not significant in small scales, primarily because of fewer deployment strategies available at smaller scales, leading to reasonable solutions for both methods. In larger-scale environments, the performance of the SA algorithm decreases, possibly due to the randomness of the initial solution leading to a local optimum. Additionally, the QL solution shows fluctuations that worsen with increasing size. This is attributed to the vast state space in QL in large-scale environments, making it difficult to find global optima within limited iterations. It is worth noting that the performance of the DFS algorithm improves with increasing scale, which can be attributed to the algorithm's design. Similar to the ISA, DFS considers deploying according to the weight of the paths, indirectly validating the rationale of the main path. To summarize, the experimental results effectively demonstrate the effectiveness and superiority of the proposed algorithms in various scenarios.

(iv). OMDD-ISA achieves the lowest total makespan on both real-world microservice benchmarks. As shown in Table 2, on the Lakeside dataset, OMDD-ISA attains a makespan of 4.01, markedly outperforming other OMDD variants such as OMDD-MPE (5.45), OMDD-UB (7.30), and OMDD-US (6.84), as well as classical heuristics. On the FTGO dataset, OMDD-ISA again yields the best result with a makespan of 3.77, whereas the closest competing method, OMDD-MPE, produces a makespan of 5.59, and the remaining methods exhibit even higher values, demonstrating its superior scheduling effectiveness under practical microservice dependencies. The results indicate that the adaptive strategy incorporated in OMDD-ISA effectively reduces the end-to-end execution time across diverse real-world microservice architectures, and consistently yields more efficient microservice deployment decisions than both classical heuristics (SA, BFS, DFS) and other OMDD variants. Then, we compare performance across both datasets, several trends emerge. SA and BFS provide relatively competitive results on Lakeside, but their makespans increase significantly on FTGO, suggesting limited adaptability to complex or highly heterogeneous service dependencies. In addition, QL and DFS show inconsistent performance, with DFS exhibiting substantial degradation on FTGO, indicating sensitivity to the structural characteristics of the microservice graph. Among the OMDD variants, OMDD-MPE maintains more stable performance but still falls short of OMDD-ISA. Therefore, the real-world benchmark evaluations further validate the effectiveness of OMDD-ISA in practical microservice deployment.

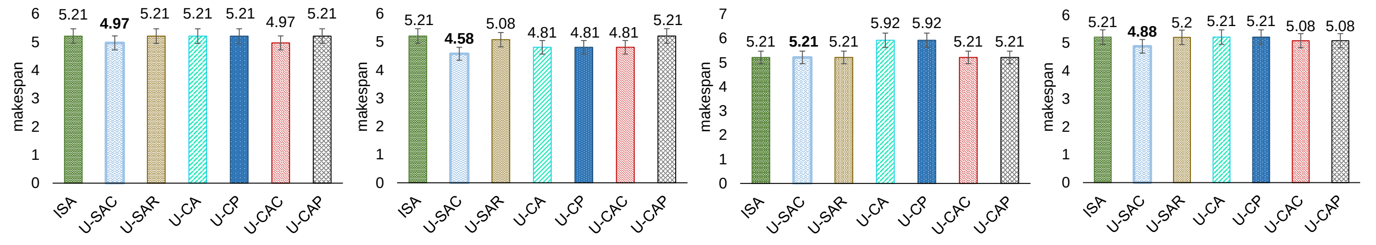
5.2.2 *Microservice Updating*

We evaluated different algorithms at various scales to compare their performance in microservices updates when



(a) Service 2. (b) Service 3. (c) Service 4. (d) Service 5.

Fig. 6. Services updating under the 25 microservices (10 servers).



(a) Service 1. (b) Service 2. (c) Service 4. (d) Service 5.

Fig. 7. Services updating under the 50 microservices (20 servers).

TABLE 4
Comparison of services updating under 120 microservices (50 servers).

Service	ICA	U-SAC	U-SAR	U-CA	U-CP	U-CAC	U-CAP	**Std.Deviation**
S_1	9.31	**8.09**	9.31	**8.65**	**8.65**	**8.65**	9.31	0.4712
S_2	9.31	9.31	9.31	9.31	9.31	10.03	9.31	0.2721
S_3	9.31	9.31	9.31	9.31	9.31	10.03	9.31	0.2721
S_4	9.31	9.31	9.31	9.31	9.31	10.03	9.31	0.2721
S_5	9.31	9.31	9.31	9.31	9.31	10.03	9.31	0.2721
S_6	9.31	9.31	9.31	9.31	9.31	10.03	9.31	0.2721
S_7	9.31	9.31	9.31	9.31	9.31	10.03	9.31	0.2721
S_8	9.31	**8.27**	9.31	9.31	9.31	**9.31**	9.31	0.3931
S_9	9.31	**8.83**	9.31	9.31	9.31	**9.44**	**9.7**	0.2579
S_{10}	**8.65**	**7.68**	**8.65**	**7.7**	**7.7**	**7.7**	**8.65**	**0.5105**
Avg	9.244	8.873	9.244	9.083	9.083	9.528	9.283	–

releasing different services. The experimental results are shown in Figures 5 and 6, and Table 3. We conclude that: (i). The merging critical paths approach in U-SCA effectively reduces the makespan. As shown in Table 3, U-SCA consistently exhibits the shortest makespan, even with the lowest average makespan in various cases (highlighted and underlined in the table). For instance, in Figure 5, at the scale of 10 servers and 25 microservices, after releasing service 1, U-SCA achieves a makespan of 3.26, outperforming all other methods. Despite all methods utilizing critical path merging, the results obtained by the U-CAC algorithm are relatively less favorable. For example, in Figure 7 (a), U-CAC exhibits poor performance with a makespan as high as 6.11. This is because, although the servers identified by U-CAC include the critical path and have surplus resources for updating microservices, the complexity arising from the trade-off between computation time and communication time in the deployment of dependent microservices results in an update strategy that may not be suitable for the current scenario, thus increasing the makespan. In this example, the makespan of U-SAR and U-CAP remains unchanged compared to the scenario where merging is not performed after release (ISA). This is because of the stochastic nature of U-SAR, where the randomly chosen server along the path selected may not release any resources, preventing microservices updates and resulting in an unchanged makespan. Additionally, U-CAP has limitations. It aims to find the critical path for merging but if the server with the highest computing capability on the critical path has no resource release, it may lead to a situation where merging does not occur, causing the makespan to remain unchanged. U-CA and U-CP exhibit relatively good performance when releasing different services. Both of these algorithms first identify servers with resource changes, ensuring the microservices update process. Therefore, they only show the results where the makespan is reduced or remains unchanged due to not updating the location of microservices on the critical path. While not as effective as U-SCA, these two algorithms still perform well.

(ii). For the same scale, the released services have an impact on the makespan after updates. As shown in Table 3 and Table 4, when the released services include the critical path, such as S_1, S_3, and S_{10}, the makespan will decrease even without any updating. This is because the makespan is updated to the completion time of the original subcritical path when the services on the critical path are released according to equation (19). However, it is worth noting that when some services are released, any updating strategy has no effect on the makespan which is shown in Figures 6 (d) and S_1 in Table 4. This is because there is no correlation between the released services and the microservices on the critical path, which means that the position of the non-released microservices on the critical path remains unchanged. Therefore, the makespan still occupies the longest critical path temporarily and remains unchanged. When the released services include microservices on the critical path, the makespan will be updated, as shown in Table 3 with S_{10}. Hence, different released services lead to different updated makespans.

(iii). The number of servers and microservices also affects the results. As shown in Figure 5, we assigned a random capacity in the range $[1, 5]$ for each server, resulting in a scenario with a total capacity of 29, closest to the 25 microservices. However, the makespan in Figure 5 increased by 16.6%. In Figure 6, we similarly assigned a random capacity in the range [1, 5] for each server, resulting in a dataset with a total capacity of 69, with 50 microservices and more available capacity. In this case, U-SAC increased the makespan by 12%. As shown in Table 2, the generated dataset had a total capacity of 127, with 120 microservices, resulting in a 13% increase in makespan under this scenario. We found that when the constraints were very stringent, specifically when the number of microservices and the server capacity were closer, U-SAC demonstrated better performance. This is because, in our improved simulated annealing algorithm, as the available capacity increases, more iterations are needed to find a satisfactory solution. When the number of iterations remains constant, the ability to reduce makespan is more pronounced in situations with smaller remaining capacity and more extreme conditions.

6 Conclusion

This paper focuses on addressing the microservices deployment and updating with dependencies in a resource-constrained mobile edge computing environment. We explore how to optimize the deployment of microservices in various application scenarios, and how to update the locations of microservices to reduce makespan. We initially consider two straightforward scenarios: one with unlimited storage resources under the bandwidth constraint, and the other with unlimited bandwidth resources under the storage constraint. For each of these two scenarios, we introduce a novel enhanced graph construction method and design two optimal solutions. For Scenario 3, which involves complex constraints on server capacity, computational capability, and communication resources, we present an optimization method based on main path partitioning and the simulated annealing algorithm. By flexibly exploring the solution space, we incrementally optimize microservices deployment to adapt to real-world environments and resource constraints. Next, we discuss a more complex and realistic scenario where resources are dynamically released after completing tasks. We propose an update algorithm based on the critical path that adjusts the locations of microservices to reduce the overall makespan. Across multiple experimental results, our approach significantly improves microservices deployment efficiency and overall performance compared to baseline strategies.

In this work, we focus on the issue of microservice deployment and updating with dependencies under resource constraints. However, the current method does not consider user distribution. In our future work, QoS information will be further optimized based on the distribution and trajectory of mobile users.

Acknowledgments

This work was supported by the National Natural Science Foundation of China (62502017, 62072016), the Beijing Natural Science Foundation (4244074), and the Hainan Province Science and Technology Special Fund (ZDYF2024GXJS034).

References

[1] Lv W, Wang Q, Yang P, et al. Microservice deployment in edge computing based on deep Q learning[J]. IEEE Transactions on Parallel and Distributed Systems, 2022, 33(11): 2968-2978.

[2] Duc T L, Leiva R G, Casari P, et al. Machine learning methods for reliable resource provisioning in edge-cloud computing: A survey[J]. ACM Computing Surveys (CSUR), 2019, 52(5): 1-39.

[3] Cerny T, Abdelfattah A S, Bushong V, et al. Microservice architecture reconstruction and visualization techniques: A review[C]//2022 IEEE International Conference on Service-Oriented System Engineering (SOSE). IEEE, 2022: 39-48.

[4] Oakes E, Yang L, Zhou D, et al. SOCK: Rapid task provisioning with Serverless-Optimized containers[C]//2018 USENIX annual technical conference (USENIX ATC 18). 2018: 57-70.

[5] Ding Z, Wang S, Jiang C. Kubernetes-oriented microservice placement with dynamic resource allocation[J]. IEEE Transactions on Cloud Computing, 2022.

[6] Tang B, Guo F, Cao B, et al. Cost-aware Deployment of Microservices for IoT Applications in Mobile Edge Computing Environment[J]. IEEE Transactions on Network and Service Management, 2022.

[7] Wang S, Guo Y, Zhang N, et al. Delay-aware microservice coordination in mobile edge computing: A reinforcement learning approach[J]. IEEE Transactions on Mobile Computing, 2019, 20(3): 939-951.

[8] Samanta, A., Nguyen, T. G., Ha, T., Mumtaz, S. (2022). Distributed resource distribution and offloading for resource-agnostic microservices in industrial iot. IEEE Transactions on Vehicular Technology, 72(1), 1184-1195.

[9] Gu, L., Chen, Z., Xu, H., Zeng, D., Li, B., Jin, H. (2022, May). Layer-aware collaborative microservice deployment toward maximal edge throughput. In IEEE INFOCOM 2022-IEEE Conference on Computer Communications (pp. 71-79). IEEE.

[10] He, X., Tu, Z., Wagner, M., Xu, X., Wang, Z. (2022). Online deployment algorithms for microservice systems with complex dependencies. IEEE Transactions on Cloud Computing.

[11] Adeppady, M., Giaccone, P., Karl, H., Chiasserini, C. F. (2023). Reducing microservices interference and deployment time in resource-constrained cloud systems. IEEE Transactions on Network and Service Management.

[12] Kumar, M., Samriya, J. K., Dubey, K., Gill, S. S. (2023). QoS-aware resource scheduling using whale optimization algorithm for microservice applications. Software: Practice and Experience.

[13] Zhao H, Deng S, Liu Z, et al. Distributed redundant placement for microservice-based applications at the edge[J]. IEEE Transactions on Services Computing, 2020, 15(3): 1732-1745.

[14] Niu Y, Liu F, Li Z. Load balancing across microservices[C]//IEEE INFOCOM 2018-IEEE Conference on Computer Communications. IEEE, 2018: 198-206.

[15] Deng S, Zhao H, Xiang Z, et al. Dependent function embedding for distributed serverless edge computing[J]. IEEE Transactions on Parallel and Distributed Systems, 2021, 33(10): 2346-2357.

[16] Pallewatta S, Kostakos V, Buyya R. QoS-aware placement of microservices-based IoT applications in Fog computing environments[J]. Future Generation Computer Systems, 2022, 131: 121-136.

[17] Hu, Y., Wang, H., Wang, L., Hu, M., Peng, K., Veeravalli, B. (2023). Joint Deployment and Request Routing for Microservice Call Graphs in Data Centers. IEEE Transactions on Parallel and Distributed Systems.

[18] Guo, F., Tang, B., Tang, M. (2022). Joint optimization of delay and cost for microservice composition in mobile edge computing. World Wide Web, 25(5), 2019-2047.

[19] Liao H, Li X, Guo D, et al. Dependency-aware application assigning and scheduling in edge computing[J]. IEEE Internet of Things Journal, 2021, 9(6): 4451-4463.

[20] Guo Y, Wang S, Zhou A, et al. User allocation-aware edge cloud placement in mobile edge computing[J]. Software: Practice and Experience, 2020, 50(5): 489-502.

[21] Li Y, Zhou A, Ma X, et al. Profit-aware edge server placement[J]. IEEE Internet of Things Journal, 2021, 9(1): 55-67.

[22] Pinedo M L. Scheduling[M]. New York: Springer, 2012.

[23] Lv W, Yang P, Zheng T, et al. Graph Reinforcement Learning-based Dependency-Aware Microservice Deployment in Edge Computing[J]. IEEE Internet of Things Journal, 2023.

[24] Wang, C., Jia, B., Yu, H., Li, X., Wang, X., Taleb, T. (2022, December). Deep Reinforcement Learning for Dependency-aware Microservice Deployment in Edge Computing. In GLOBECOM 2022-2022 IEEE Global Communications Conference (pp. 5141-5146). IEEE.

[25] Qi, J., Zhang, H., Li, X., Ji, H., Shao, X. (2023, March). Edge-edge Collaboration Based Micro-service Deployment in Edge Computing Networks. In 2023 IEEE Wireless Communications and Networking Conference (WCNC) (pp. 1-6). IEEE.

[26] Li, X., Zhou, J., Wei, X., Li, D., Qian, Z., Wu, J., Qin, X. and Lu, S., 2023. Topology-Aware Scheduling Framework for Microservice Applications in Cloud. IEEE Transactions on Parallel and Distributed Systems, 34(5), pp.1635-1649.

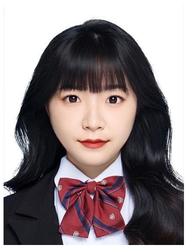
[27] Singh, Vindeep, and Sateesh K. Peddoju. ”Container-based microservice architecture for cloud applications.” 2017 International Conference on Computing, Communication and Automation (ICCCA). IEEE, 2017.

[28] Zambianco, Marco, Silvio Cretti, and Domenico Siracusa. ”Cost Minimization in Multi-cloud Systems with Runtime Microservice Re-orchestration.” 2024 27th Conference on Innovation in Clouds, Internet and Networks (ICIN). IEEE, 2024.

[29] Sampaio, Adalberto R., et al. ”Improving microservice-based applications with runtime placement adaptation.” Journal of Internet Services and Applications 10 (2019): 1-30.

[30] Hossen, Md Rajib, Mohammad A. Islam, and Kishwar Ahmed. ”Practical efficient microservice autoscaling with QoS assurance.” Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing. 2022.

Shuaibing Lu is currently a lecturer at the College of Computer Science at Beijing University of Technology. She received her PhD in Computer Science and Technology from Jilin University, Changchun in 2019. She was supported by the China Scholarship Council as a visiting scholar under the supervision of Prof. Jie Wu in the Department of Computer and Information Science at Temple University (2016-2018). She is a member of IEEE. Her current research focuses on distributed computing, cloud computing, and edge computing.



Ran Yan received her B.Sc. in Network Engineering at Beijing Information Science and Technology University. Currently, she is working toward her M.Sc. degree in the College of Computer Science at Beijing University of Technology. Her research interests include cloud computing and edge computing.

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

Zhi Cai is an associate professor in the College of Computer Science, Beijing University of Technology, China. He obtained his M.Sc. in 2007 from the School of Computer Science at the University of Manchester and Ph.D. in 2011 from the Department of Computing and Mathematics of the Manchester Metropolitan University, U.K. His research interests include Information Retrieval, Ranking in Relational Databases, Keyword Search, and Intelligent Transportation Systems.

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

Shuyang Zhou is currently a junior at Beijing Jiaotong University, majoring in Software Engineering. His current research interests are focused on edge computing and cloud computing.

Haiming Liu is currently a lecturer in the School of Software Engineering at Beijing Jiaotong University. He received his Ph.D. degree in Computer Science and Technology (Bioinformatics) from Jilin University, Changchun, in 2019. He is a member of the Chinese Association for Artificial Intelligence (CAAI). His current research focuses on edge computing, data mining, and bioinformatics.

Juan Fang Juan Fang, received her M.S. degree from Jilin University of Technology, Changchun, China in 1997, and her Ph.D. degree from the College of Computer Science, Beijing University of Technology, Beijing, China, in 2005. In 1997, she joined the College of Computer Science, Beijing University of Technology. From 2015, she has been a professor at Beijing University of Technology. Her research interests include high performance computing, edge computing and big data technology.