International Journal of Innovative Computing, Information and Control
Volume **21**, Number **4**, August **2025**

ICIC International ©2025 ISSN 1349-4198
pp. **859–883**

AN EDGE SERVER LOAD BALANCING METHOD BASED ON PARTICLE SWARM OPTIMIZATION

Xiong Yang[1], Yucheng Shi[2], Ruhui Chen[1] and Huihua Xu[3,*]

[1]Department of Computer Engineering
Fuzhou University Zhicheng College
No. 50, Yangqiao West Road, Gulou District, Fuzhou 350000, P. R. China
02116828@fdzcxy.edu.cn; 241027054@fzu.edu.cn

[2]College of Computer and Data Science
Fuzhou University
No. 2, Wulongjiang North Avenue, Minhou County, Fuzhou 350000, P. R. China
221027167@fzu.edu.cn

[3]Department of Economics and Law
Concord University College Fujian Normal University
No. 68, Xuefu South Road, Minhou County, Fuzhou 350000, P. R. China
*Corresponding author: fbx20210003@yjs.fjnu.edu.cn

Received November 2024; revised March 2025

Abstract. *With the rapid advancement of technology, the proliferation of portable smart devices, such as smartphones and smart bracelets, has led to increasingly demanding task-processing requirements from users. Mobile Edge Computing (MEC) addresses the latency issues inherent in traditional cloud computing by offloading tasks to proximate edge servers for execution. However, the dynamic nature of user locations and the stochasticity of task generation introduce task distribution imbalances, with some edge servers becoming overloaded while others still need to be utilized. Therefore, load balancing among edge servers has emerged as a critical challenge in MEC. To address this challenge, we propose an Edge Server Load Balancing method based on Particle Swarm Optimization called ESLB-PSO, which achieves load balancing of each edge server by minimizing the maximum response time of tasks. This method models the MEC environment as a topology, abstracting the corresponding problem according to the operational principles of real-world edge servers. Each particle represents a solution, and the task offloading plan is optimized by adjusting the particle's speed, direction, and approach toward the global optimum. Experimental results demonstrate that ESLB-PSO outperforms baseline methods, effectively achieving load balancing across edge servers, and obtains a better maximum response time of the tasks.*

Keywords: Mobile edge computing, Load balancing, Particle swarm optimization, Task offloading

1. **Introduction.** With the advent of the 5G and the widespread adoption of mobile devices, many mobile applications generate vast amounts of real-time data that require immediate processing. Applications such as augmented reality and image recognition demand substantial computational and storage resources, which mobile devices are inherently incapable of handling locally. To address this challenge, Mobile Cloud Computing (MCC) [1] has emerged, utilizing the extensive computational and storage capabilities of cloud data centers to offload complex tasks from mobile devices, thereby meeting the high-resource demands of mobile applications [2]. However, transmitting task data to remote data centers for processing and returning the results introduces significant latency, which

DOI: 10.24507/ijicic.21.04.859

severely impacts user experience and fails to meet the stringent low-latency requirements of the 5G [3].

In response to these challenges, Mobile Edge Computing (MEC) [4] has emerged as a promising solution, significantly reducing network latency and meeting the real-time service requirements of mobile applications. By deploying edge servers at the network edge, MEC brings computing and storage resources closer to the data source, creating an open platform that integrates network, computing, storage, and application core capabilities. Compared to traditional MCC, MEC offers several notable advantages [5]. By positioning servers at the network edge, MEC minimizes the distance between end devices and servers, reducing task transmission latency. Additionally, the substantial computational power of edge servers enhances task processing speed. MEC allows mobile devices to offload computational tasks to the edge servers, alleviating the computational burden on mobile devices and reducing their energy consumption, ultimately improving energy efficiency.

5G networks carry a variety of service types, covering services such as smart home, and intelligent transportation, which have a high demand for global information and a long cycle [6] but low sensitivity to latency, as well as augmented reality, online games, etc., which have a very high demand for latency but a small amount of data processing. To ensure efficient processing, different types of tasks require reasonable allocation of computing resources [7]. However, because edge servers are deployed at the edge of the network, their computation and storage capacities are limited, and the hardware configurations of different servers are different, resulting in an imbalance of the overall computation resources. In addition, the dynamic arrival pattern of tasks further aggravates the uneven load distribution, causing some servers to be overloaded for a long time while other servers are idle, resulting in wasted computing resources and decreased storage utilization. Therefore, the impact of load imbalance on the MEC system is particularly significant. First, the task queuing time of overloaded servers increases, leading to an increase in the overall task processing latency and affecting the system real-time performance. Second, the unbalanced utilization of computing resources reduces the overall throughput of the MEC network and limits the system service capability. In addition, in high latency sensitive application scenarios (e.g., autonomous driving), overloading of the edge servers may lead to the delay or even loss of critical computing tasks, which seriously affects driving safety. Therefore, implementing an efficient load balancing strategy is crucial to optimizing the allocation of computing resources and improving the overall performance of the MEC system. Load balancing not only determines the task processing capability of MEC network, but also directly affects the energy efficiency and user experience of the system, which is one of the core issues in MEC research [8].

Recently, many studies have focused on solving the MEC load balancing problem using different decision-making methods. Some studies [9,10] used static offloading methods. However, they lacked adaptivity to adjust to the dynamic load changes of edge servers, which can easily lead to local overload or resource waste. Some studies [11,12] used traditional heuristic optimization methods. However, their search efficiency is low, easy to fall into local optimums, and the computational overhead is large in large-scale MEC networks, which makes it difficult to meet the real-time scheduling requirements of low-latency tasks. Some studies [13,14] used reinforcement learning-based methods. However, since MEC task offloading involves high-dimensional state spaces, traditional reinforcement learning methods have slow convergence speeds, and it is difficult to efficiently explore complex policies during the training process.

To solve the above problem, we propose an Edge Server Load Balancing method based on Particle Swarm Optimization called ESLB-PSO. ESLB-PSO utilizes the global optimization capability of PSO to model MEC task offloading as an optimization problem

and optimize the task offloading plan through iterative particle swarm search. Specifically, ESLB-PSO adopts a matrixed representation of the task offloading plan, where each particle corresponds to one task offloading plan, and dynamically adjusts the movement operation of the solution and the parameters of the particle swarm during the searching process, so as to continuously optimize the task offloading. Meanwhile, ESLB-PSO dynamically adjusts the inertia weights to enhance the global exploration capability at the initial stage to avoid falling into the local optimum, and accelerates the convergence at the later stage to improve the computational efficiency and task response performance. The main contributions are summarized as follows.

- A load balancing scenario with multi-edges server collaboration in MEC is considered. Each edge server receives independent tasks offloaded from mobile devices. The edge servers can offload tasks to neighboring edge servers for processing over a wireless connection to reduce their load rate. The optimization goal is to find the target edge server load balancing plan to minimize the task maximum response time in the edge servers.
- A Particle Swarm Optimization (PSO)-based task offloading strategy is proposed. By representing the task offloading plan in a matrix format, the algorithm generates valid solutions randomly while adhering to specific constraints. Moreover, the global search capability of the PSO is enhanced through dynamic adjustments to the particle movement and swarm parameters. This refinement enables the algorithm to optimize the task allocation process in a multi-edge server environment, thereby achieving efficient load balancing within the MEC framework.
- Simulation experiments verify the effectiveness of the proposed ESLB-PSO. The results show that ESLB-PSO can effectively balance load in different scenarios. In addition, the task maximum response time of ESLB-PSO outperforms the baselines in different scenarios.

2. **Related Work.** Due to the limited computational power of mobile devices, they cannot perform computationally intensive tasks locally. The rapid development of 5G, characterized by its low latency and high bandwidth, has opened up the possibility of task migration, prompting extensive research into offloading tasks from mobile devices to the cloud. This approach, known as MCC, aims to overcome the inherent limitations of mobile devices. However, MCC still faces significant challenges. The considerable distance between mobile devices and cloud data centers results in non-negligible transmission times for task data. Additionally, network latency, influenced by varying network quality, further impacts the overall performance of this approach.

Kemp et al. [9] proposed a runtime system based on the Cuckoo framework that dynamically decides whether application tasks should be executed locally or offloaded to the cloud. Cuervo et al. [10] introduced the MAUI system, which supports partitioning mobile application tasks and determining which tasks should be offloaded to the cloud. Kosta et al. [15] proposed the ThinkAir system, which leverages mobile virtualization to migrate applications to the cloud, reducing execution time and power consumption through parallel processing with multiple virtual machines. Somula and Sasikala [16] developed an algorithm based on edge node load and distance, which selects the optimal edge node for task offloading cyclically to reduce user request wait times in server queues. Zhang et al. [17] presented a service utility-based scheduling algorithm that prioritizes task scheduling for users with the highest service utility. The above studies mainly focus on optimizing task allocation and offloading from the user side to reduce latency. Although these studies improve efficiency to some extent, in real-world scenarios, the continuous movement of mobile users often prevents real-time optimal decision-making based on the state of the

edge network. Typically, tasks are offloaded to the nearest edge server, leading to load imbalances among edge servers. Therefore, task redistribution at the edge network level to achieve load balancing is critical for reducing user task response times.

Jia et al. [11] studied the problem of edge placement and mobile user task assignment and proposed an algorithm that positions edge servers in user-dense areas, subsequently assigning mobile user tasks to these edges. This approach balances the load of edge servers and reduces the distance between most users and the edge servers through strategic edge location selection. Ma et al. [12] utilized a PSO algorithm to select optimal edge locations, addressing the latency issue. Xu et al. [18] employed heuristic algorithms to place multiple edge servers with varying computational capabilities at critical locations within an infinite local area network, aiming to reduce the average access latency for mobile users at different wireless access points. Yang et al. [19] proposed a Differential Evolution (DE)-based multi-drone deployment mechanism to address the computational offloading problem for ground-based IoT nodes, achieving load balancing among unmanned aerial vehicles. Merah et al. [20] proposed a Genetic Algorithm (GA)-based load balancing method for Internet of Things (IoT), which intelligently distributes loads among servers through genetic algorithms to achieve load balancing. The above studies adopt a heuristic algorithm; however, the computational overhead is large in large-scale MEC networks, the search efficiency is low, and it is easy to fall into the local optimum. In addition, although these studies consider user mobility, they overlook the queuing time required for subsequent tasks when multiple tasks are at an edge server. This limitation hinders their ability to meet the demands of real-world scenarios fully. Du et al. [13] proposed a load balancing method for cloud-edge environments based on Q-learning, which searches for the optimal Q-value of the movement to reduce the processing time of the tasks. Li et al. [14] proposed a Deep Q-Network (DQN) based load balancing method for IoT, which manages and offloads local tasks from each edge server through the network probabilities output by DQN to minimize the mean square deviation of loads among different edge servers. The above studies use a reinforcement learning based method; however, due to the high dimensional state space involved in task offloading, traditional reinforcement learning methods converge slowly and it is difficult to efficiently explore complex policies during the training process.

Unlike the above studies, we study a load balancing scenario in an MEC environment with multiple tasks and consider the queuing of subsequent tasks. We design a PSO algorithm for the scenario to derive the optimal load balancing policy through continuous loop iteration.

3. **Problem Definition.**

3.1. **System model.**

As shown in Figure 1, we consider an MEC environment where edge servers are widely distributed, forming a topological structure. The set of N edge servers in this environment is defined as $E = \{e_1, e_2, e_3, \ldots, e_N\}$. Each edge server can receive tasks uploaded by mobile devices within its coverage area via a wireless network and connect to nearby edge servers for mutual task offloading. This paper assumes that tasks are independent, meaning they can be dynamically partitioned and scheduled on edge servers [11]. For instance, 50% of the task load on a particular edge server can be executed locally, while the remaining 50% can be offloaded to several neighboring edge servers. Due to the mobility of mobile devices and the uncertainty of task offloading to edge servers, the load rates of edge servers can fluctuate significantly. High load rates on edge servers lead to excessive task execution delays, while low load rates result in severe resource wastage. Therefore, a rapid and efficient task offloading plan is urgently needed

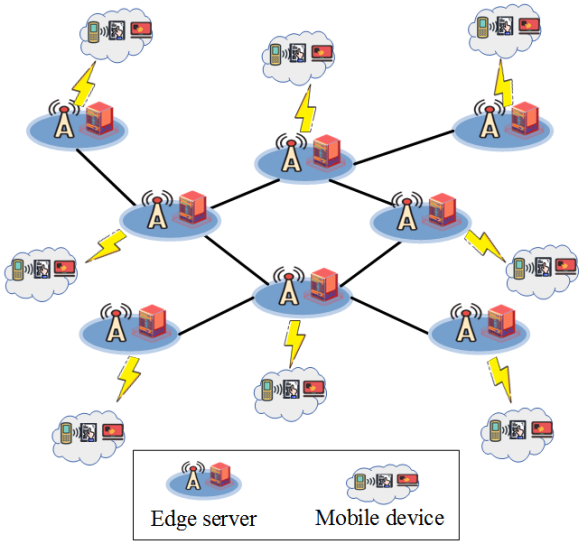


FIGURE 1. Load balancing for an MEC environment

TABLE 1. Definition of symbols

Symbol	Definition
E	Set of edge servers
e_i	The i-th edge server
N	Number of edge servers
μ_i	Computing power of e_i
λ_i	Arrival task of e_i
L_i	Set of neighboring edge servers of e_i
$Move(i,j)$	Number of tasks offloaded from e_i to e_j
$Solution$	Task offloading plan
$Ttotal^i$	Task response time of e_i
$Tnet^i$	Transmission time for e_i to receive the task
$Texe^i$	Execution time for e_i to receive the task
$Distance$	Transmission time matrix for MEC environments
d_{ij}	Transmission time from e_i to e_j
λ_i	Load on e_i

to enable effective collaboration between edge servers, achieve load balancing, reduce overall task response time, and improve system resource utilization. The main symbols used in this paper are defined in Table 1.

We define the computing power of N edge servers as $\mu = \{\mu_1, \mu_2, \mu_3, \ldots, \mu_N\}$, where μ_i indicates the number of tasks that the i-th edge server e_i can handle per unit of time. Tasks transmitted from mobile devices to edge servers are called arrival tasks. We define the task arrival rates for the N edge servers as $\lambda = \{\lambda_1, \lambda_2, \lambda_3, \ldots, \lambda_N\}$, where λ_i indicates the number of arrival tasks received by the e_i per unit of time. To ensure that the arrival tasks do not exceed the computing power of the edge servers, we impose the constraint $\lambda < \mu$.

Based on the above definitions, we define an edge server as a quadruple:

$$e_i = \{Number_i, \mu_i, \lambda_i, L_i\} \tag{1}$$

where $Number_i$ indicates the identifier of the i-th edge server, and L_i indicates the set of neighboring edge servers connected to the i-th edge server. Each edge server is allowed to connect to at most K neighboring servers.

We assume that each edge server can offload a portion of its arrival tasks to its neighboring edge servers for processing. We define $Move(i, j)$ as the number of tasks offloaded from e_i to e_j. Considering practical scenarios, the task offloading process must satisfy the following constraints:

- The number of tasks offloaded by e_i to its neighboring edge servers must be less than or equal to its task arrival rate:

$$\sum_{j=1}^{N} Move(i, j) < \lambda_i \tag{2}$$

- After all edge servers complete task offloading, the number of tasks remaining on e_i must not exceed its computing power. Otherwise, it would impact task processing in the next time step:

$$\lambda_i + \sum_{j=1}^{N} Move(j, i) - \sum_{j=1}^{N} Move(i, j) < \mu_i \tag{3}$$

- For the MEC environment, the total number of offloaded tasks must equal the total number of received tasks, ensuring a balanced task transfer:

$$\sum_{i=1}^{N} \sum_{j \in L_i} Move(i, j) = \sum_{i=1}^{N} \sum_{j \in L_i} Move(j, i) \tag{4}$$

We use the matrix $Solution \in R^{N*N}$ to store the task offloading between the edge servers in the MEC environment. The matrix $Solution$ is initialized as a zero matrix, and then the values of $Move(i, j)$ are sequentially placed into the positions $Solution[i][j]$, indicating the task offloading plan.

We define the task response time $Ttotal^i$ for e_i to complete all of its current tasks as the sum of the task transmission time $Tnet^i$ and the task execution time $Texe^i$:

$$Ttotal^i = Tnet^i + Texe^i \tag{5}$$

The task transmission time $Tnet^i$ depends on the distance between edge servers, which is the network latency. We denote the transmission time for offloading a task from e_i to e_j as d_{ij}, and then the transmission time for the environment is denoted as a matrix $Distance \in R^{N*N}$:

$$Distance = \begin{bmatrix} d_{11} & \cdots & d_{1N} \\ \vdots & \ddots & \vdots \\ d_{N1} & \cdots & d_{NN} \end{bmatrix} \tag{6}$$

Based on the above definitions, when the number of tasks offloaded from e_i to e_j, the transmission time required for e_j to receive these tasks is $Move(i, j) \times d_{ij}$. Therefore, the total transmission time for e_i to receive tasks offloaded from neighboring servers is

$$Tnet^i = \sum_{j \in L_i} Move(i, j) \times d_{ij} \tag{7}$$

The task execution time $Texe^i$ consists of the queuing time and the processing time [14]:

$$Texe^i(\lambda) = \frac{C\left(n_i, \frac{\lambda}{\mu_i}\right)}{n_i\mu_i - \lambda} + \frac{1}{\mu_i} \tag{8}$$

where μ_i indicates the computing power of e_i. Using the Erlang-C formula [21], we can calculate the probability that a task must wait before processing. This formula is based on the traffic intensity and the number of edge servers. The Erlang-C formula is expressed as

$$C(n,\rho) = \frac{\left(\frac{(n\rho)^n}{n!}\right)\left(\frac{1}{1-\rho}\right)}{\sum_{k=0}^{n-1}\frac{(n\rho)^k}{k!} + \left(\frac{(n\rho)^n}{n!}\right)\left(\frac{1}{1-\rho}\right)} \tag{9}$$

Each edge server has $n = 1$. By combining Equations (8) and (9), we can obtain the task execution time as

$$Texe^i(\lambda) = \frac{\lambda_i}{\mu_i\left(\mu_i - \lambda_i\right)} + \frac{1}{\mu_i} \tag{10}$$

where λ_i indicates the actual load of e_i at the current time. As tasks are continuously offloaded between edge servers, the actual load of each edge server changes dynamically. Therefore, before calculating the task execution time, it is necessary to update the value of λ_i based on the task offloading matrix. The updated λ_i can be expressed as

$$\lambda_i = \lambda_i - \sum_{j\in L_i} Move\,(i,j) + \sum_{j\in L_i} Move\,(j,i) \tag{11}$$

3.2. **Optimization goal.** In this paper, we consider a multi-edge server environment. The optimization goal is to minimize the maximum response time of tasks among all edge servers while maintaining a reasonable and balanced load among the edge servers. Therefore, we define the optimization goal as

$$\begin{aligned}
&\textbf{minimize}\max\left\{Ttotal^1, Ttotal^2, Ttotal^3, \ldots, Ttotal^N\right\}\\
&C1:\ \sum_{j=1}^{N} Move(i,j) < \lambda_i\\
&C2:\ \lambda_i + \sum_{j=1}^{N} Move(j,i) - \sum_{j=1}^{N} Move(i,j) < \mu_i\\
&C3:\ \sum_{i=1}^{N}\sum_{j\in L_i} Move(i,j) = \sum_{i=1}^{N}\sum_{j\in L_i} Move(j,i)
\end{aligned} \tag{12}$$

where constraint $C1$ indicates that the number of tasks offloaded out of each server cannot exceed its task arrival rate, constraint $C2$ indicates that the task load of each server cannot exceed its computing power, and constraint $C3$ indicates that the total number of offloaded tasks in the MEC environment needs to be equal to the total number of received tasks. This objective function embodies our optimization goal of finding a global task offloading plan $Move(i,j)$ to optimize the quality of service of the entire MEC environment. We use the global task offloading plan $Move(i,j)$ as a task allocation decision, whose decision result is crucial for the efficiency of task response in a multi-edge server environment. By achieving the minimization goal, we aim to improve the overall performance and achieve load balancing while minimizing the maximum response time.

4. The Proposed ESLB-PSO.

4.1. Algorithmic solution.

4.1.1. *Solution representation.* We denote the task offloading plan by the matrix *Solution* $\in R^{N*N}$. The matrix *Solution* $\in R^{N*N}$ consists of a series of offloaded tasks, representing a feasible task offloading plan. We consider *Solution* as a solution of the algorithm. The global optimal solution at the i-th iteration of the PSO is defined as

$$Solution^i = \begin{bmatrix} s_{11}^i & \cdots & s_{1N}^i \\ \vdots & \ddots & \vdots \\ s_{N1}^i & \cdots & s_{NN}^i \end{bmatrix} \tag{13}$$

where s_{ab}^i indicates the number of tasks offloaded from the a-th edge server to the b-th edge server during the i-th iteration. Specifically, when $a = b$, $s_{ab}^i = 0$, indicating that a server cannot offload tasks to itself. Additionally, this solution satisfies the constraints defined by Equations (2)-(4), which ensure the validity of the task offloading plan and its adherence to the operational constraints of the MEC environment.

4.1.2. *Solution generation operation.* We first generate a base matrix based on the initial parameters and conditions to generate a solution randomly distributed in the solution space while also satisfying the constraints. After that, we verify if the generated solution satisfies all the constraints. If it does, the solution is accepted. If it does not meet the constraints, the solution is discarded, and a new one is generated. This process of solution generation is outlined in Algorithm 1.

Algorithm 1's input is the set of edge servers E and the number of edge servers N. If a valid solution is generated, it is returned. Otherwise, *False* is returned, and based on the result, it is determined whether to regenerate.

Step 1: The algorithm initializes a zero matrix *Solution* (Line 3) and generates the random ratio value *ratio* for the current solution (Line 4).

Step 2: Each edge server is processed in turn corresponding to its row in the solution matrix *Solution*. For the i-th edge server, the average computing power $E_i \cdot \mu_i$ is obtained, and the range of tasks to be offloaded is calculated as $arr = ratio \times E_i \cdot \mu_i$ (Lines 6-7). Then, the set of neighboring edge servers for the i-th edge server is obtained. For each neighboring j-th edge server, a random value from $[0, arr]$ is chosen and filled into $Solution[i][j]$, representing the number of tasks transferred from the i-th edge server to the j-th edge server (Lines 8-10).

Step 3: For each edge server, the i-th row and j-th column of the solution matrix *Solution* are traversed to obtain the output task number *JobOutput* and input task number *JobInput* (Lines 14-19). Based on *JobOutput* and *JobInput*, whether the solution satisfies the constraints of Equations (2)-(4) is determined.

Step 4: The current solution is returned if the *Solution* meets all constraints. Otherwise, *False* is returned, and based on the algorithm's return value, it is determined whether to regenerate.

4.1.3. *Solution movement operation.* During the PSO iteration process, the particle representing a solution must continuously move based on its direction and the global trend. Given the matrix form of the solution, we define the solution movement operation as a matrix $V \in R^{N*N}$. We consider each row as a whole in the movement operation matrix V. For instance, we represent the i-th row as $v_i = \{v_{i1}, v_{i2}, v_{i3}, \ldots, v_{iN}\}$, where non-zero values indicate the task offloading status of the i-th edge server.

Algorithm 1: Solution generation algorithm

Input: E and N
Output: S or *False*
Initialize: Zero matrix $S \in R^{N*N}$
Generate random numbers $ratio \in [0, 1]$
for $i = 0$ to $i = N - 1$ **do**
    Obtain the average computing power $E_i \cdot \mu_i$
    Obtain the ratio of $E_i \cdot \mu_i$ as the range of offloaded tasks for e_i, denoted as $arr = ratio \times E_i \cdot \mu_i$
    for j in $E_i \cdot L_i$ **do**
        $S[i][j] = uniform(0, arr)$
    end for
end for
$judge = True$
for $i = 0$ to $i = N - 1$ **do**
    $JobInput = 0$
    $JobOutput = 0$
    for $j = 0$ to $j = N - 1$ **do**
        $JobOutput+ = S[i][j]$
        $JobInput+ = S[j][i]$
    end for
    if $JobOutput > E_i \cdot \mu_i$ **do**
        $judge = False$
    end if
    if $E_i \cdot \lambda_i + JobInput - JobOutput > E_i \cdot \mu_i$ **do**
        $judge = False$
    end if
end for
if $judge = True$ **do**
    return S
else do
return *False*

1) When $v_i > 0$, the i-th edge server currently has an excessive number of tasks, resulting in a longer task processing time, thus requiring an increase in the number of tasks offloaded to neighboring edge servers.
2) When $v_i < 0$, the i-th edge server currently has fewer tasks relative to its processing capability, thus requiring a decrease in the number of tasks offloaded to neighboring edge servers.

When the movement operation is applied to the solution, the following constraints need to be enforced.

1) When the particle's corresponding solution changes, the values of the movement operation matrix for the row corresponding to the i-th edge server are effective only for the indices of neighboring edge servers.
2) When the values in the *Solution* need to be increased, i.e., $v_i > 0$, it is necessary to calculate whether the sum of the i-th row in the *Solution* exceeds the task arrival rate

of the corresponding edge server after the increase. If it does, the values in the i-th row of the *Solution* are randomly decreased until the requirements are met. Conversely, when the values in the *Solution* need to be decreased, i.e., $v_i < 0$, they are set to zero if the decreased values fall below zero.

4.2. **PSO-based task offloading strategy.** PSO [23] is a swarm intelligence optimization algorithm inspired by the cooperation and information sharing observed in animal groups such as flocks of birds, herds of animals, and schools of fish. In the PSO, individuals collaborate and share information to find the optimal solution. This algorithm is straightforward to implement and requires minimal parameter tuning. It is widely applicable in various fields such as function optimization, neural network training, cloud computing load balancing, fuzzy system control, and other applications where genetic algorithms are utilized. Consequently, PSO is a heuristic global optimization algorithm [24].

4.2.1. *Definition of particle swarm.* We define each particle in the swarm as a feasible solution in the MEC environment:

$$Group = \begin{bmatrix} g_{11}^t & \cdots & g_{1N}^t \\ \vdots & \ddots & \vdots \\ g_{N1}^t & \cdots & g_{NN}^t \end{bmatrix} \tag{14}$$

where g_i^t indicates the result of the i-th particle in the population at the t-th iteration. The entire population set is denoted as *Group*. Additionally, each particle in the population has a corresponding velocity v, which is defined as

$$v_i^t = \begin{bmatrix} v_{11}'^t & \cdots & v_{1N}'^t \\ \vdots & \ddots & \vdots \\ v_{N1}'^t & \cdots & v_{NN}'^t \end{bmatrix} \tag{15}$$

4.2.2. *Movement operations of particle swarm.* In each iteration of the PSO process, every particle needs to move to update its position continuously. Each particle moves towards the current global best solution while maintaining its travel direction. Therefore, the velocity and direction of the particle are constantly being updated:

$$V_i^{t+1} = \omega V_i^t + c_1 r_1 \left(pbest_i - S_i^t\right) + c_2 r_2 \left(gbest_i - S_i^t\right) \tag{16}$$

where ωV_i^t indicates the influence of the particle's previous iteration speed on the current iteration's speed and direction. c_1 and c_2 indicate the cognitive and social learning factors, respectively. r_1 and r_2 indicate random numbers in the range $[0, 1]$, respectively. $pbest_i$ indicates the best solution found by the i-th particle so far. $gbest_i$ indicates the global best solution found by the entire swarm up to the current iteration. $pbest_i - S_i^t$ indicates the vector from the particle's current position to its personal best position, reflecting the influence of the particle's own experience. $gbest_i - S_i^t$ indicates the vector from the particle's current position to the global best position, indicating the collaborative influence among particles in the swarm.

During the PSO iteration process, the primary objective in the early stages is to expand the search range within the solution space, exploring as many possibilities as possible to reduce the likelihood of getting trapped in local optima. In the later stages, the goal shifts to quickly converging towards the global optimal solution to obtain the best final result. Therefore, we have incorporated an adjustment operation for the value into the algorithm design to enhance its performance:

$$\omega = \omega_{\max} - t \cdot \frac{(\omega_{\max} - \omega_{\min})}{Gen}, \quad \omega \in [\omega_{\min}, \omega_{\max}] \tag{17}$$

Using the above equation, as the number of iterations t increases, the ratio of t to the total number of iterations Gen also increases. This causes the ω value to decrease, gradually reducing the influence of the particle's inertia on its velocity V. Consequently, the tendency for particles to converge towards the global optimal solution becomes stronger with each iteration.

4.2.3. *Iterative updating of particle swarm.* After each update of the particle's position and velocity, the global best solution *gbest* and the particle's local best solution *pbest* need to be updated. In one iteration, for particle i, the evaluation of the current solution $Ttotal_i$ is calculated and compared with the previous local best solution $Ttotalbest_i$. If $Ttotal_i < Ttotalbest_i$, then $Ttotalbest_i = Ttotal_i$, updating the local best solution of particle i.

Based on the above definition, we propose a process for generating a task offloading plan based on PSO, as shown in Algorithm 2.

Algorithm 2: A process for generating task offloading plan based on PSO

Input: E, $Distance$, $Pnum$ and $Generation$
Output: $gbest$ and $gTtotalbest$
Initialize: $\omega_{\max} = 0.8$, $\omega_{\min} = 0.3$, $c_1 = c_2 = 2$
Initialize: Generate the initial population set $Group$ according to the input particle number $Pnum$
Initialize: Calculate the individual best value set $pTtotalbest$ and the corresponding solutions set $pbest$, and calculate the global best value $gTtotalbest$ and the corresponding solutions set $gbest$.
Initialize: velocity set of each particle V
for $gen = 0$ to $gen = Generation$ **do**
    Update the inertia weight ω by $\omega = \omega_{\max} - gen \cdot \frac{(\omega_{\max} - \omega_{\min})}{Generation}$
    for each g in $Group$ **do**
        Generate random numbers $r_1, r_2 \in [0, 1]$
        Update the velocity of the particle by $V_i^{t+1} = \omega V_i^t + c_1 r_1 (pbest_i - S_i^t) + c_2 r_2 (gbest_i - S_i^t)$
        Update the velocity of particle i
        Check if the updated particle velocity satisfies conditions Equations (2)-(4)
        If not satisfied, adjust the velocity by reducing the corresponding term or taking the minimal value until satisfied
        Calculate the evaluation $Ttotal_i$ of the particle
        if $Ttotal_i < Ttotalbest_i$ **do**
            $Ttotalbest_i = Ttotal_i$
            $pbest_i = Group_i$
        end if
        if $Ttotal_i < gTtotalbest$ **do**
            $gTtotalbest = Ttotal_i$
            $gbest = Group_i$
        end if
    end for
end for
return $gTtotalbest$

Algorithm 2's input is the set of edge servers E, the transmission time matrix $Distance$, the number of particles $Pnum$, the number of iterations $Generation$, the upper and lower limits of the inertia weight $\omega_{\max}$ and $\omega_{\min}$ defined in the algorithm, and the learning factors c_1 and c_2. The output of Algorithm 2 is the evaluation value of the global optimal solution and the optimal offloading plan corresponding to the global optimal solution.

Step 1: Generate $Pnum$ feasible solutions through Algorithm 1 to form the particle swarm set $Group$, initialize the individual best value set $pTtotalbest$ and the individual best solution set $pbest$, initialize the global best value $gTtotalbest$ and the global best solution $gbest$, and initialize the initial velocity set V of each particle in the particle swarm set $Group$ (Lines 3-6).

Step 2: Perform iterative updates on the particle swarm, updating the value of the inertia weight ω. For each particle in the particle swarm, perform the following operations.

1) Generate random numbers r_1 and r_2 in the range $[0, 1]$. Update the velocity of particle i according to ω, r_1, r_2, the current velocity, the global best solution $gbest$, and the local best solution $pbest_i$ using Equation (16) (Lines 10-11).
2) Update particle i according to its velocity V_i. Note that if any value in the matrix is less than 0 after the update, it is set to 0. After the update, check whether the particle's operation in the MEC environment meets the constraints of Equations (2)-(4). If constraints are not satisfied, randomly reduce the values in the matrix to a minimum of 0 until they are met, completing the update of particle i (Lines 12-14).
3) Calculate the evaluation of particle i in the current iteration, i.e., its corresponding maximum response time $Ttotal_i$, and compare it with the local best solution of particle i. If the current particle i evaluation is better than the local best solution time, update the local best solution. Additionally, compare $Ttotal_i$ with the global best value $gTtotalbest$. If $Ttotal_i < gTtotalbest$, then set it to update the global best solution (Lines 15-23).

4.3. **Random migration-based task offloading strategy.** The Random Migration Algorithm (RMA) [25] is one of the comparison algorithms we selected. This algorithm generates many solutions that meet the constraints through random generation, forms a set, and then sequentially tests and calculates the maximum response time of each solution in the set. The optimal solution is then extracted and returned. In this paper, we use Algorithm 1 to generate many random matrices to implement the Random Migration Algorithm. The generated solution set is then validated, and the optimal solution is returned. The task scheduling strategy based on the Random Migration Algorithm is shown in Algorithm 3.

Algorithm 3's input is the set of edge servers E and the number of randomly generated matrices $RTnum$, and outputs the smallest maximum response time corresponding to the randomly generated set of solutions.

Step 1: Based on the value of $RTnum$, randomly generate the corresponding number of valid solutions that satisfy the constraints in Equations (2)-(4), and add these solutions to the solution set $RTSolutions$ (Lines 4-12).

Step 2: Sequentially apply the solutions in the $RTSolutions$ set to the edge server set E, calculate the maximum response time for each solution, and record the smallest value in the variable $RTMin$. Finally, return the value of the optimal solution $RTMin$ (Lines 13-17).

4.4. **Greedy-based task offloading strategy.** The Greedy Algorithm [26] is one of the comparison algorithms we selected. In this algorithm, the optimal solution is chosen at each iteration step. In this study, during each iteration, the edge server with the longest

Algorithm 3: A process for generating task offloading plan based on RMA

Input: E and $RTnum$
Output: $RTMin$
Initialize: $RTSolutions = [\,]$
while $i < RTnum$ **do**
    Generate the random migration matrix S by **Algorithm 1**
    if $S \cdot type = bool$ **do**
        continue
    end if
    Add S to the solution set $RTSolutions$
    $i+ = 1$
end while
$RTMin = 999999$
for each $Solution$ in $RTSolutions$ **do**
    Calculate $Ttotal$ corresponding to $Solution$
    if $Ttotal < RTMin$ **do**
        $RTMin = Ttotal$
    end if
end for
return $RTMin$

response time is selected as the task migration source, and the adjacent edge server with the shortest response time is selected as the task migration destination. Then, a certain proportion of tasks is offloaded from the migration source to the destination. Continuous iterations return the obtained solution once the maximum response time reaches the predetermined threshold. A maximum number of iterations is set to prevent the algorithm from entering an infinite loop due to an overly small threshold, resulting in continuous task migration without reaching the threshold. If the number of iterations exceeds this maximum, the current solution is returned directly. The task scheduling strategy based on the Greedy Algorithm is shown in Algorithm 4.

Algorithm 4's input is the set of edge servers E, the range of determination thresholds $rtMin$ and $rtMax$, the network transmission time $Distance$, and the output is the maximum response time corresponding to the optimal solution.

Step 1: Generate the zero matrix as the basis of the solution, calculate the set of response times rt_set and and the remaining accommodating tasks $lefts$ for each edge server. Compare the value in the set rt_set with the threshold value rt in this round; if the response times of all edge servers are less than rt, add the current solution to the solution set and complete the generation of the solution for this threshold value (Lines 3-17).

Step 2: If the first step is not finished, take the edge server with the longest response time from rt_set as the output end, record its number in $OutputEdge$, then take the edge server with the smallest response time from the set of neighboring edges of the edge $OutputEdge$ as the task relocation end, record its number in $OutputEdge$ and record its number in delta as the relocation out then judge the edge server $InputEdge$. Take the remaining migration capacity of the edge server $InputEdge$ by a certain ratio to whether it is enough to receive $rtdelta$ tasks; if it is enough, then carry out the reception and update the $lefts$ set and rt_set set, otherwise add the current solution to the solution set $Solutions$, and end the cycle of this round of thresholding at the same time (Lines 18-24).

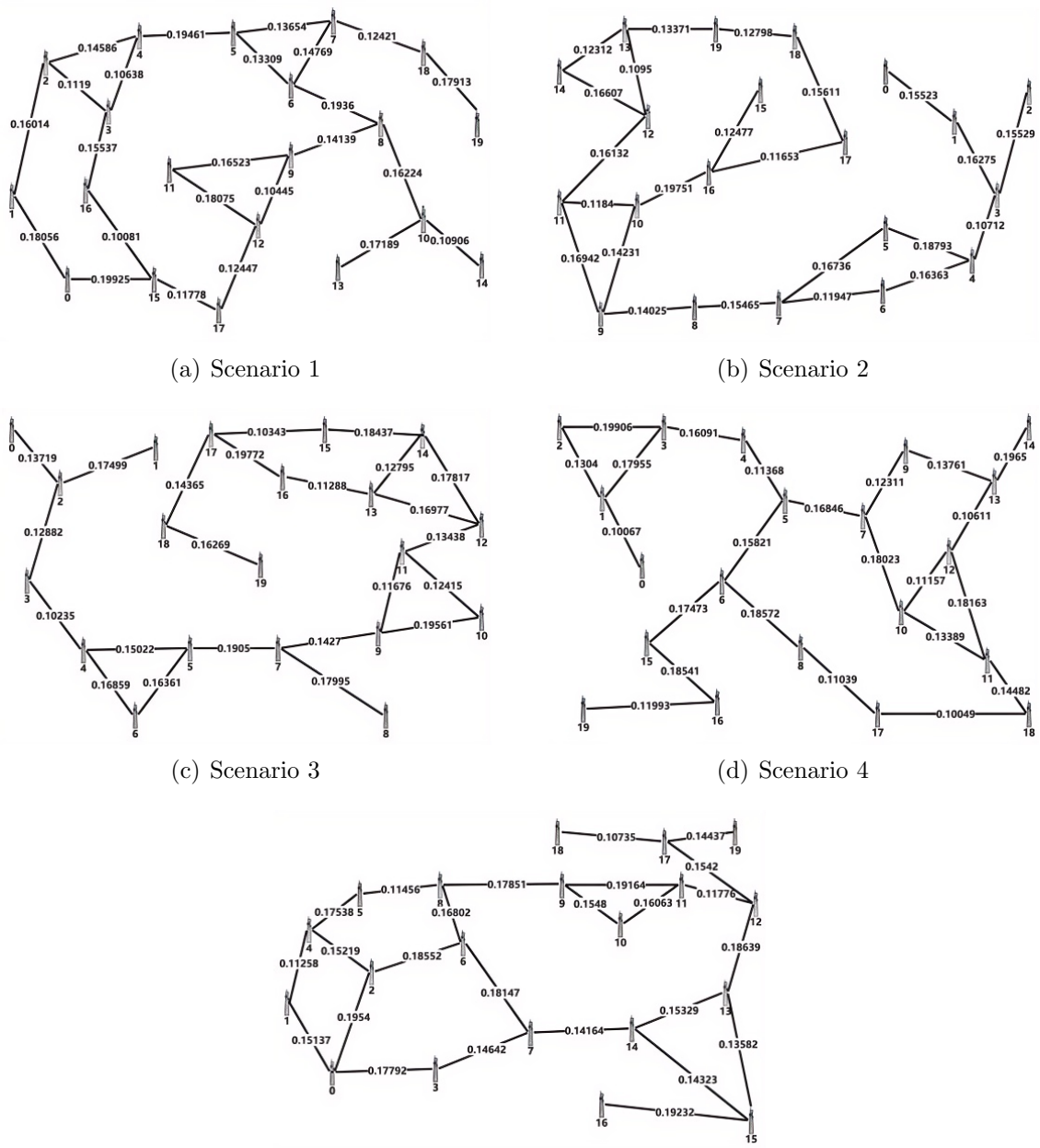
Algorithm 4: A process for generating task offloading plan based on Greedy Algorithm

1: **Input:** E, *rtMin*, *rtMax* and *Distance*
2: **Output:** *GAMin*
3: **Initialize:** $Solutions = [\,]$, generate the zero matrix $Solution \in R^{N\times N}$
4: Calculate the response time of each edge server to form the response time set *rt_set*
5: Calculate each edge server's maximum task migration capacity to form set *lefts*.
6: $rt = rtMin$
7: **while** $rt < rtMax$ **do**
8: **for** $i = 0$ to $i = 20000$ **do**
9: $Judge = True$
10: **for** *rt_set_num* in *rt_set* **do**
11: **if** $rt\_set\_num > rt$ **do**
12: $Judge = False$
13: **end if**
14: **end for**
15: **if** $Judge = True$ **do**
16: Save *Solution* to *Solutions*
17: break
18: **end if**
19: Get the index of the maximum value in *rt_set* and mark it as *OutputEdge*
20: Obtain the edge server with the shortest response time from the adjacent edge server set L of the edge numbered *OutputEdge*, and label its index as *InputEdge*
21: Calculate the task processing rate $\mu_{OutputEdge} * 0.03$ for the edge corresponding to *OutputEdge*, which represents the amount of task migration and stores it in *delta*
22: **if** $lefts[InputEdge] < delta$ **do**
23: Save *Solution* to *Solutions*
24: break
25: **else do**
26: $Solution[OutputEdge][InputEdge] += delta$
27: **end if**
28: Update the task response times of the input and output edge servers.
29: $lefts[OutputEdge] += delta$
30: $lefts[InputEdge] += delta$
31: **end for**
32: $rt += 0.1$
33: reset the sets *lefts* and *rt_set*
34: **end while**
35: Iteratively calculate the maximum response time corresponding to each solution in *Solutions*, and store the minimum value in *GAMin*.
36: **return** *GAMin*

Step 3: The solutions in the *Solutions* are tested for maximum response time, and the smallest of them and their corresponding solutions are saved to return the smallest maximum response time (Lines 35-36).

5. **Performance Evaluation.**

5.1. **Simulation setup.** We utilize the Shanghai Telecom base station dataset[1], which provides detailed information about base stations and Internet access across Shanghai [27-29]. For our simulation experiments, we randomly select several edge servers within the latitude and longitude range of $(31.1, 121.5)$ to $(31.2, 121.5)$. We construct five load balancing scenarios involving multiple edge servers, as shown in Figure 2. Specifically, we consider three scenarios with varying numbers of edge servers: 5, 10, and 15. Each scenario is as follows: edge servers numbered 0-4 for the 5-server scenario, edge servers numbered 0-10 for the 10-server scenario, and edge servers numbered 0-14 for the 15-server scenario.



(a) Scenario 1

(b) Scenario 2

(c) Scenario 3

(d) Scenario 4

(e) Scenario 5

FIGURE 2. Load balancing scenarios

[1]http://sguangwang.com/TelecomDataset.html

The number of arrival tasks, the computing power of edge servers, and the transmission time between edge servers follow normal distribution. Meanwhile, in order to prevent the tasks on the edge servers from entering an infinite queue, we control the number of arrival tasks to be lower than the computing power of edge servers and satisfy the following constraints:

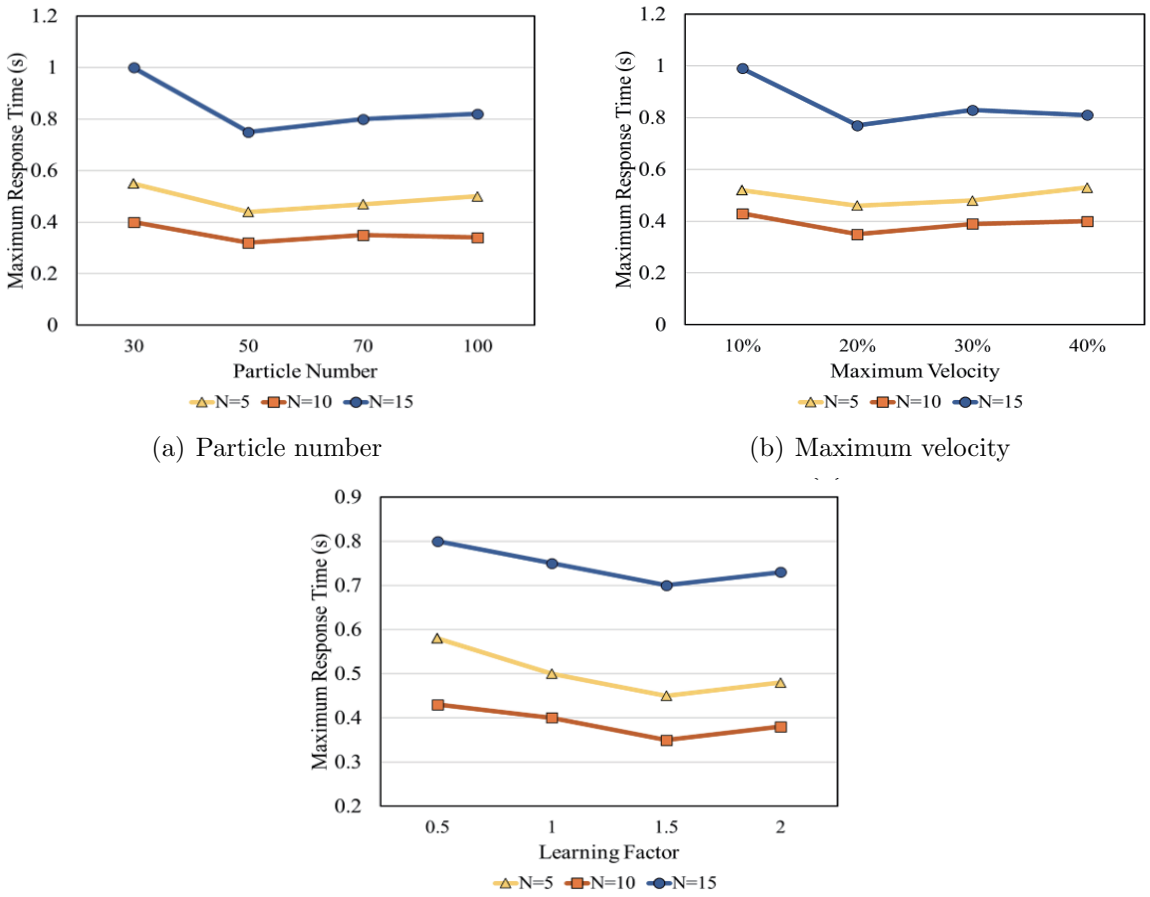
$$\mu_i - \lambda_i \geq 0.2 \tag{18}$$

According to [11], we set the specific parameters of the MEC environment in Table 2.

TABLE 2. Parameter setting

Parameter	Value
Number of edge servers N	$\{5, 10, 15\}$
Computing power μ_i	$N(15, 6)$
Number of arrival tasks λ_i	$N(10, 4)$
Maximum number of edge server neighbor sets K	3
Transmission time d_{ij}	$[0.1, 0.2]$

We conducted a sensitivity analysis of the key parameters of the ESLB-PSO algorithm, as shown in Figure 3. Specifically, Figure 3(a) demonstrates the effect of particle number on the maximum response time, where the particle number is set to 30, 50, 70, and 100,



(a) Particle number

(b) Maximum velocity

(c) Learning factor

FIGURE 3. Sensitivity analysis of critical parameters of ESLB-PSO in different scenarios

respectively. The result shows that the maximum response time of the system is lowest when the particle number is set to 50. Figure 3(b) demonstrates the effect of maximum velocity on maximum response time where maximum velocity is set to 10%, 20%, 30% and 40% of the range of decision variables, respectively. The result shows that the maximum response time of the system is lowest when the maximum speed is set to 20% of the decision variable range. Figure 3(c) demonstrates the effect of learning factor on maximum response time, where learning factor is set to 0.5, 1, 1.5 and 2, respectively. The result shows that the maximum response time of the system is lowest when the learning factor is set to 1.5. Based on the above results, we finally selected the parameters of ESLB-PSO as 50 particles, 20% of the maximum velocity of the decision variable range, and the learning factor of 1.5, which resulted in the optimal performance.

To ensure the fairness and reasonableness of the experiments, the proposed ESLB-PSO and baselines are implemented in Python 3.10 environment, running on a system with AMD Ryzen 7 5800H with Radeon Graphics CPU and 16GiB RAM. We compare the proposed ESLB-PSO with the following baselines:

- Random Migration Algorithm (RMA) [22]: As described in Section 4.3, a Random Migration Algorithm is employed for task offloading between edge servers.
- Greedy Algorithm (Greedy) [23]: As described in Section 4.4, a Greedy Algorithm is employed for task offloading between edge servers.
- Differential Evolutionary Algorithm (DEA) [17]: The method utilizes variation, crossover, and selection mechanisms to optimize the task offloading plan by vectorial difference variation to improve the load balancing optimization efficiency while maintaining the global search capability.
- Genetic Algorithm (GA) [20]: The method iteratively optimizes the task offloading matrix through selection, crossover and mutation operations to minimize the maximum task response time of the edge servers and balances the global search with local optimization during population evolution.
- Q-learning [13]: The method learns the state-action value (i.e., Q-value) of task offloading decisions, and gradually optimizes task allocation based on the ε-greedy policy to improve load balancing performance.
- DQN [14]: The method uses neural networks to approximate the Q-value function and performs deep reinforcement learning in a high-dimensional task offloading state space to maximize the long-term cumulative rewards in order to optimize the task scheduling policy of edge servers.

5.2. **Effectiveness of the proposed ESLB-PSO.** To evaluate the effectiveness of the proposed ESLB-PSO, we tested the effectiveness of adjusting the load rate in different numbers of edge servers, as shown in Tables 3, 4, and 5. The ideal plan in the tables results from constantly searching and approximating the optimal performance by trying and evaluating many load balancing scenarios in different scenarios. As shown in Tables 3, 4, and 5, our proposed ESLB-PSO can load adjust in various initial states under different numbers of edge servers and finally obtains a load balancing state close to the ideal plan. In addition, although the ideal plan can obtain the optimal solution with a small number of edge servers, it cannot arrive at a valid solution at $N = 15$. This is because the ideal plan, which is an optimal solution derived by continuously trying all the scenarios, has a great algorithmic complexity and thus is unrealistic when facing larger-scale scenarios. In contrast, our proposed ESLB-PSO can effectively face MEC environments of different scales with better adaptability and tuning efficiency.

In addition, we tested the maximum task response time difference between the proposed ESLB-PSO and the ideal plan for different numbers of edge servers, and the results are

TABLE 3. ESLB-PSO adjustment effect on load rate at $N = 5$ (%)

Scenario	**Edge server (e_i)**	e_0	e_1	e_2	e_3	e_4
1	Initial state	63	65	81	69	69
	Ideal plan	71	68	72	70	69
	ESLB-PSO	73	70	74	72	70
2	Initial state	70	86	75	81	62
	Ideal plan	75	71	76	74	75
	ESLB-PSO	80	75	78	78	79
3	Initial state	65	76	61	88	89
	Ideal plan	76	73	70	75	76
	ESLB-PSO	78	74	74	76	80
4	Initial state	71	62	71	86	88
	Ideal plan	76	74	74	76	79
	ESLB-PSO	78	76	76	78	82
5	Initial state	62	79	67	88	63
	Ideal plan	73	67	72	72	71
	ESLB-PSO	74	72	73	76	77

TABLE 4. ESLB-PSO adjustment effect on load rate at $N = 10$ (%)

Scenario	**Edge server (e_i)**	e_0	e_1	e_2	e_3	e_4	e_5	e_6	e_7	e_8	e_9
1	Initial state	69	74	74	89	62	77	67	65	88	62
	Ideal plan	73	75	78	75	76	76	71	66	71	73
	ESLB-PSO	80	79	81	78	79	77	75	73	75	76
2	Initial state	80	77	62	83	71	73	71	87	81	79
	Ideal plan	74	76	77	74	76	74	73	77	70	76
	ESLB-PSO	77	80	81	78	82	76	75	81	80	73
3	Initial state	87	74	64	63	67	81	89	73	66	83
	Ideal plan	74	70	73	83	70	75	75	72	73	70
	ESLB-PSO	75	76	81	85	73	79	77	78	74	76
4	Initial state	83	80	76	87	84	63	82	67	76	76
	Ideal plan	76	78	72	76	76	79	76	65	81	73
	ESLB-PSO	77	79	75	81	77	81	81	68	84	75
5	Initial state	68	63	86	61	78	85	84	80	83	61
	Ideal plan	76	77	70	74	75	76	74	76	75	76
	ESLB-PSO	80	81	73	76	76	80	76	78	76	77

averaged over several scales, as shown in Figure 4. The difference in the maximum response time of our proposed ESLB-PSO compared to the ideal plan always stays below 7%. This indicates that compared to the ideal plan obtained by spending a lot of search time, ESLB-PSO can make appropriate task offloading actions according to the current system state, and take account of the task's need to be sensitive to the completion delay based on the final performance approaching the ideal plan.

5.3. **Performance improvement of the proposed ESLB-PSO.** To evaluate the proposed ESLB-PSO's performance improvement, we tested its maximum task response time

TABLE 5. ESLB-PSO adjustment effect on load rate at $N = 15$ (%)

Scenario	**Edge server (e_i)**	e_0	e_1	e_2	e_3	e_4	e_5	e_6	e_7	e_8
1	Initial state	74	63	62	70	69	75	84	80	60
	Ideal plan	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
	ESLB-PSO	71	73	78	77	70	79	79	78	70
2	Initial state	78	60	62	78	81	89	87	72	75
	Ideal plan	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
	ESLB-PSO	78	79	74	74	75	73	72	80	75
3	Initial state	83	84	74	64	62	66	85	62	74
	Ideal plan	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
	ESLB-PSO	73	77	74	79	74	76	75	73	80
4	Initial state	85	87	68	71	72	85	77	76	69
	Ideal plan	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
	ESLB-PSO	72	71	78	79	80	74	71	76	74
5	Initial state	89	60	62	78	83	84	81	63	70
	Ideal plan	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
	ESLB-PSO	73	63	62	74	80	82	73	65	70
Scenario	**Edge server (e_i)**	e_9	e_{10}	e_{11}	e_{12}	e_{13}	e_{14}			
1	Initial state	87	70	86	71	85	64			
	Ideal plan	N/A	N/A	N/A	N/A	N/A	N/A			
	ESLB-PSO	75	79	75	71	72	77			
2	Initial state	69	89	88	69	70	75			
	Ideal plan	N/A	N/A	N/A	N/A	N/A	N/A			
	ESLB-PSO	75	77	74	79	76	76			
3	Initial state	74	74	88	82	69	62			
	Ideal plan	N/A	N/A	N/A	N/A	N/A	N/A			
	ESLB-PSO	74	76	73	72	74	77			
4	Initial state	70	86	67	70	72	63			
	Ideal plan	N/A	N/A	N/A	N/A	N/A	N/A			
	ESLB-PSO	72	72	79	72	72	70			
5	Initial state	81	67	78	63	63	85			
	Ideal plan	N/A	N/A	N/A	N/A	N/A	N/A			
	ESLB-PSO	74	71	76	76	78	78			

compared to the baselines for different numbers of edge servers, as shown in Figures 5, 6, and 7.

Although RMA can optimize the maximum response time to some extent by generating a large number of random results, its optimization capability still needs to be improved. The main drawback of RMA is that its results could be more stable, and the results may be very different after several runs. In some cases, there may even be no optimization effect. This is because RMA relies on a completely random search strategy and needs more systematic guidance, making it difficult to function stably in complex load balancing problems. Therefore, RMA performs as the least effective of the three algorithms.

Compared with RMA, Greedy can select the optimal migration scheme at each step, so its optimization effect and stability are significantly better than that of RMA. However,

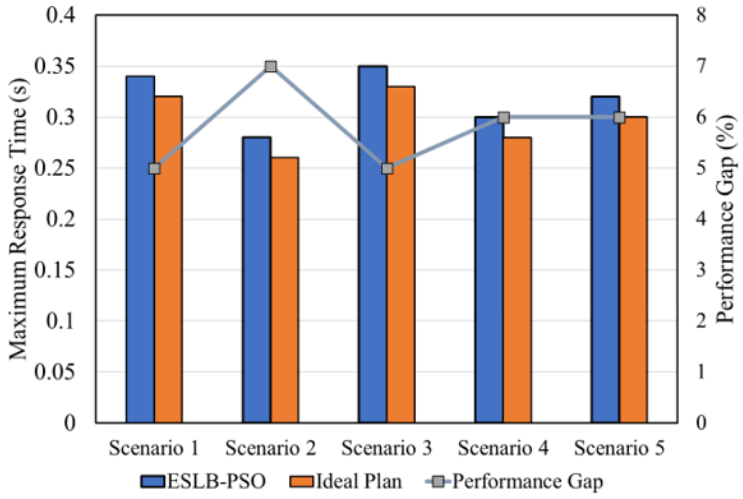


FIGURE 4. Maximum response time comparison between ESLB-PSO and ideal plan in different scenarios

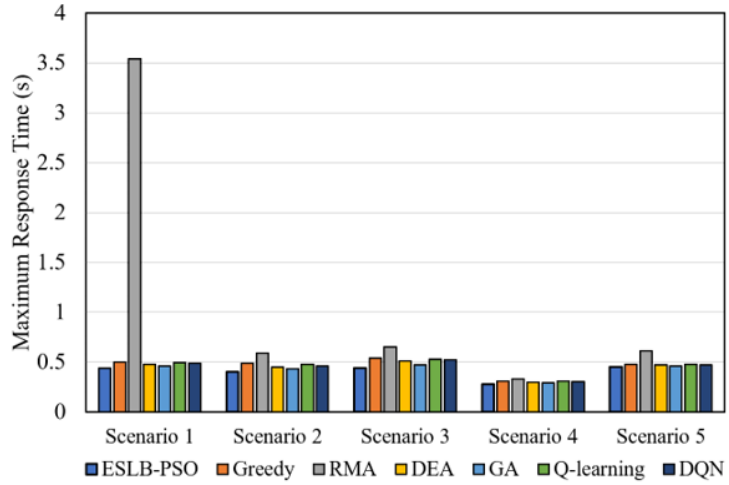


FIGURE 5. Maximum response time comparison between ESLB-PSO and baselines at $N = 5$ in different scenarios

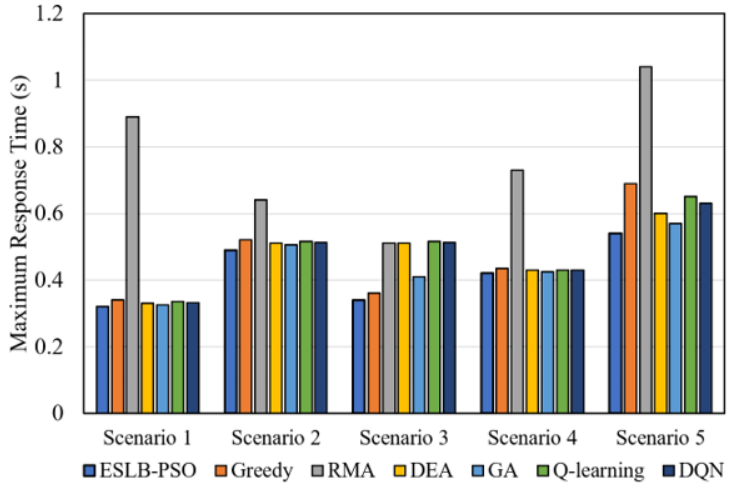


FIGURE 6. Maximum response time comparison between ESLB-PSO and baselines at $N = 10$ in different scenarios

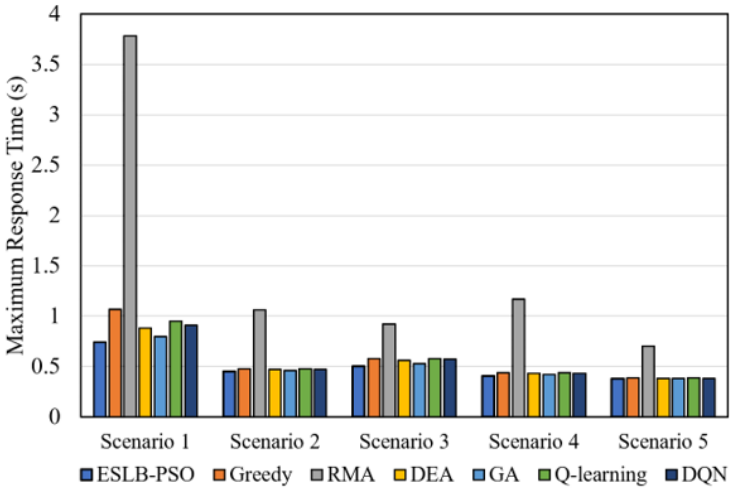


FIGURE 7. Maximum response time comparison between ESLB-PSO and baselines at $N = 15$ in different scenarios

Greedy has a very high dependence on the setting of the threshold, and whether the threshold is set reasonably or not will greatly affect the efficiency and results of the algorithm. In addition, when the threshold is not set reasonably, the search process of Greedy may be limited, resulting in its inability to give full play to its optimization capability. When an edge server is adjacent to only one edge server and the response time of both servers is high, tasks may keep migrating back and forth between these two servers, causing the algorithm to stagnate. Although this problem is mitigated in this paper by limiting the maximum number of passes, it still cannot be avoided completely. Therefore, although the results and optimization stability of Greedy are better than that of RMA, it still has some limitations.

Q-learning is a reinforcement learning based method to learn task offloading decisions by continuously exploring and updating Q-values. Although Q-learning is able to adaptively optimize the offloading decision, it faces the problem of excessive state space dimensionality in MEC environments. Q-learning usually requires offline training and a lot of empirical playback, which results in its slower convergence in high-dimensional and dynamic environments, and is unable to adapt to the system's load changes in a timely manner. In addition, the training process of Q-learning may require a lot of time and data to learn an effective policy. Therefore, it is limited in task offload timeliness and global optimization capability.

DQN is an extension of Q-learning, which utilizes deep neural networks to approximate the Q-value function to deal with high-dimensional state space. DQN can capture complex environmental information, which is suitable for high-dimensional state space problems in task offloading, and can improve the efficiency of policy learning to a certain extent. However, DQN still has some problems in practical applications. First, the training process of DQN requires a large number of samples and a long training time, and the convergence may be unstable in some environments. In addition, due to the dynamic nature of MEC networks, DQN may not be able to adapt to the changes in network load in time when facing real-time task offloading, which affects their performance. Therefore, despite the theoretical capability of DQN, there are still challenges in practical applications, resulting in its performance being slightly inferior to that of heuristic-based algorithms.

DEA relies on random search and variance operations to have global search capability when solving load balancing problems. DEA can explore a wider solution space through differential variation and crossover operations and improve the probability of

finding a globally optimal solution. However, DEA has relatively low search efficiency in high-dimensional spaces and under complex constraints and is prone to falling into local optimality. This is because the complexity and computation of the solution space increase significantly with the increase of dimension, and the variation and crossover operations of DEA may only be able to cover some of the solution space effectively, leading to local convergence problems during the search process. Nevertheless, DEA can still show better optimization results and stability when dealing with low-dimensional and medium-complexity problems.

GA can effectively find a better task offloading plan by performing global optimization through selection, crossover and mutation operations. Similar to DEA, GA adopts a population evolution strategy to optimize the task offloading plan by simulating the process of natural selection. The advantage of GA is that it can avoid falling into local optima and can handle more complex task offloading problems. However, GA may encounter the problem of slow convergence in the solution process, especially when the search space is large, and more iterations may be needed to find the optimal solution. Despite its relative slowness, GA is able to consistently give better task offloading plan in practical applications, and thus it is ranked higher than DEA.

Compared to RMA, Greedy, and DEA, our proposed ESLB-PSO exhibits lower maximum response time and superior optimization-seeking performance. This advantage stems from the dynamic adjustment strategy we implement for the inertia weights ω during the PSO iterations. At the beginning of the iteration, the larger inertia weight encourages the particles to rely more on their inertia direction for extensive exploration, which effectively broadens the search range and thus enhances the potential of the algorithm to discover the global optimal solution. As the number of iterations increases, the inertia weights gradually decrease, accelerating the convergence of the particle swarm to the global optimal solution and ensuring that the algorithm can lock the optimal solution more quickly.

In addition, we tested the iterative convergence of ESLB-PSO, GA, DEA, Greedy and RMA, as shown in Figure 8. ESLB-PSO decreases rapidly at the beginning of the iteration and has dropped to the more optimal performance at 400 iterations, and finally its performance stabilizes at more than 800 iterations, which shows its excellent global search capability and local fine-tuning mechanism. In contrast, although GA can also reduce the better performance faster, the decline curve is relatively smooth, and the performance

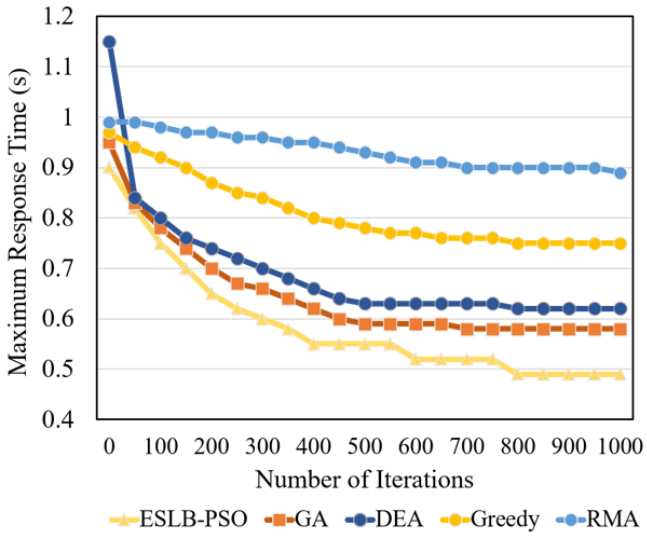


FIGURE 8. Convergence performance of the ESLB-PSO with different methods

stabilizes after 1000 iterations, which indicates that although GA can explore the better solution in the process of selection, crossover and mutation, its stochastic nature makes the local search not precise enough. The convergence trend of DEA is similar to that of GA, with a fast decline at the initial stage, but stabilizes at about 500 iterations, which reflects its higher sensitivity to the parameters in the global search. Greedy has a slow convergence process because it only considers the local optimal decision at each step, indicating that it is difficult to achieve the optimal solution of overall load balancing by solely relying on local information. Overall, the fast convergence and low final objective value of ESLB-PSO fully proves its advantages of efficient global exploration and fine optimization through dynamic parameter tuning in solving the MEC task offloading problem, which significantly outperforms the baselines such as GA, DEA and Greedy.

6. **Conclusions.** In this paper, we propose a method based on the PSO to address the load balancing of edge servers in MEC environments. In this method, the particles of PSO are conceptualized as the solution space matrix of the problem, and the iterative search process through the population of particles aims to uncover a better task offload solution. A comparison of experimental data shows that the proposed ESLB-PSO exhibits significant advantages in reducing the maximum response time of tasks and improving the efficiency of load balancing among edge servers compared to the baselines.

Acknowledgment. This work is partially supported by "Financial Research General Funding Project of Fujian Province, No. 2023CZ50" and "Innovation Strategy Research Plan Project of Fujian Province, No. 2023R0034". The authors also gratefully acknowledge the helpful comments and suggestions of the reviewers, which have improved the presentation. In the future, we will further explore multi-objective optimization (e.g., trade-off between energy consumption and response time) and load balancing strategies in heterogeneous edge server environments, and also consider incorporating security mechanisms (e.g., federated learning) to enhance data privacy protection during task offloading.

REFERENCES

[1] A. S. Ahmad, H. Kahtan, Y. I. Alzoubi, O. Ali and A. Jaradat, Mobile cloud computing models security issues: A systematic review, *Journal of Network and Computer Applications*, vol.190, pp.103152-103168, 2021.

[2] S. Tian, C. Chang, S. Long, S. Oh, Z. Li and J. Long, User preference-based hierarchical offloading for collaborative cloud-edge computing, *IEEE Transactions on Services Computing*, vol.16, no.1, pp.684-697, 2023.

[3] M. Iorio, F. Risso and C. Casetti, When latency matters: Measurements and lessons learned, *ACM SIGCOMM Computer Communication Review*, vol.51, no.4, pp.2-13, 2021.

[4] Y. Siriwardhana, P. Porambage, M. Liyanage and M. Ylianttila, A survey on mobile augmented reality with 5G mobile edge computing: Architectures, applications, and technical aspects, *IEEE Communications Surveys & Tutorials*, vol.23, no.2, pp.1160-1192, 2021.

[5] S. Khan, J. Zheng, S. Khan, Z. Masood and M. P. Akhter, Dynamic offloading technique for real-time edge-to-cloud computing in heterogeneous MEC-MCC and IoT devices, *Internet of Things*, vol.24, pp.100996-101028, 2023.

[6] W. Strielkowski, M. Dvořák, P. Rovný, E. Tarkhanova and N. Baburina, 5G wireless networks in the future renewable energy systems, *Frontiers in Energy Research*, vol.9, pp.714803-714817, 2021.

[7] A. Mohamed, M. Hamdan, S. Khan, A. Abdelaziz, S. F. Babiker, M. Imran and M. N. Marsono, Software-defined networks for resource allocation in cloud computing: A survey, *Computer Networks*, vol.195, pp.108151-108172, 2021.

[8] C. Jiang, T. Fan, H. Gao, W. Shi, L. Liu, C. Cérin and J. Wan, Energy aware edge computing: A survey, *Computer Communications*, vol.151, pp.556-580, 2020.

[9] R. Kemp, N. Palmer, T. Kielmann and H. Bal, Cuckoo: A computation offloading framework for smartphones, *Proc. of the 12th Conf. on Mobile Computing, Applications, and Services*, Berlin, Heidelberg, 2012.

[10] E. Cuervo, A. Balasubramanian, D. Cho, A. Wolman, S. Saroiu, R. Chandra and P. Bahl, MAUI: making smartphones last longer with code offload, *Proc. of the 8th Conf. on Mobile Systems, Applications, and Services*, pp.49-62, 2010.

[11] M. Jia, J. Cao and W. Liang, Optimal cloudlet placement and user to cloudlet allocation in wireless metropolitan area networks, *IEEE Transactions on Cloud Computing*, vol.5, no.4, pp.725-737, 2017.

[12] L. Ma, J. Wu, L. Chen and Z. Liu, Fast algorithms for capacitated cloudlet placements, *Proc. of the IEEE 21st International Conf. on Computer Supported Cooperative Work in Design*, Wellington, New Zealand, pp.439-444, 2017.

[13] Z. Du, C. Peng, T. Yoshinaga and C. Wu, A Q-learning-based load balancing method for real-time task processing in edge-cloud networks, *Electronics*, vol.12, no.15, pp.3254-3573, 2023.

[14] Z. Li, K. Yu, H. Zhou and X. Wu, DQN-based collaborative computation offloading for edge load balancing, *Proc. of the IEEE 8th International Conference on Network Intelligence and Digital Content*, Beijing, China, pp.1-6, 2023.

[15] S. Kosta, A. Aucinas, P. Hui, R. Mortier and X. Zhang, ThinkAir: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading, *Proc. of the 2012 IEEE INFOCOM*, Orlando, FL, USA, pp.945-953, 2012.

[16] R. Somula and R. Sasikala, A load and distance aware cloudlet selection strategy in multi-cloudlet environment, *International Journal of Grid and High Performance Computing*, vol.11, no.2, pp.85-102, 2019.

[17] Q. Zhang, L. Gui, F. Hou, J. Chen, S. Zhu and F. Tian, Dynamic task offloading and resource allocation for mobile-edge computing in dense cloud RAN, *IEEE Internet of Things Journal*, vol.7, no.4, pp.3282-3299, 2020.

[18] Z. Xu, W. Liang, W. Xu, M. Jia and S. Guo, Capacitated cloudlet placements in wireless metropolitan area networks, *Proc. of the IEEE 40th Conf. on Local Computer Networks*, Clearwater Beach, USA, pp.570-578, 2015.

[19] L. Yang, H. Yao, J. Wang, C. Jiang, A. Benslimane and Y. Liu, Multi-UAV-enabled load-balance mobile-edge computing for IoT networks, *IEEE Internet of Things Journal*, vol.7, no.8, pp.6898-6908, 2020.

[20] M. Merah, Z. Aliouat and H. Mabed, Dynamic load balancing of traffic in the IoT edge computing environment using a clustering approach based on deep learning and genetic algorithms, *Cluster Computing*, vol.28, no.77, 2025.

[21] N. J. Gunther, Erlang Redux: An ansatz method for solving the M/M/m queue, *arXiv Preprint*, arXiv: 2008.06823, 2020.

[22] B. Ma, Y. Xu, Y. Pan and C. Li, A multi-user mobile edge computing task offloading and trajectory management based on proximal policy optimization, *Peer-to-Peer Networking and Applications*, pp.1-20, 2024.

[23] A. G. Gad, Particle swarm optimization algorithm and its applications: A systematic review, *Archives of Computational Methods in Engineering*, vol.29, no.5, pp.2531-2561, 2022.

[24] A. Pradhan, S. K. Bisoy and A. Das, A survey on PSO based meta-heuristic scheduling mechanism in cloud computing environment, *Journal of King Saud University – Computer and Information Sciences*, vol.34, no.8, pp.4888-4901, 2022.

[25] Z. Manzoor, M. T. A. Qaseer and K. M. Donnell, A comprehensive bi-static amplitude compensated range migration algorithm (AC-RMA), *IEEE Transactions on Image Processing*, vol.30, pp.7038-7049, 2021.

[26] Z. Zhao, M. Zhou and S. Liu, Iterated greedy algorithms for flow-shop scheduling problems: A tutorial, *IEEE Transactions on Automation Science and Engineering*, vol.19, no.3, pp.1941-1959, 2022.

[27] Y. Li, A. Zhou, X. Ma and S. Wang, Profit-aware edge server placement, *IEEE Internet of Things Journal*, vol.9, no.1, pp.55-67, 2022.

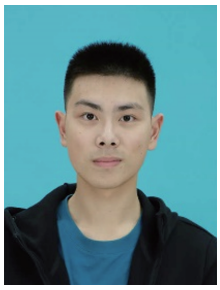
[28] Y. Guo, S. Wang, A. Zhou, J. Xu, J. Yuan and C. Hsu, User allocation-aware edge cloud placement in mobile edge computing, *Software: Practice and Experience*, vol.50, no.5, pp.489-502, 2020.

[29] S. Wang, Y. Guo, N. Zhang, P. Yang, A. Zhou and X. Shen, Delay-aware microservice coordination in mobile edge computing: A reinforcement learning approach, *IEEE Transactions on Mobile Computing*, vol.20, no.3, pp.939-951, 2021.

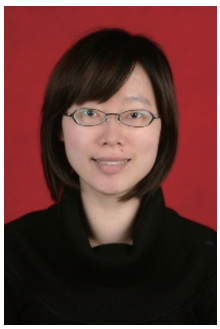
Author Biography

Xiong Yang received the B.S. degree in Automation and M.S. degree in Detection and Automation Equipment from Xiamen University, Xiamen, China, in 2007 and 2010, respectively. He is currently an Associate Professor at Fuzhou University Zhicheng College. He is also a Professional Member of the China Computer Federation (CCF). His research interests include edge computing, machine learning and computer vision.

Yucheng Shi received the B.S. degree from Sichuan University of Science and Engineering in 2024. He is currently pursuing the M.S. degree with the College of Computer and Data Science, Fuzhou University, Fuzhou, China. His research interests include edge computing, deep learning and computer vision.



Ruhui Chen is currently pursuing the B.S. degree at Fuzhou University Zhicheng College, Fuzhou, China. His research interests include deep learning and computer vision.



Huihua Xu received the M.S. degree from Xiamen University, Xiamen, China, in 2010. She is currently an Associate Professor at Concord University College Fujian Normal University, and is currently pursuing the Ph.D. degree at School of Economics, Fujian Normal University, Fuzhou, China. Her research interests include financial technology, financial big data and machine learning, financial security.