Ingénierie des Systèmes d'Information
Vol. 31, No. 6, June, 2026, pp. 1763-1785
Journal homepage: http://iieta.org/journals/isi

Enhanced Rat Swarm Optimization for Energy–Latency-Aware Task Offloading in Mobile Cloud Computing

Hadjer Fellah[1,2*], Chaker Mezioud[1], Hichem Rahab[3], Youcef Bezza[3], Aridje Mebarki[2], Maissoun Mordjane[2]

[1] LISIA Laboratory, University of Constantine 2 Abdelhamid Mehri, Constantine 25017, Algeria
[2] Department of Computer Science, Abbes Laghrour University, Khenchela 40004, Algeria
[3] ICOSI Laboratory, Department of Computer Science, Abbes Laghrour University, Khenchela 40004, Algeria

Corresponding Author Email: hadjer.fellah@univ-constantine2.dz

https://doi.org/10.18280/isi.310601

Received: 12 March 2026
Revised: 13 June 2026
Accepted: 21 June 2026
Available online: 30 June 2026

Keywords:
application partitioning, computation offloading, energy–latency optimization, mobile cloud computing, metaheuristic optimization, Rat Swarm Optimization

ABSTRACT

Mobile cloud computing (MCC) enables resource-constrained mobile devices to execute computation-intensive applications by transferring selected tasks to cloud servers. Effective computation offloading, however, requires a careful trade-off between device energy consumption and application response time, particularly under changing network and cloud conditions. Existing approaches often rely on fixed offloading costs or static graph partitioning, which limits their ability to adapt to heterogeneous execution environments. This study formulates task offloading as a bi-objective optimization problem and proposes an energy-aware Enhanced Rat Swarm Optimization (ERSO) algorithm for adaptive application partitioning. The method introduces a revised search strategy to improve the balance between exploration and exploitation while reducing premature convergence. ERSO was evaluated using 16-node and 50-node weighted call graphs across seven MCC scenarios, with cloud speedup factors ranging from 5 to 30 and network bandwidths ranging from 3 to 100 Mbps. Its performance was compared with local execution, full offloading, minimum-cut partitioning, the standard Rat Swarm Optimizer, binary Particle Swarm Optimization, and a binary-coded genetic algorithm. The results show that ERSO consistently identifies effective task allocations under different operating conditions. Under the most constrained setting, with a bandwidth of 3 Mbps and a cloud speedup factor of 5, ERSO reduced the composite fitness cost by 33% compared with local execution. Overall, the proposed method provides a robust mechanism for jointly reducing energy consumption and response time in dynamic MCC environments.

1. INTRODUCTION

The rapid growth of mobile computing is primarily driven by the spectacular increase in the number of mobile devices such as smartphones, smartwatches, laptops, and more. In 2026, the number of smartphones worldwide is estimated at 7.58 billion, while the number of smartphone users will reach 5.65 billion [1]. Users of these devices typically expect a high quality of service, particularly in terms of response time, energy efficiency, and storage costs. Reducing the energy consumption of mobile devices has become both a major user requirement and an important objective in addressing the global energy crisis and promoting environmental sustainability. Current estimates indicate that mobile applications collectively consume approximately 20.3 terawatt-hours of electricity annually, slightly below the annual electricity consumption of Ireland [2]. Striking a balance between these non-functional requirements demands careful attention to ensure both a seamless user experience and the preservation of our planet. Several strategies have been adopted by both academia and industry to address energy efficiency and performance in mobile devices, covering multiple facets:

•At the hardware level, techniques like dynamic voltage and frequency scaling (DVFS) help adjust processor power based on workload. Newer approaches, such as MetaDVFS, have shown up to 17% improvement in performance-per-watt by using metadata to guide frequency scaling [3]. Modern mobile chips also integrate specialized hardware accelerators like neural processing units (NPUs) alongside graphics processing units (GPUs) and digital signal processors (DSPs). These allow tasks such as AI inference to run on the most energy-efficient unit available [4, 5]. Storage technology has also evolved: universal flash storage has largely replaced embedded MultiMediaCard (eMMC) by enabling full-duplex data transfer and command queuing, which translates to faster speeds and better power efficiency [6]. Meanwhile, smarter cache hierarchies help reduce the energy cost of memory operations [7].

•At the software level, energy-aware operating systems coordinate power management across different layers. Research shows that optimizing across hardware, operating system, network, and applications together yields better

energy savings than tuning each layer separately [8]. Virtualization and containerization also help consolidate resources efficiently [7].

•User best practices, simple practices like lowering screen brightness, closing background apps, and following proper charging routines can extend battery life. A recent study [9] found that user awareness of these practices was a key factor in managing battery drain effectively.

•Finally, green energy harvesting offers a sustainable path forward. Systems that capture ambient energy from light, motion, or heat can power devices without relying solely on batteries. Light-based harvesters typically use solar cells, power management chips with Maximum Power Point Tracking, and small batteries or supercapacitors. The main challenges are dealing with inconsistent energy availability and ensuring that harvested power meets the device's average needs [10].

In this study, we focus on computation offloading (CO). While hardware improvements, software optimization, user practices, and green energy each contribute to mobile device energy efficiency, CO offers a uniquely flexible solution that can dynamically adapt to varying workloads and environmental conditions. Rather than executing all computations locally on the device, offloading them to external resources, specifically the cloud, can be an effective way to meet user requirements. Mobile cloud computing (MCC) has emerged as a transformative paradigm, enabling mobile devices to offload intensive computations to the cloud and leverage its virtually unlimited resources. This practice has become a key approach for enhancing both performance and energy efficiency in mobile environments [11-13].

The CO is a complex process that involves a three-step workflow [14]:

•Application partitioning: This step divides a given mobile user application into offloadable and nonoffloadable components, i.e., determining which components are executed inside the device and which need to be externalized to the cloud. It is a prerequisite for efficient CO.

•Preparation: This step handles all tasks required for the offloadable components in mobile applications, including selecting a remote server, transferring, and installing the code, among others.

•Offloading decision: Before remote execution, a decision is made based on the current execution context to determine whether the remote component will be used in the user's mobile application.

A thorough analysis of existing studies reveals the five key dimensions constituting the CO ecosystem (Figure 1): (D1) the highly dynamic nature of MCC, where device performance, bandwidth, and network latency fluctuate continuously, impacting the overall offloading cost; (D2) the underlying infrastructure of mobile devices including available memory, battery level, and CPU speed, as well as network specifications (such as 3G, Wi-Fi, and available bandwidth) that enable connectivity between devices and the cloud; (D3) user preferences, including data preferences; (D4) the CO algorithms responsible for application partitioning and decision-making; and (D5) the used non-functional requirements that CO algorithms have to satisfy. However, most existing CO algorithms do not incorporate the dynamic nature of the MCC environment. As a result, an effective offloading strategy should account for these fluctuations, optimizing energy consumption and resource usage in real time by adapting to changes in network conditions and service node capabilities, including the processing speed of both the mobile device and cloud servers. These challenges highlight the need for adaptive approaches that dynamically adjust offloading decisions based on current environmental conditions [15, 16].

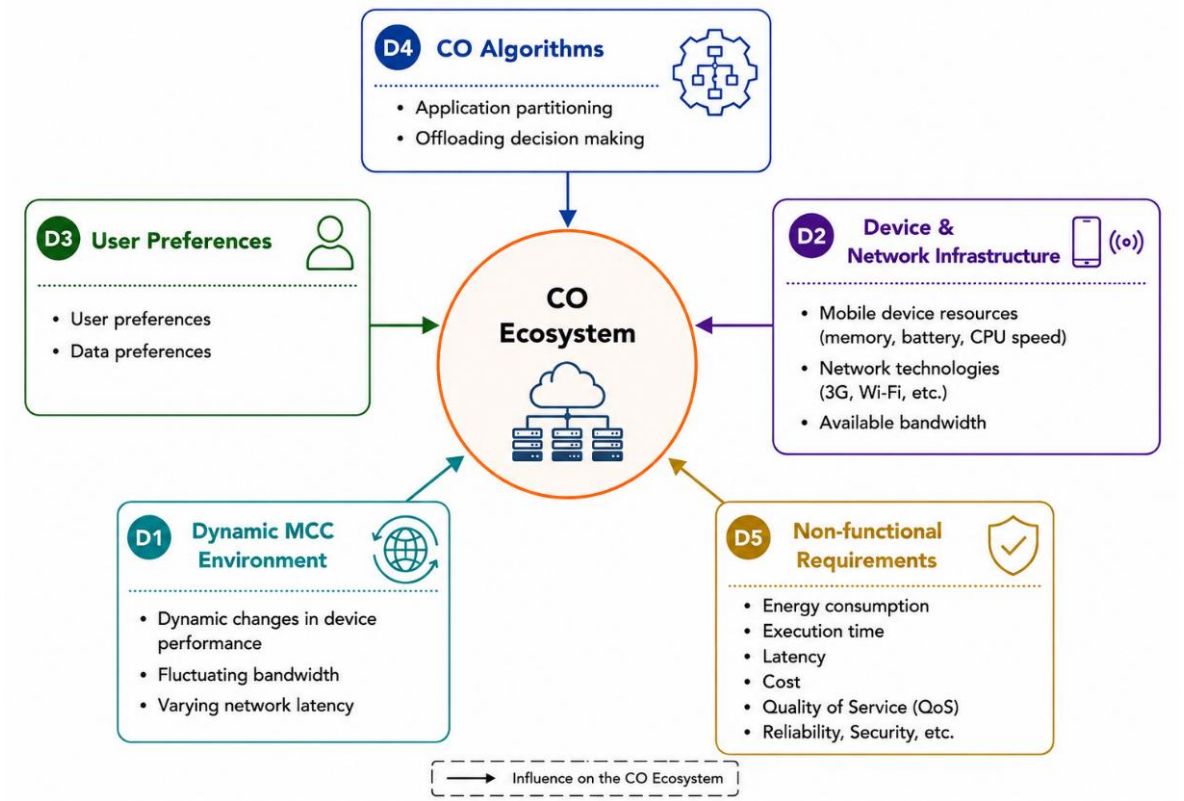


Figure 1. Five dimensions of the computation offloading (CO) ecosystem

Due to its importance, let us focus on application partitioning. To perform this CO step, the mobile user application can be formally modeled as a graph, where vertices represent computational components (annotated with metrics such as execution time, memory usage, and input/output), and edges represent communication costs between these components [15, 17]. The goal of graph partitioning is to determine the optimal allocation of components between the mobile device and cloud servers, minimizing communication costs while maximizing performance and energy savings. Achieving this goal is challenging, as the graph partitioning problem is known to be nondeterministic polynomial-time complete (NP-complete) [18].

Several algorithms have been proposed for CO-oriented graph partitioning, including heuristic and metaheuristic approaches. Among these, swarm intelligence-based algorithms, such as Ant Colony Optimization (ACO), Particle Swarm Optimization (PSO), and Whale Optimization Algorithm (WOA), have demonstrated promising results in balancing trade-offs between energy efficiency, processing power, and communication costs. However, these algorithms are static and do not account for the dynamic MCC environment [19]. Recently, the Rat Swarm Optimizer (RSO) has emerged as a novel swarm intelligence-based algorithm for solving global optimization problems. Inspired by the behavior of rats chasing and fighting their prey, RSO has shown significant success in continuous optimization tasks [20]. This study focuses on enhancing the RSO algorithm to increase its effectiveness for MCC. The objective is to modify RSO to efficiently manage the allocation of tasks between mobile devices and cloud resources, prevent entrapment in local optima, and reduce both energy consumption and execution time.

The remainder of this study is organized as follows: Section II reviews existing research on CO and optimization methods in MCC. Section III presents the cost function formulation for energy optimization. We also introduce the proposed Enhanced Rat Swarm Optimization (ERSO) algorithm, detailing how it adapts traditional RSO concepts to reduce energy consumption while maintaining efficient task completion. Section IV outlines the simulation setup and performance metrics used to evaluate our approach, followed by an in-depth discussion of the experimental results. Finally, Section V provides concluding remarks and suggests potential directions for future research on applying intelligent swarm-based optimization to MCC offloading problems.

2. LITERATURE REVIEW

CO tackles limitations of mobile devices such as limited battery lifetime, processing capabilities, and storage capacity by partitioning the computation-intensive applications, such as augmented reality, face and voice recognition, into components that are offloaded to remote rich Cloud servers, with better performance and resources, for remote execution. [12, 13, 21]. The potential of mobile offloading mainly depends on mobile network technologies such as cellular networks and Wi-Fi. Cellular network data transmission requires a considerable amount of energy from the mobile device, contrary to Wi-Fi, which can provide high bandwidth connections. Therefore, CO should be performed only when the local execution consumes more time and energy than the remote execution. An offloading algorithm is designed to make decisions that minimize the overall cost of running an application. The decision depends on whether to improve performance or save battery life, though these two goals often align. Unless network transmission uses significantly more power than computation, offloading typically boosts performance (faster execution) while saving battery life. Several MCC proposal works have focused on CO due to its energy consumption and execution time benefits. Categorized by the granularity and partition algorithm being used, the current work follows two main approaches [16]:

(1) Full migration, also referred to as coarse granulation or Application-level offloading [12, 22-24], is a software-as-a-service model where the entire process, application, program, or entire Virtual Machine is offloaded for remote execution on remote servers. The application's input data will be transmitted to the server when it is selected to be offloaded. The server will then execute the application and transmit its output data back to the mobile device [16, 25]. Every application is handled separately. The primary disadvantages of full migration are its rigidity and coarse granularity.

(2) Partial offloading, also referred to as fine-grained or task-level offloading [26, 27], involves dividing the application into inherited components [25]. With this approach, developers specify precisely how the application should be divided and which components should be offloaded. As well as how to modify the offloading plan in response to modifications in network conditions. By using this approach, applications can benefit from remote execution by offloading particular use-only subparts. Its main objective is enhancing mobile devices' energy efficiency [25, 28].

Remote execution offers the ability to improve energy efficiency and enhance the performance of programs running on weak devices [29, 30]. As mentioned above, CO employs partitioning in its mechanism for remote execution, dividing the functionality of mobile applications into distinct components capable of independent operation within a distributed environment [31]. Since not every application task can be performed remotely, they must be divided into two categories: one will be executed on the mobile terminal, and the other will be executed on cloud servers [25].

While researchers have explored heuristic, metaheuristic, and machine-learning solutions to CO, the application of SI techniques in application partitioning within MCC specifically remains an underexplored area compared to its use in cloud and edge computing.

Before analyzing specific CO algorithms, it is essential to classify them according to the five key dimensions of the CO ecosystem introduced in the previous section. Table 1 presents this classification, which guides the critical analysis that follows.

Existing research on CO has explored various optimization techniques to address mobile devices' energy and resource constraints, such as heuristic approaches and metaheuristic algorithms. These techniques can be classified into three categories: heuristics (deterministic, rule-based), metaheuristics (stochastic, nature-inspired), and hybrid algorithms (combining multiple strategies). Traditional heuristic methods such as Genetic Algorithms (GA) [32, 33] and Simulated Annealing (SA) [34, 35] have been widely used to partition tasks and optimize resource allocation in MCC. These methods demonstrate robust performance in static environments but are less effective in dynamic settings. Specifically, GA fails to satisfy D1 (dynamic network conditions) as it assumes stationary environments, while SA's

slow convergence violates D5 (latency requirements) for real-time offloading decisions. For instance, studies have combined optimization approaches [36-39] to create hybrid algorithms that address the dual objectives of energy consumption and execution time. On the other hand, metaheuristic approaches, inspired by natural and biological phenomena, have gained prominence in MCC offloading. PSO has been applied to optimize offloading decisions by minimizing energy and computational costs [40-43]. However, PSO is prone to premature convergence, especially in complex, dynamic scenarios. This premature convergence limits the ability to satisfy D1, as particles collapse to suboptimal solutions when bandwidth or latency fluctuates unexpectedly. Furthermore, PSO lacks memory across consecutive offloading decisions, forcing it to re-explore from scratch for each new task arrival [40-43]. ACO has also been utilized for resource allocation, leveraging pheromone-based communication to optimize task partitioning [44-46]. While effective, the high computational overhead of ACO limits its scalability. ACO's computational complexity scales quadratically with the number of tasks, failing to satisfy D2 (device resource constraints) and D5 (sub-second latency requirements). Additionally, the pheromone evaporation rate requires manual tuning: too fast loses memory of good solutions, too slow fails to adapt to D1 changes [42-44]. More recently, the WOA has shown promise in balancing performance and energy efficiency but requires careful parameter tuning for adaptability in dynamic environments [47]. WOA's parameter sensitivity (specifically the 'a' parameter that decreases from 2 to 0) requires prior knowledge of environmental variability, which is unavailable in real MCC scenarios, thus violating D1. Recent advances have introduced more sophisticated metaheuristics to CO. The Harris Hawks Optimization (HHO) algorithm, inspired by cooperative hunting behavior, has been applied to MEC offloading by Behera et al. [48], achieving an improvement in energy consumption compared to existing methods. However, HHO's performance degrades under D1 dynamics because its exploration-exploitation balance relies on a linearly decaying prey's escape-energy factor that cannot track unpredictable bandwidth fluctuations [48]. Yuan et al. [49] proposed the L´evy Flights and SA-based Grey Wolf Optimizer (LSAG), an enhanced GWO algorithm that combines L´evy flight and SA strategies to improve exploration capability and avoid premature convergence in CO optimization. Nevertheless, GWO's social hierarchy assumes static relationships among candidate solutions; when D1 conditions change, the algorithm cannot reassign its hierarchy without restarting the optimization process. Two hybrid metaheuristic approaches [50, 51], namely Genetic Simulated-annealing-based Particle Swarm Optimization (GSP) and its enhanced version Genetic Simulated-annealing-based Particle Swarm Optimization with Auto-Encoder (GSPAE), were proposed for CO in hybrid edge-cloud systems. By combining PSO, SA, and genetic operators, and further incorporating an Autoencoder in GSPAE, these methods achieve significant reductions in total system cost while satisfying latency and resource constraints. Nevertheless, the integration of multiple optimization mechanisms increases algorithmic complexity and computational requirements. Furthermore, both approaches rely on static optimization based on predefined system information and lack online adaptive learning capabilities, making them less suitable for highly dynamic mobile edge computing environments where network conditions, user mobility, and resource availability continuously evolve.

Table 1. Classification of computation offloading (CO) algorithms by type

Algorithm Type	Examples	Key Mechanism	Primary Limitation (CO Dimension)
Heuristic	GA [32, 33]; SA [34, 35]	Iterative improvement, rule-based search	Static assumption (D1)
Hybrid heuristic	GA-SA [36, 39]	Global exploration (GA) + local exploitation (SA)	High computational overhead (D2)
Metaheuristic (swarm intelligence)	PSO [40-43]; ACO [44-46]	Collective behavior, information sharing	Premature convergence; parameter sensitivity (D1)
Metaheuristic (biology-inspired)	WOA [47]; HHO [48]; GWO [49]	Hunting/foraging behavior mimicry	Parameter sensitivity; static hierarchy (D1)
Hybrid multi-strategy	GSP [50, 51]	GA + SA + PSO + autoencoder	Exceeds mobile CPU latency limits (D2, D5)

Note: SA: Simulated Annealing; GA: Genetic Algorithm; PSO: Particle Swarm Optimization; ACO: Ant Colony Optimization; WOA: Whale Optimization Algorithm; HHO: Harris Hawks Optimization; GWO: Grey Wolf Optimization; GSP: Genetic Simulated-annealing-based Particle Swarm Optimization.

Table 2. Comparison of existing algorithms against requirements for an ideal computation offloading (CO) algorithm

Algorithm	R1	R2	R3	R4	R5
GA	Assumes static; no adaptation to changing bandwidth	Heavy; large population and generations	No user preference mechanism	Premature convergence; loses diversity	No learning across runs; random start
SA	Assumes static; no real-time reaction	Lightweight; single solution only	No dynamic user input	Poor temp control; too random or too greedy	No memory; random start each run
PSO	Collapses to global best; cannot track changes	Lightweight; few parameters	No user input mechanism	Premature convergence; loses diversity early	Personal best only; no cross-run learning
ACO	Fixed evaporation; no adaptation to changes	Heavy; $O(n^2)$ complexity	Yes; weights encode preferences	Too random or too greedy	Pheromone decays; no long-term memory
WOA	Linear decay; assumes predictability	Lightweight; simple updates	No user input	Good balance in static cases	No memory; random Start
HHO	Linear escape energy; fails under change	Lightweight; similar to PSO	No user input	Good balance in static cases	No memory; independent

GWO	Static hierarchy; cannot reassign	Lightweight; simple updates	No user input	Good balance in static cases	Runs No memory; random Start
GSP	Offline only; no runtime adaptation	Very heavy	No dynamic user input	Excellent balance	Needs retraining; not real-time

Note: SA: Simulated Annealing; GA: Genetic Algorithm; PSO: Particle Swarm Optimization; ACO: Ant Colony Optimization; WOA: Whale Optimization Algorithm; HHO: Harris Hawks Optimization; GWO: Grey Wolf Optimization; GSP: Genetic Simulated-annealing-based Particle Swarm Optimization.

Table 3. Standard Rat Swarm Optimizer (RSO) requirements analysis

Requirements	Satisfied?	Mechanism
R1	√	Dynamic parameter adjustment
R2	√	Linear $O(n \times d)$ complexity
R3	×	Not included (gap to address)
R4	√	Adaptive search ratios
R5	√	Historical position retention

A critical synthesis of the literature indicates that existing methods, although insightful, frequently demonstrate limitations in the realm of MCC offloading, including premature convergence, sensitivity to parameter tuning, inadequate adaptability to dynamic network and device conditions, absence of memory-based learning from prior solutions, and insufficient mechanisms for balancing global exploration with local exploitation.

When mapped against the five CO dimensions, these limitations manifest as:

(1) Violation of D1 (dynamic adaptation): All existing algorithms assume static or slowly varying environments;

(2) Violation of D2 (device constraints): Hybrid algorithms like GSP exceed mobile CPU limitations;

(3) Violation of D3 (user preferences): No algorithm dynamically incorporates user-specified trade-offs between energy and latency;

(4) Violation of D5 (Non-Functional requirements satisfaction): Premature convergence leads to suboptimal energy consumption or unacceptable latency.

Given these limitations, an ideal CO algorithm for MCC should satisfy five requirements: (R1) adapt to D1 dynamics without restarting; (R2) maintain low computational overhead suitable for D2; (R3) optionally incorporate D3 user preferences; (R4) balance exploration and exploitation for D5; and (R5) include memory mechanisms to learn from past offloading decisions. Table 2 evaluates existing algorithms against these requirements.

As far as we know, no particular research examines the direct application of RSO to task partitioning in MCC offloading. Nonetheless, RSO has proven effectiveness in high-dimensional optimization problems. Unlike conventional SI algorithms, RSO addresses several of the limitations identified above. In contrast to conventional SI algorithms, RSO integrates adaptive mechanisms that equilibrate exploration (global search) and exploitation (local refinement). This equilibrium assists RSO in surmounting challenges such as premature convergence and suboptimal task allocation. Specifically, RSO satisfies R1 (adaptation to D1) through its dynamic parameter adjustment, R2 (low overhead for D2) through linear $O(n*d)$ complexity, R4 (exploration-exploitation balance) through adaptive search ratios, and R5 (memory mechanism) through historical position retention [20]. Regarding R3 (user preferences), this remains an open gap that our proposed enhancement addresses. Applying the proposed ERSO algorithm to MCC offloading is expected to improve energy efficiency, computational efficiency, and convergence speed (Table 3). The aim is to alter RSO to proficiently navigate dynamic situations, avoid entrapment in local optima, and diminish both energy expenditure and execution duration. The RSO incorporates:

•A fitness function that considers both local energy costs and communication energy costs, emphasizing energy awareness.

•Adaptive exploration and exploitation strategies are essential for achieving robust optimization in dynamic MCC scenarios.

•Implementation of memory-guided search and mutation mechanisms to prevent stagnation in local optima.

3. PROBLEM FORMULATION AND PROPOSED APPROACH

MCC presents several challenges, including variable network bandwidth, fluctuating server performance, and the limited resources of mobile devices, such as battery capacity, memory, and computational power [52]. These issues underscore the imperative for formulating efficient and flexible solutions for CO, which involves reallocating resource-intensive workloads from mobile devices to cloud servers [53]. A significant problem with existing approaches is their propensity to become trapped in local optima, resulting in unsatisfactory solutions. This issue occurs because numerous algorithms emphasize enhancing current solutions instead of sufficiently investigating fresh opportunities. Moreover, energy consumption continues to be a significant issue for mobile devices with limited battery life [19]. An efficient offloading approach must achieve a balance between performance and energy efficiency [54], including both processing and communication costs.

Before presenting the proposed approach, we formally define the system model and the optimization problem that our ERSO algorithm aims to solve.

3.1 System model

3.1.1 Application model

The mobile application is modeled as a directed graph $G = (V, E)$, where $V = \{v_1, v_2, ..., v_n\}$ represents the set of computational components (methods or tasks), and $E = \{(v_i, v_j)\}$ represents the communication dependencies between these components. Each vertex v_i has associated costs: L_i for local execution on the mobile device and Ri for remote execution on the cloud server. Each edge (v_i, v_j) has an associated communication cost C_{ij}, which represents the data transfer overhead if methods v_i and v_j are executed on different nodes. The goal of application partitioning is to divide the graph G into two disjoint subsets: VL (methods executed locally on the mobile device) and VR (methods offloaded to the cloud server), such that the total cost is minimized while satisfying system constraints.

3.1.2 Computation model

The execution time for methods executed locally on the mobile device is given by:

$$T_{Local} = \sum_{i \in L} t_i^{Local}$$

where, L represents the set of methods executed locally, and t_i^{Local} is the execution time for method i on the mobile device.

The execution time for methods offloaded to the cloud server is given by:

$$T_{Remote} = \sum_{i \in R} t_i^{local} / F$$

where, R represents the set of methods offloaded to the cloud, and F is the speedup factor representing the computational advantage of the cloud server over the mobile device. Typical values for F range from 5 to 50 depending on the cloud server type [14, 55, 56].

The energy consumption for local execution is calculated as:

$$E_{Local} = \sum_{i \in L} (t_i^{Local} * P_L)$$

where, P_L is the power consumption during local execution.

The energy consumption for remote execution is calculated as:

$$E_{Remote} = \sum_{i \in R} (t_i^{Remote} * P_R)$$

where, P_R is the power consumption during remote execution.

3.1.3 Communication model

When methods are offloaded to the cloud, data must be transmitted between the mobile device and the cloud server. The communication time between methods i and j is determined by:

$$C_{ij} = \frac{Data_{ij}}{B}$$

where, $Data_{ij}$ is the amount of data transmitted between methods i and j, and B is the network bandwidth.

The total communication time is given by:

$$T_{Com} = \sum_{(i,j) \in E} C_{ij}$$

The energy consumption for communication is given by:

$$E_{Com} = \sum_{(i,j) \in E} (C_{i\,j} * P_{Com})$$

where, P_{Com} is the power consumption for data transfer during communication.

3.2 Optimization problem formulation

The CO problem is formulated as a bi-objective optimization problem aiming to minimize both execution time and energy consumption.

3.2.1 Decision variables

Let $x_i \in \{0, 1\}$ be the decision variable for each method $v_i \in V$, where:

$$x_i = \begin{cases} 0, \text{if method } v_i \text{ is executed locally on the mobile device} \\ 1, \text{if method } v_i \text{ is offloaded to the cloud server} \end{cases}$$

The offloading strategy is represented by the binary vector $x = [x_1, x_2, ..., x_n]$.

3.2.2 Objective function

The total execution time is the sum of local execution time, remote execution time, and communication time:

$$T_{Total} = T_{Local} + T_{Remote} + T_{Com}$$

The total energy consumption is the sum of local energy, remote energy, and communication energy:

$$E_{Total} = E_{Local} + E_{Remote} + E_{Com}$$

The objective is to minimize the weighted sum of execution time and energy consumption:

$$Minimize: Fitness = W_t * T_{Total} + W_e * E_{Total}$$

where, W_t and W_e are pre-defined weights such that $W_t + W_e = 1$. These weights allow for expressing preferences between execution time and energy consumption.

3.2.3 Constraints

The optimization problem is subject to the following constraints:

•Binary decision constraint:

$$x_i \in \{0, 1\}, \qquad \forall_i \in V$$

•Task dependency constraint:

If method v_j depends on method v_i (i.e., v_j requires output from v_i), then the execution order must be preserved.

The problem is known to be NP-complete [18], which motivates the use of metaheuristic algorithms such as the proposed ERSO to find near-optimal solutions efficiently.

3.3 Proposed Enhanced Rat Swarm Optimization algorithm

The proposed approach, ERSO for MCC offloading, depicted in Figure 2, utilizes hybrid algorithms that integrate the benefits of both linear programming and graph models to attain an optimal and dynamic solution by factoring in the previously stated costs: local, remote, and communication costs. This is achieved through the use of three modules, including profiler, analyzer, and solver, with the solver tasked with identifying the optimal partitioning outcome: which methods ought to remain on the mobile device and which should be offloaded. The application is partitioned during offloading to determine whether the process should be offloaded or executed locally. In our approach, application partitioning is a fundamental activity conducted before decision-making. It is assumed that the methods entering the decision engine are offloadable methods. In the context analysis and profiling phase, several metrics are gathered through the profiling tool for distinct factors to facilitate an

optimal offloading decision, regarded as the final step of the CO process. A decision engine helps to decide when to offload to the remote or cloud using the enhanced ERSO algorithm, designed explicitly for CO in MCC environments. The suggested approach seeks to improve the allocation of tasks between mobile devices and cloud resources, including variable environmental factors, execution costs, and energy efficiency.

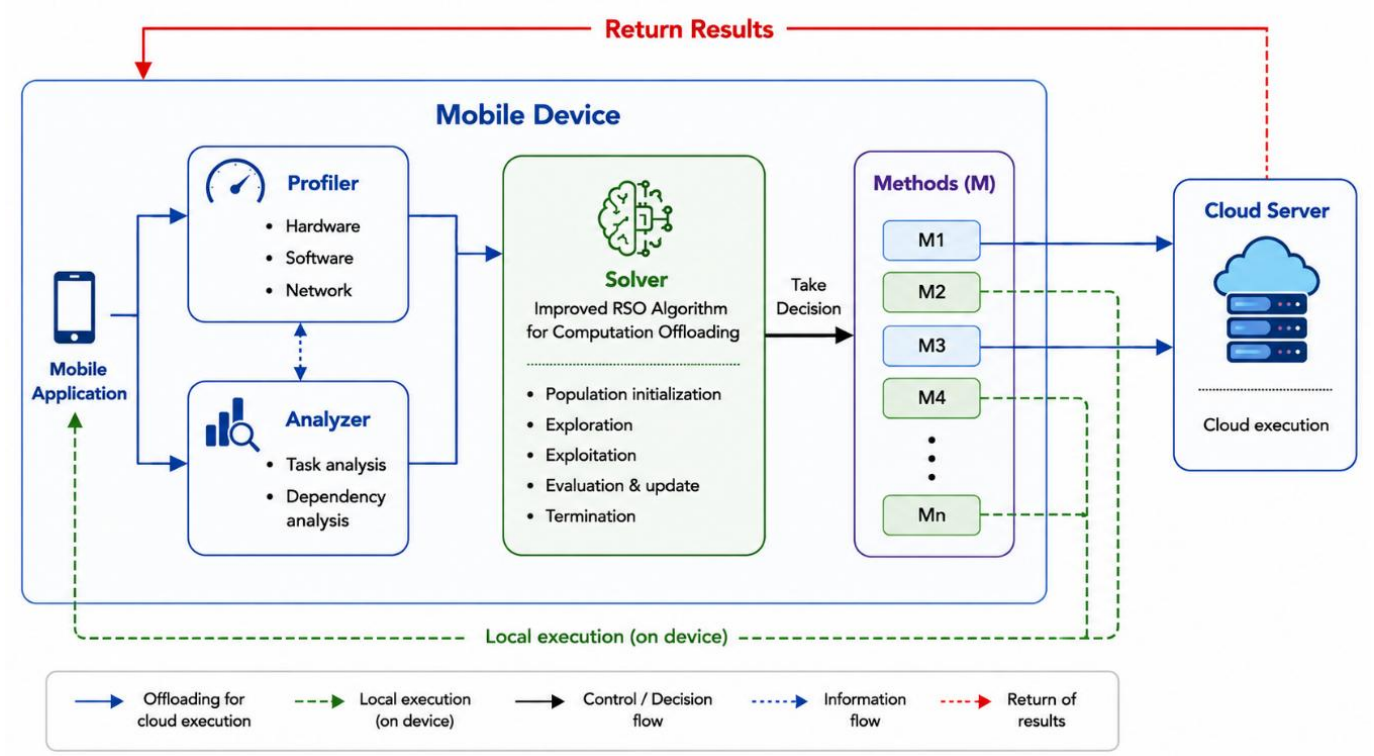


Figure 2. Schematic representation of application partitioning and offloading

Table 4. Characteristics of the proposed approach in mobile cloud computing (MCC) partitioning

Approach	Granularity	Language	Model	Profiler	Allocation Decision	Annotation
Application Partitioning Algorithm (APA) proposed	Method	Single	Hybrid	Software Hardware Network	online	Automatic

Our research contribution can be effectively visualized and compared using the thematic taxonomy of Application Partitioning Algorithm's (APA) classification illustrated in [31]. Table 4 delineates the distinct attributes of our work, situating it clearly within the established taxonomy framework:

(1) Granularity: In our approach, partitioning is method-based, which allows for effective management and offloading of individual tasks. Besides, it provides flexibility and finer control over performance optimization [57]. Table 5 illustrates a comparison between the offloading-based methods, classes, and objects:

(2) Supported language: The approach was developed using a single programming language, Python. Python was chosen due to its widespread adoption and extensive libraries. Its simplicity, versatility, and comprehensive libraries suit complex tasks like application partitioning.

(3) Partitioning model: The hybrid model employed in this approach integrates a method call graph representation with linear programming for problem formulation. This method visually represents interactions among methods and facilitates mathematical optimization via linear programming.

(4) Profiling: Profiling is essential for acquiring data regarding software, hardware, and network characteristics. It aids in making informed decisions regarding offloading mechanisms, contingent upon the capabilities and status of both the cloud server and the mobile device.

(5) Allocation decision: Real-time online decision-making that improves adaptation to changes in the application context. It entails adapting to user behavior, network configurations, or device conditions for enhanced and dynamic offloading techniques.

(6) Techniques for analysis and annotation: Automating analysis and annotation enhances the efficiency and accuracy of identifying offloading strategies. This methodology guarantees the selection of methods with the highest profit potential from offloading, considering method dependencies, conditions, and constraints.

Table 5. Comparison between the offloading of methods, classes, and objects [57]

Strategy	Difficulty of Dynamic Analysis
Method	*
Class	**
Object	***

These components jointly enhance performance and reduce energy consumption, rendering the technique adaptive, efficient, and responsive to variations in application context and system conditions.

In what follows, we give a description of each module illustrated in Figure 2:

(1) Profiling module: The profiling module actively monitors hardware, software, and network components, utilizing specific profiling techniques to gather detailed information. This ensures comprehensive tracking of system performance by concentrating on the underlying hardware infrastructure, software processes, and network activity.

•Hardware profiling: Its objective is to collect detailed information about the physical hardware resources of both mobile and cloud devices, such as the cloud server CPU, mobile device CPU, and mobile device battery capacity.

•Software profiling: It gathers data about the mobile application, including metrics such as code size, CPU usage, and other relevant software characteristics, which will be used later by the solver module to analyze the resource requirements and characteristics of the application, thereby enhancing overall performance and optimizing CPU utilization.

•Network profiling: Collects information on the network environment, including bandwidth and connectivity options (3G, 4G, 5G, Wi-Fi). This module enables informed decision-making regarding method allocation and offloading, ensuring efficient data transfer processes and optimal resource utilization.

(2) Analyzer: The analyzer has two main functionalities. First, it partitions the mobile application into methods, and second, it follows the common practice in programming by representing the mobile application as a graph (Figure 3), where each vertex symbolizes a computational component (method). Each edge denotes the communication between these components. The weight on a vertex signifies the computational load required to execute that component on the mobile and cloud servers. In contrast, the weight on an edge indicates the relative communication demand between components. Communication costs between operations that are executed in the same location (either locally or remotely) are assumed to be negligible. Graph partitioning is a fundamental problem in many computer science domains. It involves dividing a graph into k equal sets while minimizing the edges between them. This problem is particularly relevant in the context of computation offloading to the network, where the goal is to partition the computational methods into groups and dispatch them to a single network node [56, 58]. The communications between methods within the same group essentially have no overhead because the methods will be executed in the same network node. Thus, it is desirable to group methods with high communication requirements; internal communication is maximized while minimizing external communication between different groups, effectively reducing overhead and optimizing performance. When $k = 2$, this problem is known as the min-cut bipartitioning problem. Employing this strategy tackles the NP-hard challenge of finding an optimal solution for graph partitioning [59].

(3) Solver: or decision engine, is a crucial element in the offloading framework, determining whether to offload tasks to a distant server based on the parameters available from the profiling process and the analyzer's output. The profiler and analyzer are two sources of information that the solver module gathers and utilizes to enhance the solution. This module employs the RSO algorithm to partition the application into groups, facilitating the equitable distribution of workloads between local (device) and remote (cloud) execution, hence minimizing overall execution time. The suggested RSO algorithm integrates a weighted fitness function that combines execution time and energy costs. This dual-objective optimization guarantees that the solution minimizes task execution delays while simultaneously decreasing energy consumption, which is essential for resource-constrained mobile devices. The weights for execution and energy costs can be adjusted dynamically according to the application's specifications.

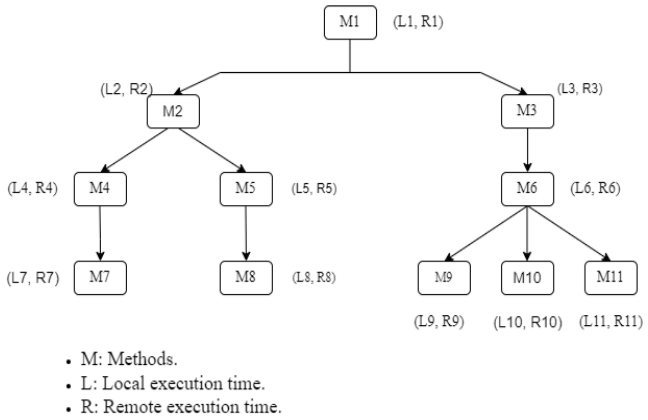


Figure 3. Example of a weighted call graph generated by the analyzer module

Rats, in their natural environments, exhibit social intelligence. The primary inspiration of the RSO algorithm is the aggressive behavior of rats towards other animals, characterized by chasing and fighting [12, 20].

(1) Chasing the prey: The primary premise in this phase is that the most effective search rat is one that is aware of the prey's location. Consequently, the subsequent equations have been provided:

$$\vec{P} = A * \vec{P_i}(x) + C(\vec{P_r}(x) - \vec{P_i}(x))$$

where, $P_i(x)$ is the rats' position in the space, and $P_r(x)$ is the position of the best rat in the population. A and C are two calculated parameters.

$$A = R - x\left(\frac{R}{Max_{iteration}}\right)$$

where, x represents the current iteration from $Max_{iteration}$ and R is a random number in the interval [1, 5].

$$C = 2 \times \text{rand}()$$

where, C is a random number in the interval [0, 2].

(2) Fighting the prey: The following equation models mathematically the process of fighting the prey by rats:

$$\vec{P_i}(x+1) = \left|\vec{P_r}(x) - \vec{P}\right|$$

Algorithm 1 illustrates the pseudo-code of the RSO algorithm [13].

Algorithm 1: Rat Swarm Optimizer (RSO)

Input: The rats population P_i (i = 1, 2, ..., n).
Output: The optimal search agent.
1: **Procedure** RSO
2: Initialize parameters A, C, and R
3: Calculate the fitness value of each search agent
4: $P_r \leftarrow$ the best search agent
5: While ($x < Max_{iteration}$) do
6: For each search agent do
7: Update the position of current search agent by Eq. (4)
8: End for
9: Update parameters A, C, and R
10: Check if there is any search agent which goes beyond the given search space and then adjust it
11: Calculate the fitness of each search agent
12: Update P_r if this is a better solution than the previous optimal solution
13: $x \leftarrow x + 1$
14: End while
15: Return P_r
16: End procedure

Table 6. Limitations of standard Rat Swarm Optimizer (RSO) and corresponding enhancements in Enhanced Rat Swarm Optimization (ERSO)

Limitation	Impact on MCC Offloading	Our Enhancement
Premature convergence	May get stuck in suboptimal partitions	Diverse initial population
No memory mechanism	Cannot learn from past good solutions	Memory component
Fixed mutation	Insufficient diversity in complex search spaces	Guided mutation targeting worst agents
No energy awareness	Cannot balance execution time and energy	Energy-aware fitness function

Note: MCC = mobile cloud computing.

The RSO algorithm promises to solve optimization problems but needs some adjustments to work effectively in MCC environments (Table 6). The suggested enhancements include:

(1). Balancing exploration and exploitation

To enhance exploration, we replace standard random initialization with a diverse random matrix generation. Standard initialization often produces clustered solutions, limiting exploration. Our approach adds small controlled perturbations to the initial population, increasing diversity and reducing the risk of premature convergence to suboptimal solutions:

$$M_{diverse} = clip(M_{random} + U(-0.1, 0.1), 0.1)$$

where, $M_{diverse}$ is a random matrix in [0, 1], $U(-0.1, 0.1)$ is a uniform perturbation in the range [−0.1, 0.1], and clip(0, 1) ensures values stay in [0, 1]. This provides a more robust foundation for optimization, particularly in high-dimensional problems.

(2). Energy-aware fitness function

A new fitness function will be introduced to evaluate solutions depending on:

• Energy consumption: This includes the energy required for the device's computational operations and communication with the cloud.

• Execution duration: This considers the time required for both local and remote execution.

The fitness function will incorporate these two components into a singular score, modified according to the user's preferences.

(3). Real-time method partitioning

The mobile application will be conceptualized as a graph, with tasks designated as nodes and dependencies represented as edges. This enables dynamic partitioning, allowing real-time decisions regarding which tasks should be executed locally and which should be delegated to the cloud.

(4). Memory component

To accelerate convergence and prevent the algorithm from revisiting suboptimal solutions, we introduce a memory mechanism that stores promising offloading strategies. The memory is defined as:

$$Mm = \{x_1, x_2, \ldots, x_k\}$$

where, k is the memory size (set to 5). When a new solution x_{new} improves the global best, it is added to memory. If memory is full, the worst solution is replaced:

$$Mm = Mm \backslash \{x_{worst}\} \cup \{x_{new}\}$$

This mechanism has two key benefits: (1) it preserves high-quality solutions for future reference, and (2) it accelerates the search by guiding the algorithm toward promising regions of the search space

(5). Mutation mechanism

To further reduce the risk of entrapment in local optima, a guided mutation mechanism is incorporated. Instead of applying mutations randomly across all agents, the mechanism targets agents with the worst fitness values. The set of worst agents is identified as:

$$W = arg \min_i Fitness\ (x_i)$$

where, W represents the set of agents with the worst fitness values, and x_i is the position (offloading strategy) of agent i. For each selected agent $i \in W$, a mutation is applied to a randomly selected bit:

$$x_i(j) = \begin{cases} 1 - x_i(j), & if\ j \in selected\ bits\ for\ mutation \\ x_i(j), & otherwise \end{cases}$$

where, $x_i(j)$ represents the j-th bit of agent i's configuration. This operation flips a random bit of the worst agents, introducing meaningful diversity where it is most needed. The mutation probability is fixed:

$$p_{mut} = 0.2$$

where, p_{mut} is the probability that an agent undergoes mutation. Only the worst agents (typically the bottom 5) are selected for mutation, ensuring that the population retains stability while still encouraging exploration where it is most needed.

This guided mutation strategy has three key advantages:

• Targeted exploration: Mutations are applied only to the worst solutions, improving them without disrupting the better

solutions.

• Population stability: The best solutions remain unchanged, preserving the algorithm's convergence.

• Avoidance of local optima: By introducing diversity in the worst-performing agents, the algorithm can escape local optima.

The ERSO algorithm functions as outlined below:

1. Initialization: Construct an initial array of potential solutions (agents) and set parameters for exploration, exploitation, and mutation.

2. Evaluation of solutions: Employ the energy-aware fitness function to assess the quality of each solution.

3. Pursue optimal solutions: Transition between exploration and exploitation according to varying parameters. Subsequently, mutation will be implemented to preserve variation among solutions. Afterward, update the memory with the optimal solutions identified thus far.

4. Dynamic method partitioning: Utilize a graph-based approach to allocate tasks to the mobile device or the cloud based on prevailing conditions.

5. Repeat and converge: Persist in refining solutions until the algorithm satisfies a termination criterion, such as a predetermined number of iterations or negligible improvement in fitness.

6. Output results: Present the optimal work allocation strategy, along with data on energy consumption and execution time.

Algorithm 2 shows the detailed steps:

Algorithm 2. Enhanced Rat Swarm Optimizer

Input:

•nb_{agents}: Number of agents (particles or solutions)

•$Max_{iterations}$: Maximum number of iterations

•MCC parameters: method graph, execution times (local and cloud), communication times

Output:

•G_{best}: Optimal offloading strategy (method allocation)

Step 1: Initialization

During the initialization phase, a diverse random matrix is used to generate the initial population. By introducing controlled perturbations, this approach ensures that the population has sufficient diversity to explore a broader range of potential solutions. Each agent represents a potential offloading strategy, modeled as a binary vector. The fitness of each solution is calculated using an energy-aware fitness function that incorporates execution time, local computation energy, and communication energy:

1.1 An initial random population of agents is generated, *P*, where each agent represents a potential offloading strategy:

$$P = \{P_1, P_2, ..., P_{nb\_agents}\}$$

Each P_i is a binary vector representing task assignments (*0* for *local execution*, *1* for *cloud execution*).

1.2 Calculates, afterwards, the fitness using the energy-aware fitness function for each agent:

a. Determine the global best solution (G_{best}), which represents the best solution among all agents in the population, and the personal best solutions (P_{best}), denoting the best solution an individual agent has attained to date:

$$G_{best} = argmin\ Fitness(P_i)$$

b. Initialize memory to store promising solutions with a maximum size of $memory_{size}$.

Step 2: Adaptive parameter update

2.1 For the current iteration *x*, compute the adaptive parameter *A*:

$$A = R * (1 - \frac{x}{Max_{iteration}})^2$$

where, *R* is a random value within the interval [1,5]; this guarantees that exploration dominates in early iterations and exploitation increases as $x \rightarrow Max_{iterations}$.

2.2 Set the random coefficient *C*. *C* is a random coefficient that adds stochastic behavior, allowing agents to explore diverse solutions:

$$C = 2 * \text{rand}()$$

where, rand() is a random value in [0,1].

Step 3: Position update

3.1 For each agent *i*, update its position based on A, C, and the best solutions. Select a guiding solution from memory if available; otherwise, use G_{best}.

$$P_i^{new} = A * P_i + C * (Guide - P_i)$$

where, P_i represents the agent's current position, whereas the guide is defined by either the global best solution or a potentially effective solution stored in memory; this update enables agents to approach high-quality solutions while exploring new regions of the solution space.

3.2 A sigmoid function is utilized to represent method partitioning decisions as binary values based on the updated positions.

$$T(v) = \frac{1}{1 + e^{-v}}$$

Values approaching 1 signify *remote offloading*, whereas values approaching 0 signify *local execution*.

Step 4: Mutation

Mutation generates variation to prevent stagnation in local optima. The mutation system dynamically finds agents with the worst (lowest) fitness values and applies targeted mutations to them. This modification improves the weakest solutions by altering bit configurations, thereby promoting exploration while maintaining population stability.

The mutation operation is defined as follows:

$$P_i^{mutated} = \begin{cases} 1 - P_{ij,} \text{ if } j \in \text{selected bits for mutation} \\ P_{ij,} \qquad \text{otherwise} \end{cases}$$

where, P_{ij} represents the *j*-th bit of the configuration of agent *i*. The selected bits for mutation are chosen based on the agent's fitness ranking, ensuring that the weakest solutions are prioritized for improvement.

This approach ensures that the algorithm explores new areas of the solution space effectively, mitigating the risk of getting trapped in local optima while maintaining meaningful diversity in the population.

Step 5: Fitness evaluation

5.1 Compute the fitness of the updated positions (mentioned in the previous section)

5.2 Compare the updated fitness with the previous G_{best} and P_{best}:

•Update G_{best} if a better solution is found.

•Update P_{best} for each agent if the new position improves the fitness.

Step 6: Memory update

Store the best solutions in memory to guide future iterations. The algorithm maintains a memory component to store the best solutions identified during the search process. If a newly discovered solution improves upon the global best, it is stored in memory. This mechanism guarantees the preservation of high-quality solutions while the algorithm explores other regions of the solution space.

Step 7: Convergence check

The algorithm terminates when the maximum number of iterations is reached or if the improvement in G_{best} stagnates over a predefined threshold. The final output includes the global best solution and its corresponding task allocation strategy.

Repeat Steps 2–6 until one of the following criteria is met:

• Maximum number of iterations is reached ($x = max_{iteration}$).

• Improvement in G_{best} stagnates for a predefined number of iterations.

Output:

The algorithm returns G_{best}, the optimal method allocation strategy minimizing energy consumption and execution time.

The pseudo-code of the enhanced algorithm is illustrated in Algorithm 3.

Algorithm 3. Enhanced Rat Swarm Optimization (ERSO)

Input: Method graph ($G = (V, E)$), execution times, communication costs.
Output: Optimal partitioning (G_{best}).
1: Initialize population ($P_1, P_2, ..., P_n$) using diverse random initialization
2: Compute the initial fitness of each agent
3: Set G_{best} = minimum fitness function and store initial memory
4: Repeat until convergence:
5: Compute adaptive parameter A and random coefficient
6: Update agent positions
7: Transform positions into binary form using a sigmoid function
8: Apply enhanced mutation to diversify solutions, targeting worst agents
9: Recalculate fitness for updated positions
10: Update G_{best} and memory if a better solution is found
11: Return G_{best} and corresponding task allocation

3.4 Complexity analysis

The computational complexity of the proposed ERSO algorithm is analyzed as follows:

Let n be the number of methods/tasks in the application (size of the graph), and N be the number of agents (population size). The complexity of each step is:

• Initialization: $O(N \cdot n)$ to generate and evaluate the initial population.

• Position update: $O(N \cdot n)$ per iteration to update each agent's position.

• Fitness Evaluation: $O(N \cdot n)$ per iteration to evaluate each agent.

• Mutation: $O(N \cdot n)$ per iteration to apply mutations.

• Memory Update: $O(N \cdot n)$ per iteration.

Overall, the time complexity of the ERSO algorithm is:

$$O(T\ N\ n)$$

where, T is the number of iterations. This complexity is linear with respect to the number of agents and tasks, making the algorithm suitable for resource-constrained mobile devices. The space complexity is $O(N \cdot n)$ for storing the population and memory. Compared to exact algorithms that have exponential complexity $O(2n)$ for the graph partitioning problem, the ERSO algorithm provides a polynomial-time heuristic solution that is practical for real-time MCC offloading decisions.

4. RESULTS AND DISCUSSION

4.1 Enhanced Rat Swarm Optimization vs. full-offloading and no-offloading strategies

Static analysis and dynamic profiling can be integrated to create the weighted communication graph (WCG) of an application. We utilize a face recognition application used in studies [11, 14] represented as a call graph (Figure 4). By dividing the local estimated execution time by the speedup factor (F), we may derive the remote estimated execution time. The communication cost between the mobile device and the server is calculated by dividing the data transmission by the bandwidth (B) when a task is offloaded to the server. This provides all the information required to construct the WCG, including transmission and remote execution costs [11, 18]. We evaluated the ERSO algorithm's performance through experiments conducted under seven different configurations of the speedup factor (F) and bandwidth (B). The values of P_L, P_R, and P_{Com} are fixed to 0.9, 0.3, and 13 watts, respectively [11, 18].

The following configurations were considered (Table 7). These configurations make it possible to examine the ERSO algorithm's flexibility in different computing and network environments.

The experimental results were evaluated according to the fitness values, which include the execution time and energy consumption, with various weights (0.6 and 0.4, respectively) for the three offloading strategies: non-offloading, full offloading, and partial offloading using ERSO. These strategies serve as benchmarks to evaluate the algorithm's ability to achieve an optimal equilibrium between computational performance and resource utilization. A simulation was performed using the WCG (Figure 5) where twenty search rats (agents) were employed across 10000 search iterations. We have obtained different results under the different configurations (Table 8). The results of partitioning and fitness will change as B or F changes. The executions were repeated ten times (Figure 6) for each couple of parameters' values (B and F) using the ERSO.

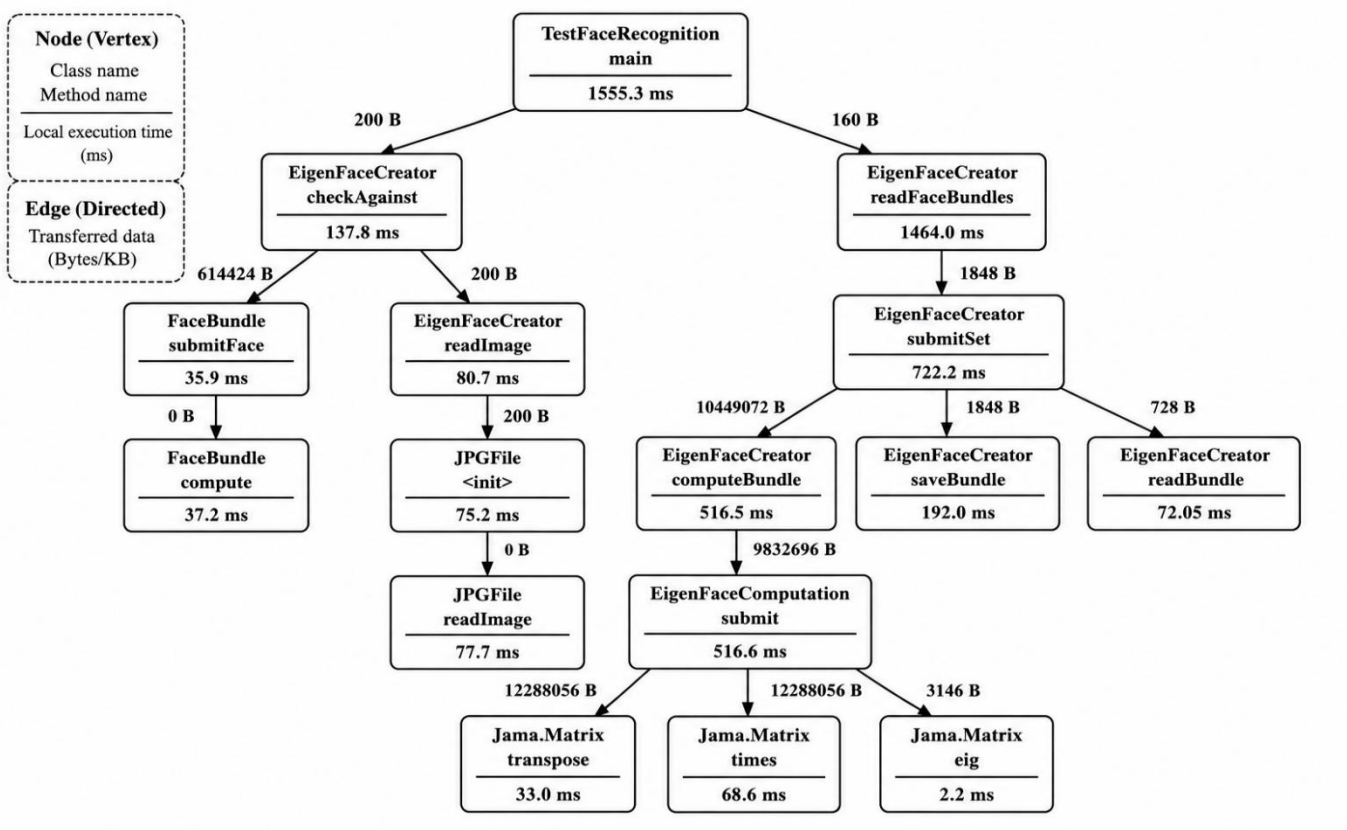


Figure 4. A call graph for the face recognition application constructed [14]

Table 7. Configurations used for experimental analysis

S	Parameters		Description
	F	B	
1	20	10	Simulates a mid-tier cloud server accessed over a moderate network connection
2	30	3	Integrates high computational power accessed over a poor network connection
3	30	10	Simulates a high-performance cloud server accessed over a moderate network connection
4	30	100	Simulates a high-performance cloud server accessed over a fast network connection
5	5	3	Low-performance cloud server with a poor network connection
6	5	20	Low-performance cloud server with a decent network connection
7	5	100	Low-performance cloud server with a high-speed network connection

Table 8. Fitness results under different configurations

S	Comparison	Fitness Value	Gain (%)	Discussion
1	ERSO vs. NO	3355.288 vs. 5363.472	37.46	In this balanced scenario, ERSO outperforms both NO and FO. The moderate speedup and bandwidth allow the ERSO algorithm to effectively balance local and remote execution, minimizing energy consumption and execution time. FO struggles due to communication overhead, while NO suffers from the mobile device's limited resources.
	ERSO vs. FO	3355.288 vs. 5727.783	41.42	
2	ERSO vs. NO	3332.188 vs. 5363.472	37.86	Despite the high computational speedup, FO performs poorly due to the very low bandwidth, which significantly increases communication overhead. Partial Offloading excels by minimizing data transfer and leveraging local computation, demonstrating its adaptability to challenging network conditions.
	ERSO vs. FO	3332.188 vs. 14114.110	76.39	
3	ERSO vs. NO	3329.191 vs. 5363.472	37.91	In this scenario, ERSO again outperforms both NO and FO. The high speed-up and moderate bandwidth allow FO to perform better than in Scenario 2, but it still underperforms compared to ERSO due to communication costs.
	ERSO vs. FO	3329.191vs. 5685.603	41.43	
4	ERSO vs. NO	2378.567 vs. 5363.472	55.65	With high computational speedup and high bandwidth, FO performs nearly as well as ERSO. However, ERSO still achieves a slight improvement by optimizing task allocation and minimizing energy consumption. This scenario highlights the algorithm's ability to leverage abundant resources effectively.
	ERSO vs. FO	2378.567 vs. 2434.607	2.30	
5	ERSO vs. NO	3593.158 vs. 5363.472	33.00	In this challenging scenario, FO performs poorly due to both low speed-up and very low bandwidth. ERSO significantly outperforms both NO and FO by dynamically balancing local and remote execution, demonstrating its robustness in resource-constrained environments.
	ERSO vs. FO	3593.158 vs. 14535.916	75.28	
6	ERSO vs.	3558.371 vs.	33.66	With low computational speedup, FO struggles to achieve significant performance gains,

	NO	5363.472		even with moderate bandwidth. ERSO outperforms both strategies by optimizing task allocation and minimizing energy consumption.
	ERSO vs. FO	3558.371 vs. 4301.300	17.27	
7	ERSO vs. NO	2788.181 vs. 5363.472	48.01	In this scenario, FO benefits from high bandwidth but is still limited by low computational speedup. ERSO achieves a slight improvement over FO and a significant improvement over NO, demonstrating its ability to adapt to varying resource conditions.
	ERSO vs. FO	2788.181 vs. 2856.413	2.39	

Note: ERSO: Enhanced Rat Swarm Optimization; NO: No offloading; FO: Full offloading.

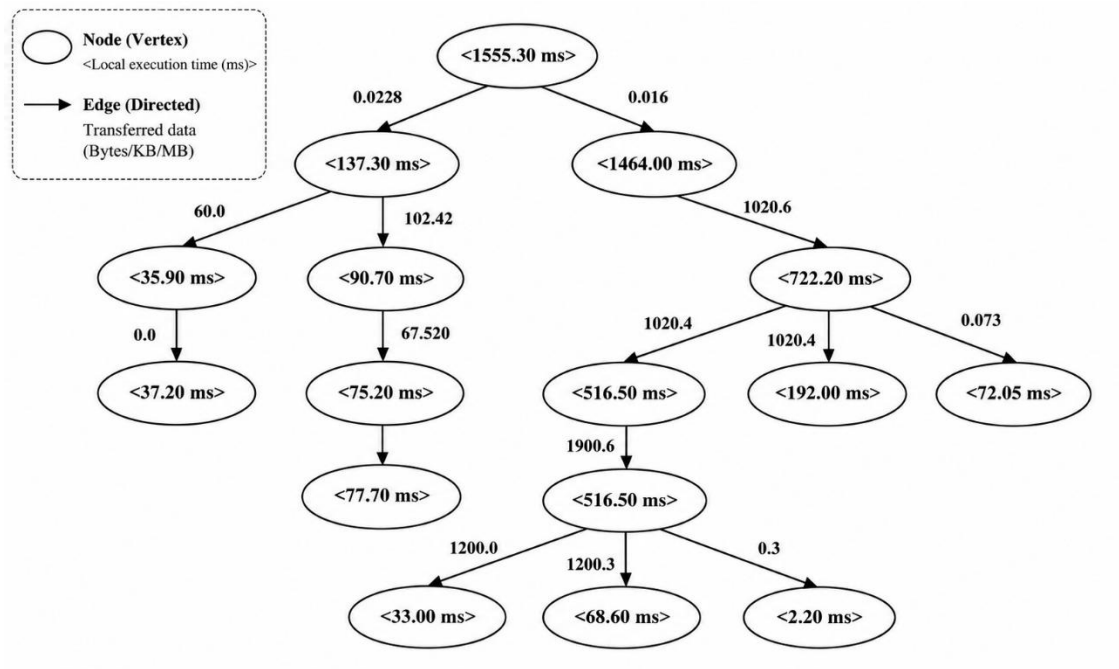
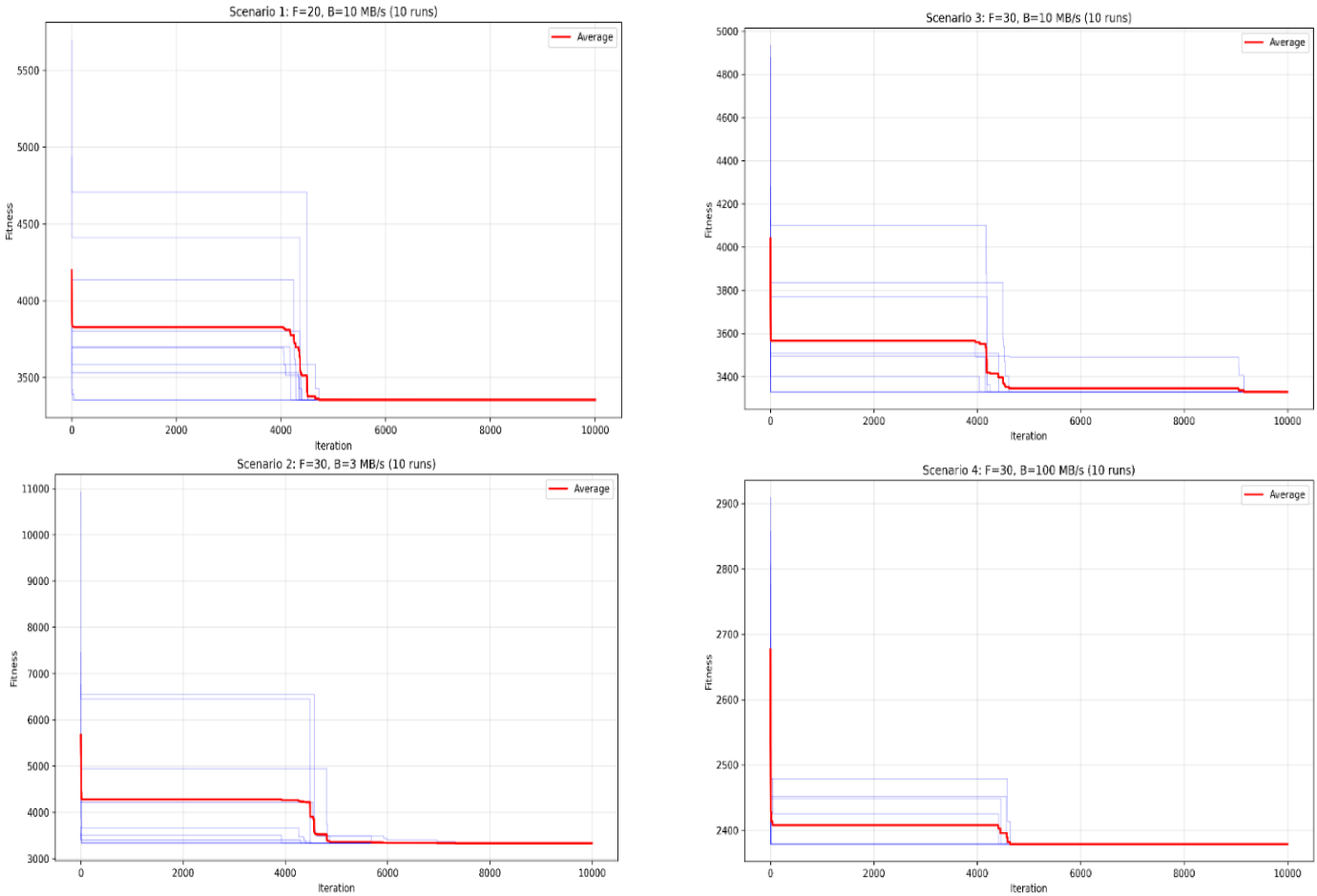


Figure 5. Weighted communication graph (WCG) when F = 20 and B = 10 MB/s



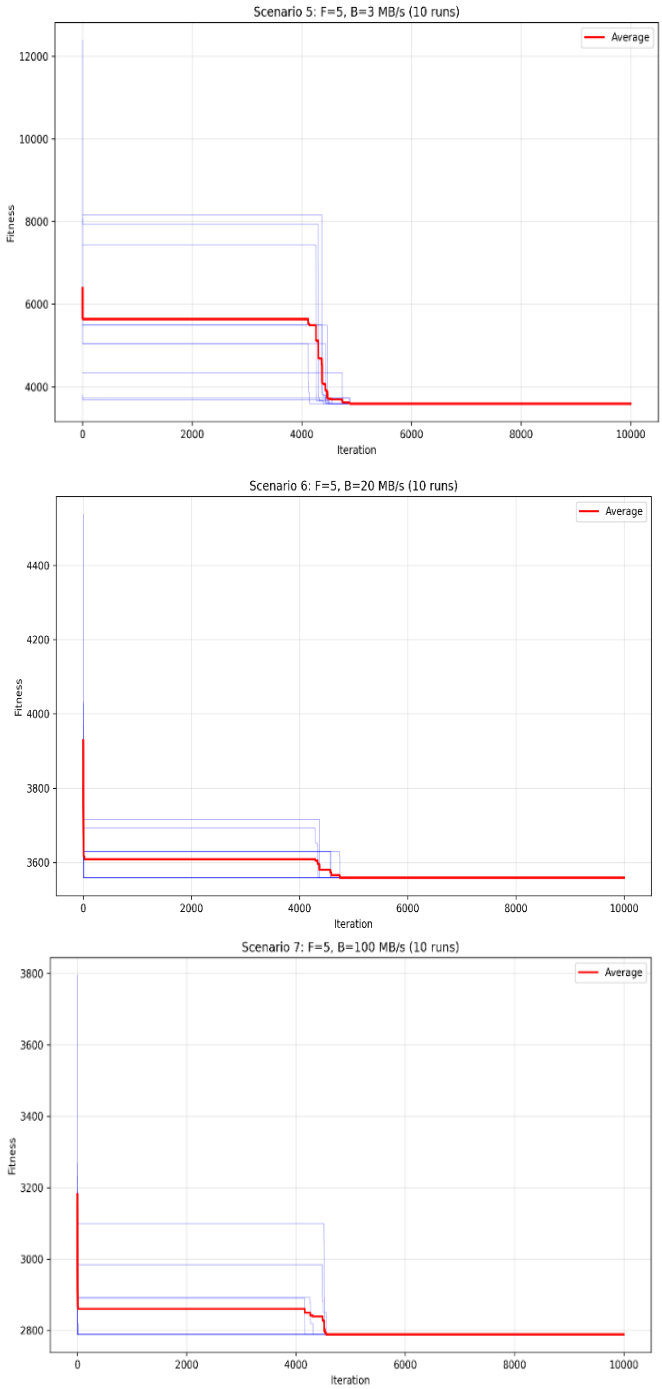


Figure 6. Convergence curve of Enhanced Rat Swarm Optimization (ERSO) over 10 executions

As mentioned above, the partitioning fitness was tested with three strategies:

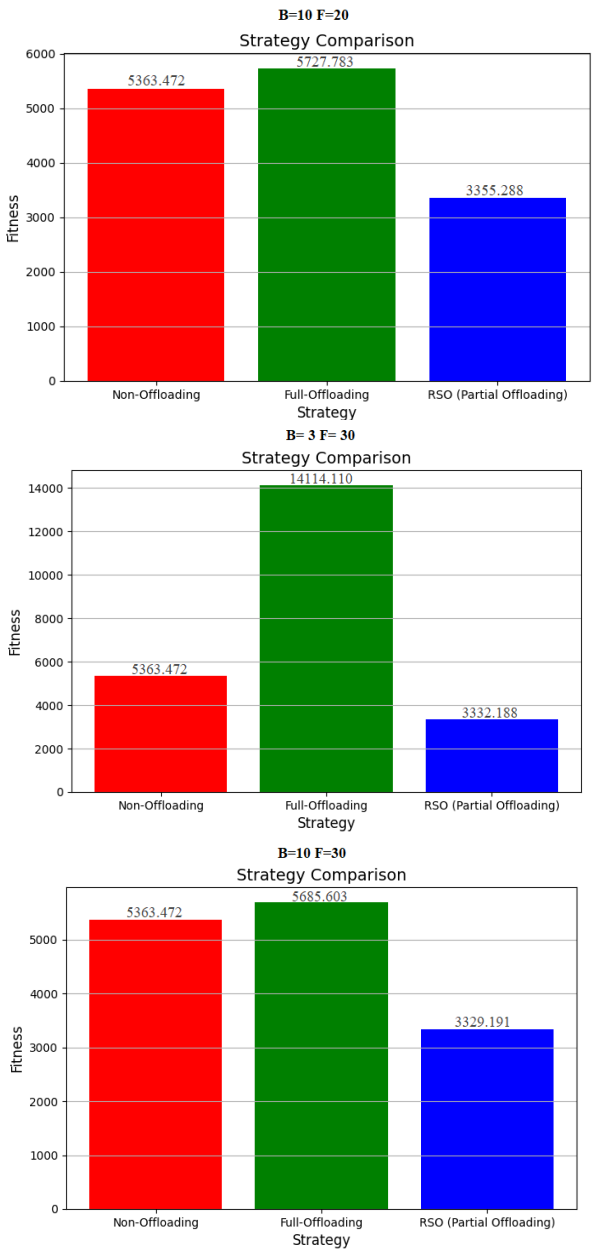
(1) Non-offloading: In this strategy, all methods are executed locally on the mobile device. While this approach reduces the communication overhead associated with data transfer to remote servers, it frequently results in longer execution times and higher energy consumption due to mobile devices' limited processing power and battery capacity. This strategy provides a baseline for determining the impact of no-task offloading on system performance.

(2) Full offloading: Here, all offloadable methods are executed on a remote server, utilizing the server's enhanced computational capabilities. This strategy substantially decreases local execution time and energy consumption by alleviating the computing load on the mobile device. The advantages of complete offloading depend on network conditions, as restricted bandwidth may negate the improvements in computational speed and efficiency

(3) Partial offloading: Partial offloading allocates methods across local and remote execution nodes according to the optimization results derived from the ERSO algorithm. This strategy dynamically identifies the appropriate partitioning of methods to reduce total execution time and energy costs, in contrast to the rigid strategies of non-offloading and full offloading. Partial offloading using the ERSO takes into account several considerations, such as the processing requirements of each method, communication overheads, and the unoffloadable nature of certain methods.

The effectiveness of the ERSO algorithm in achieving an optimal equilibrium between execution time and energy consumption is assessed through a comparison of different strategies. The gain for partial offloading (ERSO) is calculated compared to the non-offloading approach using the following formula:

$$Gain(\%) = \frac{Fitness(Non\ Offloading) - Fitness(Partial\ Offloading)}{Fitness(Non\ Offloading)} \times 100$$



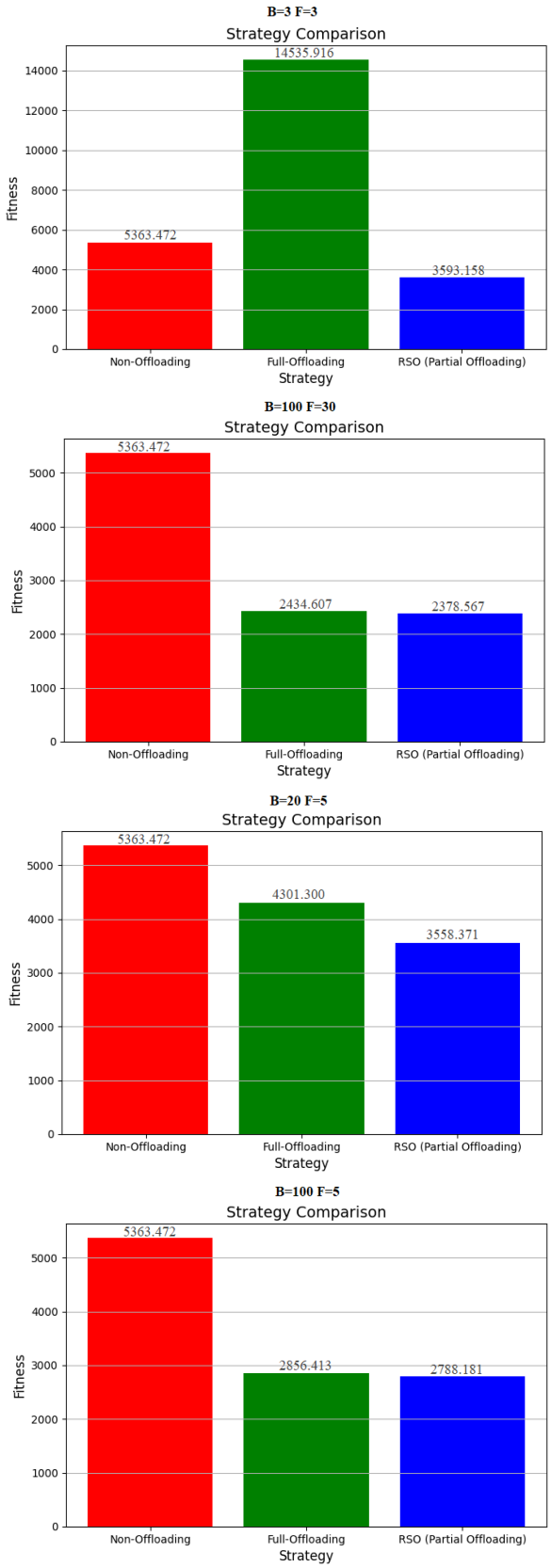


Figure 7. Strategies comparison with different values of factor (F) and bandwidth (B)

The analysis for each configuration is summarized in Table 8. This gain underscores the effectiveness of the ERSO algorithm in balancing computational and communication costs, particularly in scenarios with moderate computational and network resources (Figure 7).

The comparison of all scenarios gives several important insights regarding the performance of the ERSO algorithm under a variety of different computational and network conditions. The insights presented here highlight the algorithm's strengths and adaptability, as well as its ability to outperform the non-offloading and full offloading. The experimental results lead to several important observations regarding the performance of ERSO under different computational and network conditions:

(1) Partial offloading using ERSO outperforms non-offloading: The ERSO achieves considerable increases over non-offloading across all scenarios, with gains ranging from 33.00% to 55.65%. This illustrates the algorithm's ability to effectively use the cloud resources while simultaneously decreasing the energy consumption and execution time.

(2) Partial offloading using ERSO outperforms full offloading in most scenarios: In scenarios with low to moderate bandwidth (B = 3, 10, 20 Mbps), the ERSO notably outperforms full offloading, with gains ranging from 17.27% to 76.39%. This highlights the algorithm's ability to reduce communication costs and adjust to fluctuating network conditions.

(3) High bandwidth reduces the advantage of partial offloading: In high bandwidth scenarios (B = 100 Mbps), full offloading exhibits performance comparable to partial offloading, with gains of only 2.30% to 2.39%. Nonetheless, the ERSO still achieves slight enhancements through the optimization of method allocation.

(4) Low computational speedup limits full offloading: In scenarios with low computational speedup (F = 5), full offloading struggles to achieve significant performance gains, even with high bandwidth. The ERSO consistently outperforms full offloading in these scenarios.

(5) Robustness across diverse conditions: The ERSO algorithm demonstrates robustness and adaptability across a wide range of scenarios, from low-speedup, low-bandwidth conditions to high-speedup, high-bandwidth environments. This makes it a versatile solution for real-world MCC applications.

(6) Stability: Across all 10 executions, the algorithm consistently converges to similar fitness values, indicating its robustness and reliability.

The results across all scenarios demonstrate the ability of the ERSO algorithm in optimizing CO in MCC. By dynamically balancing local and remote execution, the algorithm achieves significant improvements in energy efficiency and execution time, even under challenging conditions. These findings validate the ERSO algorithm as a powerful tool for energy-efficient CO in MCC.

4.2 Enhanced Rat Swarm Optimization algorithm vs. min-cut algorithm

When the ERSO algorithm combines principles of swarm intelligence with specific enhancements such as diverse initial population generation, enhanced mutation targeting weaker solutions, and dynamic parameter tuning, the min-cut algorithm [14] is a graph-based optimization approach that partitions tasks based on WCG. Tasks are assigned to local or remote execution nodes by minimizing the "cut" cost, which considers execution and communication costs. Both approaches balance execution time and energy consumption to determine the optimal partitioning of tasks between local and remote execution. Table 9 demonstrates a simple comparison between the two approaches when B = 10, F = 20.

Table 9. Enhanced Rat Swarm Optimization (ERSO) approach vs. min-cut approach

Approach	Fitness Value	Gain over NO	Gain over FO
ERSO	3355.288	37.46	41.42
Min-cut [14]	3804.3102	29.07	33.59

Note: NO: No offloading; FO: Full offloading.

As shown in Table 9, ERSO achieves superior performance with the lowest fitness value (3355.288), indicating superior performance in optimizing the trade-off between execution time and energy consumption. It achieves a 37.46% improvement over non-offloading and a 41.42% improvement over full offloading, significantly outperforming the min-cut algorithm. min-cut performs well but lags behind ERSO.

The min-cut algorithm achieves a fitness value of 3804.310, with gains of 29.07% over non-offloading and 33.59% over full offloading. However, it is less effective than ERSO due to its reliance on static partitioning decisions and lack of adaptive mechanisms. ERSO's dynamic method partitioning and adaptive mechanisms allow it to outperform both non-offloading and full offloading by a significant margin. Its ability to explore diverse solutions and avoid local optima contributes to its superior performance. The diverse initial population and targeted mutation ensure a comprehensive exploration of the solution space, reducing the likelihood of suboptimal convergence. The fitness of 3355.288 showcases its balanced approach to minimizing execution time and energy consumption. While the min-cut algorithm lacks the adaptive mechanisms of ERSO, it still performs well due to its effective task partitioning based on WCGs. The gains of 29.07% over non-offloading and 33.59% over full offloading indicate that min-cut can serve as a viable alternative when computational simplicity and speed are prioritized. However, it is less effective than ERSO in a complex and dynamic MCC environment.

4.3 Enhanced Rat Swarm Optimization algorithm vs. standard Rat Swarm Optimization, Particle Swarm Optimization and Genetic Algorithms

The min-cut approach offers a simpler and faster heuristic method for task offloading. To further validate the effectiveness of the proposed ERSO algorithm, its performance was compared with three widely used binary metaheuristic algorithms: Standard RSO, PSO, and GA.

All experiments were conducted 10 times for each scenario (F, B) to ensure statistical significance. For each algorithm and scenario, we report:

•Mean fitness: The average fitness value over 10 runs.

•Standard deviation: A measure of the algorithm's stability. A low standard deviation indicates that the algorithm consistently finds similar solutions, while a high standard deviation indicates instability. It is calculated using the sample standard deviation formula:

$$\sigma = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \bar{x})^2}{n-1}}$$

where, x_i is the result of the i-th run (e.g., fitness value), $\bar{x}$ is the mean of all results, n = 10 is the number of runs.

Best Iteration (mean ± standard deviation): The average iteration at which the algorithm first found its best fitness, measuring convergence speed.

Table 10. Comparison of Enhanced Rat Swarm Optimization (ERSO), Rat Swarm Optimization (RSO), Particle Swarm Optimization (PSO), and Genetic Algorithms (GA) on the 16-node weighted communication graph (WCG)

S	Algorithms	Fitness Value	Remote (Mean ± Standard Deviation)	Best Iteration (Mean ± Standard Deviation)
S1	GA	3355.29 ± 0.00	10.00 ± 0.00	9 ± 3
	PSO	3355.29 ± 0.00	10.00 ± 0.00	10 ± 6
	RSO	3479.88 ± 143.58	8.30 ± 1.77	9 ± 4
	ERSO	3371.34 ± 50.75	9.80 ± 0.63	493 ± 339
S2	GA	3332.19 ± 0.00	10.00 ± 0.00	17 ± 15
	PSO	3332.19 ± 0.00	10.00 ± 0.00	11 ± 3
	RSO	3765.32 ± 494.01	5.50 ± 1.43	7 ± 6
	ERSO	3403.94 ± 80.80	9.10 ± 0.99	339 ± 236
S3	GA	3329.19 ± 0.00	10.00 ± 0.00	13 ± 12
	PSO	3329.19 ± 0.00	10.00 ± 0.00	11 ± 4
	RSO	3458.96 ± 169.21	8.30 ± 2.06	10 ± 6
	ERSO	3361.71 ± 68.55	9.60 ± 0.84	436 ± 413
S4	GA	2378.57 ± 0.00	12.00 ± 0.00	8 ± 3
	PSO	2378.57 ± 0.00	12.00 ± 0.00	11 ± 4
	RSO	2419.97 ± 58.73	10.90 ± 1.10	7 ± 4
	ERSO	2378.97 ± 0.85	11.80 ± 0.42	304 ± 259
S5	GA	3593.16 ± 0.00	10.00 ± 0.00	12 ± 5
	PSO	3593.16 ± 0.00	10.00 ± 0.00	12 ± 2
	RSO	3929.79 ± 449.36	6.50 ± 2.68	12 ± 9
	ERSO	3649.85 ± 73.19	9.20 ± 1.03	410 ± 238
S6	GA	3558.37 ± 0.00	11.00 ± 0.00	7 ± 3
	PSO	3558.37 ± 0.00	11.00 ± 0.00	10 ± 3
	RSO	3682.40 ± 145.22	9.20 ± 2.10	7 ± 6
	ERSO	3558.37 ± 0.00	11.00 ± 0.00	199 ± 334
S7	GA	2788.18 ± 0.00	12.00 ± 0.00	8 ± 4
	PSO	2788.18 ± 0.00	12.00 ± 0.00	9 ± 4
	RSO	2861.84 ± 78.76	10.70 ± 1.16	6 ± 3
	ERSO	2788.18 ± 0.00	12.00 ± 0.00	284 ± 240

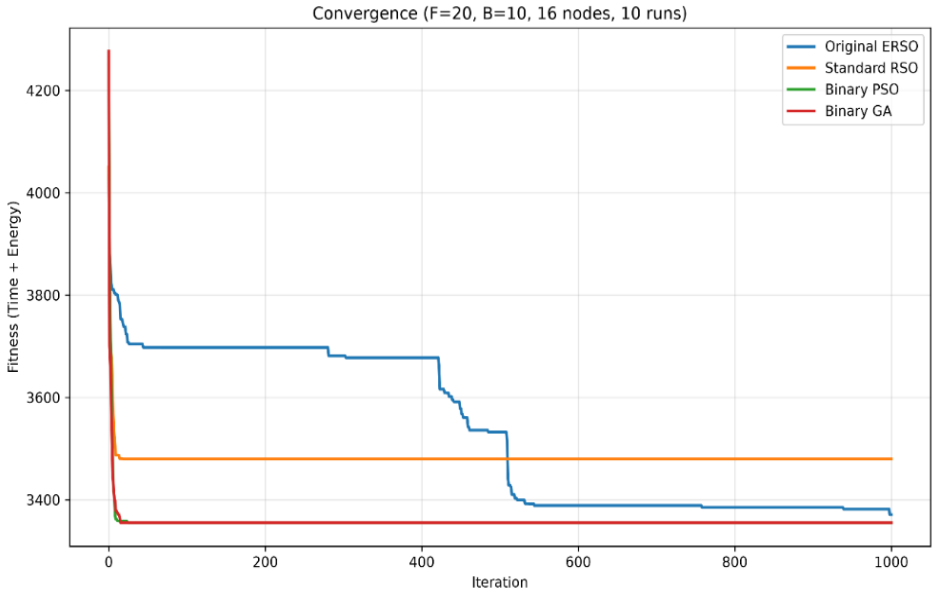


Figure 8. Convergence curves of Enhanced Rat Swarm Optimization (ERSO), Rat Swarm Optimization (RSO), Particle Swarm Optimization (PSO), and Genetic Algorithms (GA) on the 16-node weighted communication graph (WCG) under balanced conditions (F = 20, B = 10)

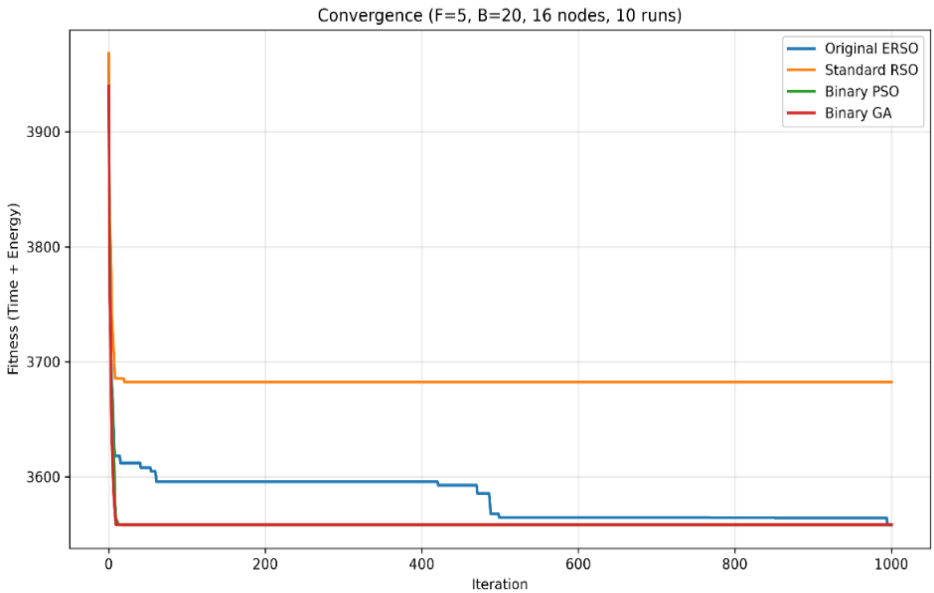


Figure 9. Convergence curves of Enhanced Rat Swarm Optimization (ERSO), Rat Swarm Optimization (RSO), Particle Swarm Optimization (PSO), and Genetic Algorithms (GA) on the 16-node weighted communication graph (WCG) under balanced conditions (F = 5, B = 20)

Table 10 presents the results for all seven scenarios. From the table, we can observe that:

• GA and PSO consistently achieve the optimal fitness with zero standard deviation across all scenarios. This confirms that the 16-node problem is simple enough for these algorithms to reliably find the global optimum (Figure 8). On the other hand, ERSO achieves the same optimal fitness as GA and PSO in Scenarios 1, 3, 4, 6, and 7 (Figure 9). In Scenarios 2 and 5, ERSO achieves fitness values close to the optimal values. This demonstrates that ERSO is a reliable algorithm that consistently finds near-optimal solutions.

• Standard RSO shows high variability, with fitness standard deviations ranging from 58.73 (Scenario 4) to 494.01 (Scenario 2). In many runs, RSO gets stuck in suboptimal solutions, confirming that the standard algorithm lacks the robustness of ERSO. GA and PSO converge extremely fast (8–17 iterations on average), which is expected for simple problems. ERSO takes longer to converge (100–500 iterations) because it explores the search space more thoroughly. This is a trade-off between convergence speed and robustness.

• ERSO consistently offloads a similar number of methods as GA and PSO (9–12 remote methods), while Standard RSO often offloads fewer methods (5–8 remote methods), especially in challenging scenarios. This indicates that ERSO makes more informed offloading decisions.

• From the previous experiments, we can conclude that the 16-node WCG is a relatively simple problem where all algorithms can find the global optimum. To demonstrate the scalability and superiority of ERSO, we need a more complex

problem. The 50-node problem has a search space of $2^{50} \approx 1.12 \times 10^{15}$ possible partitions, making it impossible for simple algorithms to explore exhaustively. In our work, a 50-node WCG was generated synthetically using a realistic model:

• Execution times: Heavy-tailed distribution (scale parameter = 150 ms) with 15% of nodes designated as "heavy" (800–3500 ms) and 10% as "light" (2–15 ms), mimicking real mobile applications.

• Communication costs: A sparse directed graph with 6% edge density, where each edge has a data transfer size between 100 bytes and 50 MB.

• Structure: The graph follows a pipeline structure with additional branches, similar to real mobile application call graphs.

Table 11 presents the results for all the scenarios, where each scenario was run 10 times.

Table 11. Comparison of Enhanced Rat Swarm Optimization (ERSO), Rat Swarm Optimization (RSO), Particle Swarm Optimization (PSO), and Genetic Algorithms (GA) on the 50-node weighted communication graph (WCG)

S	Algorithms	Fitness Value	Remote (Mean ± Standard Deviation)	Best Iteration (Mean ± Standard Deviation)
S1	GA	17930.55 ± 4717.82	20.00 ± 16.17	451 ± 255
	PSO	28448.41 ± 4031.18	22.50 ± 10.61	211 ± 104
	RSO	11928.53 ± 1790.70	38.20 ± 19.61	278 ± 237
	ERSO	11928.53 ± 1790.70	38.20 ± 19.61	202 ± 227
S2	GA	53202.43 ± 16568.71	22.40 ± 12.13	543 ± 302
	PSO	79257.70 ± 14595.21	22.30 ± 9.04	316 ± 180
	RSO	15029.95 ± 0.00	0.00 ± 0.00	271 ± 232
	ERSO	15029.95 ± 0.00	0.00 ± 0.00	280 ± 237
S3	GA	19579.36 ± 5445.93	26.40 ± 13.87	614 ± 256
	PSO	25386.63 ± 6526.69	23.60 ± 13.25	178 ± 114
	RSO	11995.83 ± 2502.15	28.50 ± 23.89	216 ± 216
	ERSO	11776.79 ± 1854.56	38.20 ± 19.61	100 ± 194
S4	GA	1455.01 ± 0.00	47.00 ± 0.00	299 ± 168
	PSO	2793.19 ± 590.73	42.00 ± 2.21	292 ± 213
	RSO	1538.11 ± 75.08	47.40 ± 0.70	108 ± 198
	ERSO	1560.71 ± 72.94	47.70 ± 0.48	56 ± 149
S5	GA	46157.78 ± 22112.54	25.20 ± 17.14	274 ± 208
	PSO	69469.65 ± 19265.44	21.70 ± 13.72	316 ± 184
	RSO	15029.95 ± 0.00	0.00 ± 0.00	334 ± 225
	ERSO	15029.95 ± 0.00	0.00 ± 0.00	278 ± 202
S6	GA	12443.78 ± 2928.95	31.90 ± 10.86	387 ± 237
	PSO	16913.93 ± 2870.55	28.80 ± 8.85	276 ± 166
	RSO	10906.31 ± 3970.71	24.20 ± 24.04	248 ± 253
	ERSO	10309.57 ± 3692.54	29.10 ± 23.33	227 ± 233
S7	GA	3317.93 ± 0.00	47.00 ± 0.00	309 ± 232
	PSO	4710.79 ± 556.97	42.50 ± 2.95	311 ± 252
	RSO	3380.54 ± 80.82	47.40 ± 0.52	109 ± 208
	ERSO	3372.21 ± 71.43	47.00 ± 0.67	267 ± 258

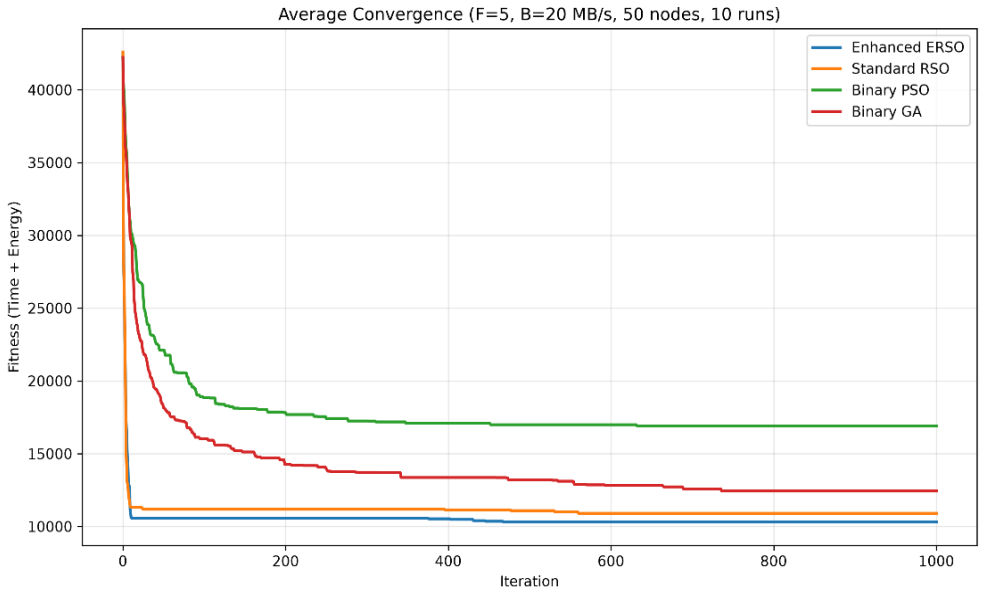


Figure 10. Convergence curves of Enhanced Rat Swarm Optimization (ERSO), Rat Swarm Optimization (RSO), Particle Swarm Optimization (PSO), and Genetic Algorithms (GA) on the 50-node weighted communication graph (WCG) under challenging conditions (F = 5, B = 20)

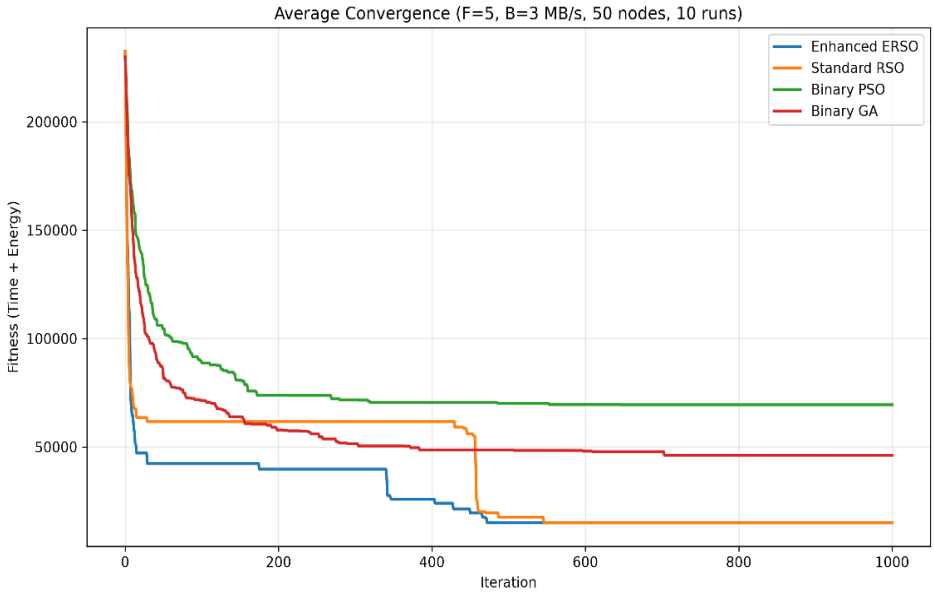


Figure 11. Convergence curves of Enhanced Rat Swarm Optimization (ERSO), Rat Swarm Optimization (RSO), Particle Swarm Optimization (PSO), and Genetic Algorithms (GA) on the 50-node weighted communication graph (WCG) under challenging conditions (F = 5, B = 3)

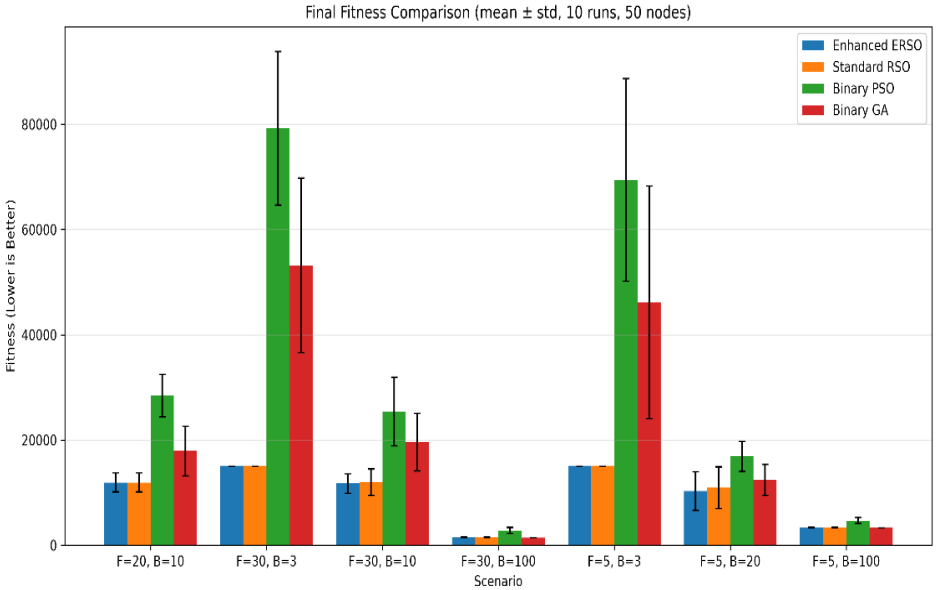


Figure 12. Fitness comparison of Enhanced Rat Swarm Optimization (ERSO), Rat Swarm Optimization (RSO), Particle Swarm Optimization (PSO), and Genetic Algorithms (GA) across all 7 scenarios on the 50-node weighted communication graph (WCG)

On the 50-node graph, GA and PSO show very high fitness values and large standard deviations. In most Scenarios, GA and PSO achieve a fitness lower than the ERSO. This confirms that simple metaheuristics struggle with complex, high-dimensional problems. In Scenarios 1, 2, 5, and 6, ERSO achieves the same optimal fitness as RSO. However, in Scenario 3, ERSO achieves 11776.79 ± 1854.56 vs. RSO's 11995.83 ± 2502.15. In Scenario 6 (F = 5, B = 20), ERSO achieves 10309.57 ± 3692.54 vs. RSO's 10906.31 ± 3970.71. This shows ERSO's ability to find better solutions to complex problems. Scenario 6 (Figure 10) represents a low-performance server (F = 5) and moderate bandwidth (20 MB/s). GA achieves 12443.78, PSO achieves 16913.93, RSO achieves 10906.31, while ERSO achieves the lowest fitness of 10309.57. This is a significant 5.5% improvement over RSO and 17% over GA. Scenario 4 is the one with high resources. With high speed-up and high bandwidth, all algorithms perform well. GA achieves the optimal fitness (1455.01 ± 0.00), while ERSO achieves 1560.71 ± 72.94. This is expected because high bandwidth and high speed-up make the offloading decision trivial. For the worst-case scenario (F = 5, B = 3) with low speed-up and low bandwidth, GA and PSO had the worst fitness (46157.78 and 69469.65) (Figure 11). ERSO and RSO both achieved 15029.95 ± 0.00 (keeping all methods local), which is the optimal decision under these conditions. A standard deviation of 0.00 means the algorithm is stable; it consistently finds the optimal solution. This result for ERSO demonstrates reliability.

The higher standard deviation of RSO shows that it is unreliable and may require more runs to achieve the optimum. The standard deviations on the 50-node graph are larger than on the 16-node graph for all algorithms, reflecting the

increased complexity of the search space. However, ERSO consistently achieves lower fitness values than the other algorithms, demonstrating its superior scalability and robustness (Figure 12).

5. CONCLUSION AND FUTURE WORK

This study presented an ERSO algorithm for intelligent and dynamic task offloading in MCC environments. Unlike conventional approaches that depend on static heuristics or deterministic partitioning, ERSO introduces a swarm intelligence-based optimization framework that adapts to changing conditions such as bandwidth variability and server load. Our approach effectively identifies optimal execution points for mobile applications by balancing energy consumption and execution time through modeling the CO process as a bi-objective optimization problem and enhancing the standard RSO algorithm with adaptive learning strategies.

Experimental results demonstrated that ERSO significantly outperforms classical offloading methods in energy savings, particularly under highly dynamic network conditions. Extensive experiments conducted on both 16-node and 50-node application graphs, under network bandwidths ranging from 3 to 100 Mbps and cloud speedup factors ranging from 5 to 30, confirmed the robustness and effectiveness of the proposed approach. Compared with the non-offloading, full offloading, and min-cut approach, as well as GA, PSO, and the standard RSO, ERSO consistently achieved superior performance in terms of overall fitness, energy consumption, and response time. In particular, under the most challenging network conditions (3 Mbps bandwidth and a cloud speedup factor of 5), the proposed algorithm achieved up to a 33% improvement in fitness over the non-offloading baseline, demonstrating its ability to maintain efficient task allocation even in adverse MCC environments.

Furthermore, ERSO shows strong convergence performance and avoids the premature convergence and local minima limitations observed in earlier RSO-based algorithms. The proposed adaptive enhancements effectively balance exploration and exploitation throughout the optimization process, resulting in improved solution quality and greater robustness under dynamic execution conditions. These findings demonstrate that ERSO constitutes a reliable and scalable optimization framework for CO in MCC.

Several directions can further improve the ERSO approach:

•Task parallelism and multi-objective offloading: Future extensions may explore the simultaneous offloading of multiple interdependent tasks, considering parallel execution across edge/cloud nodes and multi-core mobile devices.

•Hybrid optimization models: Combining ERSO with other bio-inspired optimization techniques or deep learning methods could lead to hybrid frameworks capable of learning optimal offloading strategies in real time.

•Real-world deployment: Building a real MCC simulation platform or deploying ERSO in practical mobile and edge-cloud applications will enable a more comprehensive evaluation under realistic operating conditions.

•Scalability studies: Investigating ERSO's behavior in larger and more heterogeneous environments, involving numerous users, applications, and computational tasks, is essential to further validate its scalability and robustness.

•Security and privacy constraints: Since CO often involves transmitting sensitive data, integrating privacy-preserving mechanisms and incorporating security overhead into the fitness function would further enhance the algorithm's practical applicability. In addition, considering quality of service requirements, such as latency guarantees, reliability, resource availability, and user mobility, represents a promising direction for extending the proposed framework to next-generation mobile cloud and edge computing systems.

Overall, the proposed ERSO algorithm provides an effective, adaptive, and scalable solution for CO in MCC by achieving a balanced trade-off between energy efficiency, response time, and computational performance. The encouraging experimental results demonstrate its potential for deployment in dynamic real-world MCC environments and establish a solid foundation for future research on intelligent resource management and CO strategies.

REFERENCES

[1] BankMyCell. (2025). How many phones are in the world? https://www.bankmycell.com/blog/how-many-phones-are-in-the-world, accessed on 5 Jun. 2026.

[2] Greenspector. (2019). Energy consumption of the 30 most popular mobile apps in the world. https://greenspector.com/en/energy-consumption-of-the-30-most-popular-mobile-apps-in-the-world/, accessed on 5 Jun. 2026.

[3] Yan, J.Q., He, F., Sang, Q.L., et al. (2025). Metadata-guided adaptable frequency scaling across heterogeneous applications and devices. arXiv preprint, arXiv: 2509.22707. https://doi.org/10.48550/arXiv.2509.22707

[4] Qualcomm. (2024). Faster performance, longer battery life and new features: How the NPU will turbocharge your next laptop. https://www.qualcomm.com/news/onq/2024/09/faster-performance-longer-battery-life-how-the-npu-will-turbocharge-next-laptop.

[5] Embedded. (2025). AI accelerators in embedded systems. https://www.embedded.com/ai-accelerators-in-embedded-systems/.

[6] Flexxon. (2025). eMMC vs SSD vs UFS: Storage comparison guide. https://www.flexxon.com/emmc-vs-ssd-vs-ufs-storage-comparison-guide/.

[7] Tadesse, S.S., Chiasserini, C.F., Malandrino, F. (2018). Characterizing the power cost of virtualization environments. Transactions on Emerging Telecommunications Technologies, 29(8): e3462. https://doi.org/10.1002/ett.3462

[8] Mohapatra, S., Dutt, N., Nicolau, A., Venkatasubramanian, N. (2007). DYNAMO: A cross-layer framework for end-to-end QoS and energy optimization in mobile handheld devices. IEEE Journal on Selected Areas in Communications, 25(4): 722-737. https://doi.org/10.1109/JSAC.2007.070509

[9] Rumanyika, J., Ponera, J. (2026). Awareness and practices on smartphone battery drainage: A cross-sectional study in Dodoma city higher education institutions, Tanzania. Direct Research Journal of Engineering and Information Technology, 14(1): 27-37. https://www.ajol.info/index.php/drjeit/article/view/316291.

[10] Solar energy harvesting for scalable IoT: Practical guide from industry experts at the things conference. (2025). The Things Net Work.

https://www.thethingsnetwork.org/article/solar-energy-harvesting-for-scalable-iot-practical-guide-from-industry-experts-at-the-things-conference.
[11] De, D. (2016). Mobile Cloud Computing: Architectures, Algorithms and Applications. CRC Press.
[12] Chun, B.G., Ihm, S., Maniatis, P., Naik, M., Patti, A. (2011). Clonecloud: Elastic execution between mobile device and cloud. In Proceedings of the sixth conference on Computer systems (EuroSys '11). Association for Computing Machinery, New York, NY, USA, pp. 301-314. https://doi.org/10.1145/1966445.1966473
[13] Kumar, K., Liu, J., Lu, Y. H., Bhargava, B. (2013). A survey of computation offloading for mobile systems. Mobile Networks and Applications, 18(1): 129-140. https://doi.org/10.1007/s11036-012-0368-0
[14] Wu, H.M., Knottenbelt, W.J., Wolter, K. (2019). An efficient application partitioning algorithm in mobile environments. IEEE Transactions on Parallel and Distributed Systems, 30(7): 1464-1480. https://doi.org/10.1109/TPDS.2019.2891695
[15] Ou, S., Yang, K., Liotta, A. (2006). An adaptive multi-constraint partitioning algorithm for offloading in pervasive systems. In Fourth Annual IEEE International Conference on Pervasive Computing and Communications (PERCOM'06), Pisa, Italy, pp. 125-134. https://doi.org/10.1109/PERCOM.2006.7
[16] Zhang, Y., Liu, H., Jiao, L., Fu, X.M. (2012). To offload or not to offload: An efficient code partition algorithm for mobile cloud computing. In 2012 IEEE 1st International Conference on Cloud Networking (CLOUDNET), Paris, pp. 80-86. https://doi.org/10.1109/CloudNet.2012.6483660
[17] Chen, J., Leng, Y.J., Huang, J.W. (2023). An intelligent approach of task offloading for dependent services in mobile edge computing. Journal of Cloud Computing, 12(1): 107. https://doi.org/10.1186/s13677-023-00477-9
[18] De Figueiredo, C.M.H. (2011). The P versus NP-complete dichotomy of some challenging problems in graph theory. Discrete Applied Mathematics, 160(18): 2681-2693. https://doi.org/10.1016/j.dam.2010.12.014
[19] Gu, F., Niu, J.W., Qi, Z.P., Atiquzzaman, M. (2018). Partitioning and offloading in smart mobile devices for mobile cloud computing: State of the art and future directions. Journal of Network and Computer Applications, 119: 83-96. https://doi.org/10.1016/j.jnca.2018.06.009
[20] Rahab, H., Haouassi, H., Souidi, M.E.H., Bakhouche, A., Mahdaoui, R., Bekhouche, M. (2023). A modified binary Rat Swarm Optimization algorithm for feature selection in Arabic sentiment analysis. Arabian Journal for Science and Engineering, 48: 10125-10152. https://doi.org/10.1007/s13369-022-07466-1
[21] Chandran, V., Mohapatra, P. (2024). A novel multi-strategy ameliorated quasioppositional chaotic tunicate swarm algorithm for global optimization and constrained engineering applications. Heliyon, 10(10): 30757. https://doi.org/10.1016/j.heliyon.2024.e30757
[22] Satyanarayanan, M., Bahl, P., Caceres, R., Davies, N. (2009). The case for vm-based cloudlets in mobile computing. IEEE Pervasive Computing, 8(4): 14-23. https://doi.org/10.1109/MPRV.2009.82
[23] Chun, B.G., Maniatis, P. (2009). Augmented smartphone applications through clone cloud execution. In Proceedings of the 12th Workshop on Hot Topics in Operating Systems (HotOS), pp. 8-11. https://www.usenix.org/legacy/events/hotos09/tech/full_papers/chun/chun.pdf.
[24] Zhao, B., Xu, Z., Chi, C.X., Zhu, S.C., Cao, G.H. (2011). Mirroring smartphones for good: A feasibility study. In Mobile and Ubiquitous Systems: Computing, Networking, and Services, pp. 26-38. https://doi.org/10.1007/978-3-642-29154-8_3
[25] Sharma, R., Gagandeep, D. (2017). Computation offloading in mobile cloud computing. International Journal of Current Trends in Science and Technology, 7(12): 20501-20510.
[26] Kemp, R., Palmer, N., Kielmann, T., Bal, H. (2010). Cuckoo: A computation offloading framework for smartphones. In Mobile Computing, Applications, and Services, pp. 59-79. https://doi.org/10.1007/978-3-642-29336-8_4
[27] Cuervo, E., Balasubramanian, A., Cho, D.K., et al. (2010). Maui: Making smartphones last longer with code offload. In Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services (MobiSys), New York, NY, USA, pp. 49-62. https://doi.org/10.1145/1814433.181444
[28] Maray, M., Shuja, J. (2022). Computation offloading in mobile cloud computing and mobile edge computing: Survey, taxonomy, and open issues. Mobile Information Systems, 2022(1): 1121822. https://doi.org/10.1155/2022/1121822
[29] Seifaddini, O., Abdullah, A., Hussin, M., Muhammed, A. (2014). Performance assessment of mobile computation offloading. In 2014 4th World Congress on Information and Communication Technologies (WICT), Melaka, Malaysia, pp. 129-133. https://doi.org/10.1109/WICT.2014.7077316
[30] Rudenko, A., Reiher, P., Popek, G.J., Kuenning, G.H. (1998). Saving portable computer battery power through remote process execution. ACM SIGMOBILE Mobile Computing and Communications Review, 2(1): 19-26. https://doi.org/10.1145/584007.584008
[31] Liu, J., Ahmed, E., Shiraz, M., Gani, A., Buyya, R., Qureshi, A. (2015). Application partitioning algorithms in mobile cloud computing: Taxonomy, review and future directions. Journal of Network and Computer Applications, 48: 99-117. https://doi.org/10.1016/j.jnca.2014.09.009
[32] Rekha, P.M., Dakshayini, M. (2019). Efficient task allocation approach using genetic algorithm for cloud environment. Cluster Computing, 22(4): 1241-1251. https://doi.org/10.1007/s10586-019-02909-1
[33] Ramasubbareddy, S., Swetha, E., Luhach, A.K., Srinivas, T.A.S. (2021). A multi-objective genetic algorithm-based resource scheduling in mobile cloud computing. International Journal of Cognitive Informatics and Natural Intelligence, 15(3): 58-73. https://doi.org/10.4018/IJCINI.20210701.oa5
[34] Pandit, D., Chattopadhyay, S., Chattopadhyay, M., Chaki, N. (2014). Resource allocation in cloud using simulated annealing. In 2014 Applications and Innovations in Mobile Computing (AIMoC), Kolkata, India, pp. 21-27. https://doi.org/10.1109/AIMOC.2014.6785514
[35] Huang, T.T., Li, S.J., Gao, X.Y. (2020). Computing resource allocation and offloading method based on simulated annealing algorithm. In 2020 International

Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC), Chongqing, China, pp. 276-282. https://doi.org/10.1109/CyberC49757.2020.00052
[36] Mahdi, M., YP, Z., Guoning, C.N. (2023). Resource allocation in cloud computing using genetic algorithm and neural network. In 2023 IEEE 8th International Conference on Smart Cloud (SmartCloud), Tokyo, Japan, pp. 25-32. https://doi.org/10.1109/SmartCloud58862.2023.00013
[37] Shao, K.L., Fu, H., Wang, B. (2023). An efficient combination of genetic algorithm and particle swarm optimization for scheduling data-intensive tasks in heterogeneous cloud computing. Electronics, 12(16): 3450. https://doi.org/10.3390/electronics12163450
[38] Gan, G.N., Huang, T.L., Gao, S. (2010). Genetic simulated annealing algorithm for task scheduling based on cloud computing environment. In 2010 International Conference on Intelligent Computing and Integrated Systems, Guilin, China, pp. 60-63. https://doi.org/10.1109/ICISS.2010.5655013
[39] Tanha, M., Hosseini Shirvani, M., Rahmani, A.M. (2021). A hybrid meta-heuristic task scheduling algorithm based on genetic and thermodynamic simulated annealing algorithms in cloud computing environments. Neural Computing and Applications, 33: 16951-16984. https://doi.org/10.1007/s00521-021-06289-9
[40] Zhou, W.Q., Chen, L.Y., Tang, S.P., et al. (2022). Offloading strategy with PSO for mobile edge computing based on cache mechanism. Cluster Computing, 25: 2389-2401. https://doi.org/10.1007/s10586-021-03414-0
[41] You, Q., Tang, B. (2021). Efficient task offloading using particle swarm optimization algorithm in edge computing for industrial Internet of Things. Journal of Cloud Computing, 10: 41. https://doi.org/10.1186/s13677-021-00256-4
[42] Peng, Q.X., Chen, X.D., Huang, Y.J., Ma, S.K., He, Z.L. (2023). Particle swarm optimization-based task migration in mobile-edge cloud computing. In 2023 IEEE International Conferences on Internet of Things (iThings), Green Computing & Communications (GreenCom), Cyber, Physical & Social Computing (CPSCom), Smart Data (SmartData), and Congress on Cybermatics (Cybermatics), Danzhou, China, pp. 616-623. https://doi.org/10.1109/iThings-GreenCom-CPSCom-SmartData-Cybermatics60724.2023.00113
[43] Dong, S., Xia, Y.J., Kamruzzaman, J. (2022). Quantum particle swarm optimization for task offloading in mobile edge computing. IEEE Transactions on Industrial Informatics, 19(8): 9113-9122. https://doi.org/10.1109/TII.2022.3225313
[44] Hemanth, S.V., Kirubha, D., Reddy, S.R., Chelladurai, T., Soundari, A.G., Amirthayogam, G. (2024). Multi-objective ant colony optimization technique for task scheduling in cloud computing. In 2024 3rd International Conference on Applied Artificial Intelligence and Computing (ICAAIC), Salem, India, pp. 830-835. https://doi.org/10.1109/ICAAIC60222.2024.10575423
[45] Singh, H., Bhasin, A., Kaveri, P.R. (2021). QRAS: Efficient resource allocation for task scheduling in cloud computing. SN Applied Sciences, 3: 474. https://doi.org/10.1007/s42452-021-04489-5
[46] Kniazhyk, T., Muliarevych, O. (2023). Ant colony optimization for resource allocation in cloud computing environments. In 2023 IEEE 12th International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS), Dortmund, Germany, pp. 368-372. https://doi.org/10.1109/IDAACS58523.2023.10348763
[47] Huang, M.X., Zhai, Q.H., Chen, Y.J., Feng, S.L., Shu, F. (2021). Multi-objective whale optimization algorithm for computation offloading optimization in mobile edge computing. Sensors, 21: 2628. https://doi.org/10.3390/s21082628
[48] Behera, I., Sobhanayak, S., Almakayeel, N. (2025). Task offloading in heterogeneous mobile edge computing using meta heuristic approach for sustainable smart city applications. Physical Communication, 73: 102893. https://doi.org/10.1016/j.phycom.2025.102893
[49] Yuan, H.T., Hu, Q.L., Bi, J., Lü, J.H., Zhang, J., Zhou, M.C. (2023). Profit-optimized computation offloading with autoencoder-assisted evolution in large-scale mobile-edge computing. IEEE Internet of Things Journal, 10(3): 11896-11909. https://doi.org/10.1109/JIOT.2023.3244665
[50] Bi, J., Wang, Z.Q., Yuan, H.T., Zhang, J., Zhou, M.C. (2024). Cost-minimized computation offloading and user association in hybrid cloud and edge computing. IEEE Internet of Things Journal, 11(9): 16672-16683. https://doi.org/10.1109/JIOT.2024.3354348
[51] Yuan, H.T., Hu, Q.L., Wang, S., Bi, J., Buyya, R., Lü, J.H. (2024). Cost-optimized task offloading for dependent applications in collaborative edge and cloud computing. IEEE Internet of Things Journal, 12(9): 12975-12988. https://doi.org/10.1109/JIOT.2024.3522964
[52] Abolfazli, S., Sanaei, Z., Sanaei, M.H., Shojafar, M., Gani, A. (2015). Mobile cloud computing: The state-of-the-art, challenges, and future research. In Encyclopedia of Cloud Computing.
[53] Geng, Y.L., Yang, Y., Cao, G.H. (2018). Energy-efficient computation offloading for multicore-based mobile devices. In IEEE INFOCOM 2018-IEEE Conference on Computer Communications, Honolulu, HI, USA, pp. 46-54. https://doi.org/10.1109/INFOCOM.2018.8485875
[54] Ji, T.X., Luo, C.Q., Yu, L.X., Wang, Q.L., Chen, S.H., Thapa, A. (2022). Energy-efficient computation offloading in mobile edge computing systems with uncertainties. IEEE Transactions on Wireless Communications, 21(8): 5717-5729. https://doi.org/10.1109/TWC.2022.3142685
[55] Wu, H.M., Knottenbelt, W., Wolter, K., Sun, Y. (2016). An optimal offloading partitioning algorithm in mobile cloud computing. In Quantitative Evaluation of Systems, pp. 311-328. https://doi.org/10.1007/978-3-319-43425-4_21
[56] Li, G.S., Lin, Q.Y., Wu, J.H., Zhang, Y., Yan, J.H. (2019). Dynamic computation offloading based on graph partitioning in mobile edge computing. IEEE Access, 7: 185131-185139. https://doi.org/10.1109/ACCESS.2019.2960887
[57] Geoffray, N., Thomas, G., Folliot, B. (2006). Transparent and dynamic code offloading for java applications. In On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE. pp. 1790-1806.

https://doi.org/10.1007/11914952_51
[58] Seo, S.H., Straub, J. (2017). Comparative analysis of graph partitioning algorithms in context of computation offloading. In 2017 IEEE International Conference on Electro Information Technology (EIT), Lincoln, NE, USA, pp. 1-7. https://doi.org/10.1109/EIT.2017.8053425
[59] Verbelen, T., Stevens, T., De Turck, F., Dhoedt, B. (2013). Graph partitioning algorithms for optimizing software deployment in mobile cloud computing. Future Generation Computer Systems, 29: 451-459. https://doi.org/10.1016/j.future.2012.07.003

NOMENCLATURE

$G = (V, E)$	Application call graph
$V = \{v_1, ..., v_n\}$	Set of methods (Vertices)
$E = \{(v_i, v_j)\}$	Communication dependencies (Edges)
L_i	Local execution cost of v_i
R_i	Remote execution cost of v_i
C_{ij}	Communication cost between v_i and v_j
B	Network bandwidth
F	Cloud speedup factor
P_L	Power consumption for local execution
P_R	Power consumption during remote execution
P_{Com}	Power consumption for data transfer during communication
T_{Local}	Total local execution time
T_{Remote}	Total remote execution time
T_{Com}	Total communication time
T_{Total}	Total execution time
E	Total energy consumption
w_t	Weight for execution time
w_e	Weight for energy consumption
$U(-0.1, 0.1)$	A uniform perturbation in the range [−0.1, 0.1].
M_{random}	Random matrix
x	Current iteration number
$max_{iteration}$	Maximum iterations
nb_{agents}	Number of agents (Particles)
P_i	Position of agent i
G_{best}	Global best solution
P_{best}	Personal best for agent i
A, C	Adaptive parameters in RSO
R	Random value in [1, 5]
$rand()$	Random value in [0, 1]