Expert Systems With Applications 224 (2023) 119956

Contents lists available at ScienceDirect

Expert Systems With Applications

journal homepage: www.elsevier.com/locate/eswa

An adaptive parallel evolutionary algorithm for solving the uncapacitated facility location problem

Emrullah Sonuç [a,b,*], Ender Özcan [a]

[a] Computational Optimisation and Learning (COL) Lab, School of Computer Science, University of Nottingham, Nottingham, NG8 1BB, United Kingdom
[b] Department of Computer Engineering, Faculty of Engineering, Karabuk University, Karabük, 78050, Türkiye

ARTICLE INFO

Keywords:
Adaptive operator selection
Metaheuristics
Combinatorial optimisation
Parallel algorithms

ABSTRACT

Metaheuristics, providing high level guidelines for heuristic optimisation, have successfully been applied to many complex problems over the past decades. However, their performances often vary depending on the choice of the initial settings for their parameters and operators along with the characteristics of the given problem instance handled. Hence, there is a growing interest into designing adaptive search methods that automate the selection of efficient operators and setting of their parameters during the search process. In this study, an adaptive binary parallel evolutionary algorithm, referred to as ABPEA, is introduced for solving the uncapacitated facility location problem which is proven to be an NP-hard optimisation problem. The approach uses a unary and two other binary operators. A reinforcement learning mechanism is used for assigning credits to operators considering their recent impact on generating improved solutions to the problem instance in hand. An operator is selected adaptively with a greedy policy for perturbing a solution. The performance of the proposed approach is evaluated on a set of well-known benchmark instances using ORLib and M*, and its scaling capacity by running it with different starting points on an increasing number of threads. Parameters are adjusted to derive the best configuration of three different rewarding schemes, which are instant, average and extreme. A performance comparison to the other state-of-the-art algorithms illustrates the superiority of ABPEA. Moreover, ABPEA provides up to a factor of 3.9 times acceleration when compared to the sequential algorithm based on a single-operator.

1. Introduction

In the last two decades, (meta)heuristics have been commonly and increasingly used for tackling a range of computationally hard real-world optimisation problems in various areas, such as logistics, transportation, finance, military defence, and energy. The exact solvers that guarantee optimality often fail, or require an impractical amount of time to produce a solution to a given instance (Sörensen & Glover, 2013). Therefore, inexact search algorithms, such as metaheuristics are alternatively preferred. Although such approaches do not guarantee an optimal solution, they can provide a near-optimal solution within a reasonable amount of time (Swan et al., 2022). Metaheuristics can be broadly divided into two groups: (i) single-point-based methods making use of a single active solution during the search process, such as, simulated annealing, tabu search, and (ii) population-based methods utilising multiple solutions simultaneously during the search process. Evolutionary Algorithms (EAs), such as, Genetic Algorithms, Memetic Algorithms, Differential Evolution and Particle Swarm Optimisation, attempt to find the (near-) optimal solution by transforming interacting solutions in a population at each evolutionary cycle (Talbi, 2009).

Designing a metaheuristic utilising multiple operators, exploiting the strength of each one of them, is another challenge. The success of each operator may differ while solving different problem instances with different characteristics (Sevaux, Sörensen, & Pillay, 2018). Choosing the best operator on its own can be formulated as an optimisation problem and this choice has usually crucial impact on the performance of the metaheuristic and so the quality of the resultant solutions (Fialho, 2010). The usual approach is to either carry out some initial experiments for tuning the metaheuristic to obtain the best algorithmic configuration offline, or include an adaptation method controlling the operators in an online manner, choosing the best operator at each decision point (Cowling, Kendall, & Soubeiga, 2000; Fialho, 2010). In many of the previously proposed adaptation methods, reinforcement learning has been used, where each operator is given a utility score based on a rewarding scheme during the search process. Then a selection method is used to choose the best operator at a given time considering the utility scores of all operators (Drake, Kheiri, Özcan, & Burke, 2020).

* Corresponding author at: Department of Computer Engineering, Faculty of Engineering, Karabuk University, Karabük, 78050, Türkiye.
E-mail addresses: esonuc@karabuk.edu.tr (E. Sonuç), ender.ozcan@nottingham.ac.uk (E. Özcan).

https://doi.org/10.1016/j.eswa.2023.119956
Received 22 July 2022; Received in revised form 7 March 2023; Accepted 21 March 2023
Available online 23 March 2023

As an additional challenge, the size of the combinatorial optimisation problem instances has been increasing, leading to significant interest into the design and development of parallel/distributed metaheuristics for scaling up (Talbi, 2015). The goal is to obtain higher quality solutions than the sequential algorithms by the use of parallelism. Hence, the motivation behind using a parallel metaheuristic is to solve complex optimisation problems more effectively and faster (Crainic, 2019). There are various models to develop multi-threaded search algorithms arranging the communication between multiple processors, such as island model or cellular model (Harada & Alba, 2020; Luke, 2013). Also, searching for a new solution can be achieved either dependently or independently. During the search process, threads can share their information for a better exploration and exploitation capacity (Sonuc, Sen, & Bayir, 2018). The efficacy of re-starting an algorithm from a different point in the search landscape when the search stagnates has also been observed enabling a search algorithm to explore different regions of the solution space (Alba, Luque, & Nesmachnow, 2013). There is a growing number of studies on multi-threaded parallel metaheuristics for solving the Uncapacitated Facility Location Problem (UFLP).

Balinski (1964) and Stollsteimer (1961) introduced UFLP which is a well-known NP-hard real-world combinatorial optimisation problem (Cornuéjols, Nemhauser, & Wolsey, 1983; Efroymson & Ray, 1966). Solving UFLP involves in deciding which facilities to open while minimising the overall cost given a set of facilities and different levels of demand from customers. Considering the results from previously proposed algorithms, there is still a need for designing more effective search methods for UFLP (see Section 2.2). In this study, we propose a multi-threaded parallel adaptive evolutionary algorithm controlling multiple operators with restart capability for solving UFLP. This work is an extension of the study in Sonuç (2021), introducing several significant modifications outlined as follows:

- We have used three low-level operators instead of a single one to amalgamate their strengths.
- An adaptive operator selection mechanism based on reinforcement learning is used to improve the quality of the results by choosing and applying the appropriate operator during the search process.
- We have investigated the performance of different rewarding schemes, namely *instant*, *average* and *extreme* using a different window size are empirically investigated with a greedy policy for operator selection.
- We have implemented our method using multi-threaded programming in OpenMP. We have then tested various number of threads (32, 64, 128, 256 and 512), where each thread uses a different starting solution to achieve better diversification at the beginning of the search.

The rest of the paper is organised as follows. Section 2 provides the mathematical formulation of UFLP and previous work covering some selected exact and metaheuristic approaches to the problem. The proposed parallel framework with an adaptive strategy is described in Section 3. Experimental results on benchmark problems are given in Section 4. Finally, Section 5 concludes the study and suggests directions for future work.

2. Uncapacitated facility location problem

2.1. Problem formulation

As a decision problem, UFLP has a set of customers and a set of facilities. Each customer has a location, and it demands service from the facility closest to it. The total cost consists of the service cost for each customer and the opening cost of the facility to be opened. There is no limit to how many customers demand services from a facility.

Suppose n is the number of facilities and m is the number of customers, the mathematical formulation of the UFLP can be given as follows (Erlenkotter, 1978):

$$minimise \quad \sum_{i=1}^{n}\sum_{j=1}^{m} c_{ij}x_{ij} + \sum_{i=1}^{n} f_i y_i \tag{1}$$

s.t.

$$\sum_{i=1}^{n} x_{ij} = 1, \quad j = 1, 2, \dots, m \tag{2}$$

$$x_{ij} - y_i \le 0, \quad i = 1, \dots, n, \quad j = 1, \dots, m \tag{3}$$

$$x_{ij} \in \{0, 1\}, \quad i = 1, \dots, n, \quad j = 1, \dots, m \tag{4}$$

$$y_i \in \{0, 1\}, \quad i = 1, \dots, n \tag{5}$$

where c_{ij} is the service cost demand from facility i for customer j and f_i is the opening cost of a facility at location i. The purpose of UFLP is to minimise the total cost including service costs and opening costs which are first and second terms in Eq. (1), respectively. Constraint (2) indicates that each customer can be served by only one facility. Constraint (3) ensures that customers cannot demand a service from a facility which is closed. Constraint (4) and (5) defines that decision variables can be either 0 or 1.

2.2. Related work

A range of different exact algorithms has been developed for solving UFLP since 60 s. For example, Efroymson and Ray (1966) developed a branch-and-bound algorithm for solving UFLP. Akinc and Khumawala (1977) then improved on their approach by introducing a more efficient method for solving linear programming at the nodes, reducing the amount of computation required. Schrage (1975) presented a tighter linear programming formulation for the FLP and applied a specialised linear programming algorithm for variable upper bound constraints. Galvão and Raggi (1989) proposed a three-stage exact method that combines a primal–dual algorithm with a sub-gradient optimisation and a branch-and-bound algorithm. Although exact methods can solve some of the UFLP instances, their overall computational requirements are often not ideal. They either require a large amount of memory returning a near-optimal solution at the end or an enormous time to compute, hence they are generally suitable for solving small-scale UFLP instances (Hoefer, 2003). The previous work shows that exact solvers, for example, CPLEX cannot obtain the optimal solutions using even the relaxed integer programming formulations of UFLP for the relatively ‘large’ instances, involving from 100 to 500 facilities as discussed in Monabbati (2014) and Monabbati and Kakhki (2015). As discussed before, metaheuristics, such as EAs cannot guarantee optimality in combinatorial optimisation. However, they have been highly preferred over exact methods, as they can achieve a reasonable near-optimal solution (in some cases even a ‘perfect’ solution) in a reasonable amount of time for a given UFLP instance (Akan, Agahian, & Dehkharghani, 2022; Durgut & Aydin, 2021; Sonuç, 2021) when the exact methods fail.

The number of metaheuristics for UFLP has been growing rapidly in recent years (Glover, Hanafi, Guemri, & Crevits, 2018; Zhang, He, Ouyang, & Li, 2023). In the majority of those studies, the main motivation has been creating new perturbative operators to produce high-quality solutions for UFLP. Hence, transfer functions, similarity mechanisms, bitwise operations and elitist mutation-based operators have been proposed in the design of many metaheuristics. If a metaheuristic handles a solution as a continuous variable, then the solution must be converted from continuous to the binary domain before the evaluation (He, Zhang, Mirjalili, & Zhang, 2022). Transfer functions, such as convergence functions can be used to map decision variables to the interval [0,1] (He et al., 2022; Kennedy & Eberhart, 1997; Luh, Lin, & Lin, 2011). However, García, Crawford, Soto, and Astorga (2019) concluded that some regions in the search space may not be discovered with such transformations. Thus, several proposed mechanisms use a binary vector as a decision variable and carry out search directly in the binary search space. Similarity (or dissimilarity) based operators accept two different binary vectors as input and calculate the distance between them (Choi, Cha, & Tappert, 2010). This mechanism was used in several EAs, such as differential evolution (Husseinzadeh Kashan, Husseinzadeh Kashan, & Nahavandi, 2013) and others (Akan et al., 2022; Baş & Ülker, 2020; Cinar & Kiran, 2018; Kashan, Nahavandi, & Kashan, 2012), previously. Exclusive or, denoted as XOR from now, is a logical operator and it is commonly used to develop binary operators (Aslan, Gunduz, & Kiran, 2019; Baş & Ülker, 2020; Cinar & Kiran, 2018; Durgut, 2021; Kaya, 2022; Kiran, 2021; Korkmaz & Kiran, 2018; Sonuç, 2021). XOR-based operators handle a binary solution directly and generate a new solution using different neighbours. Sonuç (2021) observed that it is a highly effective operator to get better intensification in the local region. However, an additional strategy is usually needed, as the search might get stuck at a local optimum. Thus, mutation can be used as an escape mechanism in such situations (Hakli, 2020; Hakli & Ortacay, 2019; Tohyama, Ida, & Matsueda, 2011; Tsuya, Takaya, & Yamamura, 2017).

Although some of the previously proposed use multiple operators, they either randomly choose the operator to generate a new solution or use it for a different purpose, like balancing exploration and exploitation. Adaptive EAs are worth investigating for solving UFLP, additionally considering that there are only a few studies in the scientific literature (Durgut & Aydin, 2021). On the other hand, developing a parallel approach to the problem (Talbi, 2015) is another option to improve the quality of results and design more effective algorithms. Several parallel algorithms (Cura, 2010; Wang, Wang, Yan, & Wang, 2008a; Wang, Wu, Ip, Wang, & Yan, 2008b) were proposed for UFLP, but they were not capable of finding optimal or near-optimal solutions.

3. Adaptive Binary Parallel Evolutionary Algorithm (ABPEA)

3.1. Main framework

The ABPEA framework is a multi-threaded parallel evolutionary algorithm embedding a restart method that uses binary encoded solutions conducting a search with shared memory using a master–slave model. The framework is realised using OpenMP which is one of the effective tools for programming parallel metaheuristics (Gmys, Carneiro, Melab, Talbi, & Tuyttens, 2020). An OpenMP program starts with a single thread (master thread) that creates threads (fork) running in parallel regions, such that statements in a parallel block are executed in parallel by each thread. A parallel region executes until all threads have synchronised and joined to the master thread at the end of the region (Banos, Ortega, Gil, de Toro, & Montoya, 2016).

In this study, we propose an adaptive local search mechanism in that each thread shares the part of or whole positions at each iteration according to the operator employed. Each thread generates a new solution using an operator from the pool, and this process is run by adaptive local search. We outline the adaptive binary parallel evolutionary algorithm (ABPEA) for UFLP in Algorithm 1. ABPEA begins with the initialisation stage by setting the number of threads (line 1). The first parallel region ensures that each thread initialises the starting solution (lines 2–4). Section 3.2 describes how the proposed method initialises a solution for each thread. Later, the sequential part calculates reward and assigns each operator's credit, and probability values (lines 5–7). After the initialisation, the adaptive local search stage begins, and each thread employs an operator selected from the pool. The operator generates the new solution, and the thread i updates $r_{i,t}$ (reward) and $q_{i,t}$ (credit) at time t (lines 9–15). Section 3.3 explains the working principles of adaptive local search. Then all threads are synchronised in line 15, and the master thread finds the best position among threads and updates the probabilities of the operators (lines 16–20). Before the new iteration begins, threads check if there is no improvement among all solutions after a predetermined iteration, then the multi-start mechanism steps in to create new random solutions (lines 11–23). Finally, the best solution (S^*) returns as an output (line 26).

Algorithm 1 The pseudocode of ABPEA

```
    Require: Problem Instance (PI), Number of Threads (NUM_THREADS)
    Output: The best solution found $S^*$
    Initialisation Phase:
 1: omp_set_num_threads (NUM_THREADS);
 2: #pragma omp parallel begin
 3: $S \leftarrow$ InitialSolution (PI)
 4: end omp parallel
 5: $r_{i,0} \leftarrow$ Reward_Initialisation()
 6: $q_{i,0} \leftarrow$ Credit_Initialisation()
 7: $p_{i,0} \leftarrow$ Probability_Initialisation()
 8: while $NFE \leq 80000$ do
 9:     #pragma omp parallel begin
10:     $S' \leftarrow$ AdaptiveLocalSearch (action,$S$)
11:     if $f(S') \leq f(S)$ then
12:         $S \leftarrow S'$
13:     end if
14:     $r_{i,t} \leftarrow$ Update_Reward($S$)
15:     #omp barrier
16:     if omp_get_thread_num()=0 then
17:         $S^* \leftarrow S$
18:         $q_{i,t} \leftarrow$ Update_Credit($r_{i,t}$)
19:         $p_{i,t} \leftarrow$ Update_Probability($q_{i,t}$)
20:     end if
21:     if $trial == 25$ then
22:         $S \leftarrow$ Randomised_Initialisation()
23:     end if
24:     end omp parallel
25: end while
26: return ($S^*$)
```

3.2. Starting solution

All possible solutions are feasible for UFLP, except for the case when all bits are set to 0. In order to ensure to get better exploration for initial solutions, different starting positions are assigned for each thread. Also, better diversification can be allowed by increasing the number of different starting solutions depending on the number of threads. If the number of threads is equal to or more than the problem size, then each thread is only responsible to open a single facility corresponding to its number. In other cases, all threads start solving with a random position using the Bernoulli process. Accordingly, a random number is generated in the range (0,1), and if this number is greater than or equal to 0.5, it is assigned as 1, otherwise 0.

3.3. Adaptive local search

There are various methods used in the search algorithms for adaptively choosing an operator. There is a growing number of studies focusing on the Machine Learning methods for this purpose, which can be mainly categorised as online and offline learning methods. Offline methods operate in a train-and-test fashion. This study focuses on the online methods as a part of a parallel evolutionary algorithm that gets feedback on the performance of operators during the search process, and decide which operator should be applied in the next step (Talbi, 2021). For example, Adaptive Operator Selection (AOS) (Fialho, 2010) uses two components: (i) credit assignment, measuring the empirical performance of operators and (ii) operator selection, deciding which operator to apply next. There are different credit assignment mechanisms, such as Q-learning-based (Gong, Fialho, Cai, & Li, 2011; Wauters, Verbeeck, Causmaecker, & Berghe, 2013), Score-based (Peng, Zhang, Gajpal, & Chen, 2019), Extreme Value-based (Fialho, 2010) were presented previously. Reinforcement Learning (RL) is one of the most efficient ways for credit assignment using the reward/punish mechanism (Song, Triguero, & Özcan, 2019). In RL algorithms, an agent acts with the environment to learn the target role. It takes some actions from the environment as a response to maximise the reward and minimise the risk (Sutton & Barto, 2018). In AOS, the feedback information is used as performance criteria such as objective function value, success rate, diversity of solutions, CPU time and so on. This line of research also relates to the studies on perturbative selection hyper-heuristics (Drake et al., 2020), where usually a single-point-based search is employed, attempting to improve a single candidate solution iteratively controlling multiple perturbative operators and their settings, and accepting/rejecting the newly generated solutions. For example, Cowling et al. (2000) has introduced *choice function* hyper-heuristic which is based on reinforcement learning additionally considering the pairwise performances of operators that are successively invoked during the search as well as the unused operators. The studies in Di Gaspero and Urli (2012) and Özcan, Misir, Ochoa, and Burke (2012) provide empirical results applying reinforcement learning testing various reward/penalty schemes along with heuristics selection on different problem domains. The components that we have used for adaptation as a part of our approach, including the operators, credit assignment, and selection methods are discussed in the following sections.

3.3.1. Operators

The number of operators in an adaptive approach is another challenge because its performance may get poorer when the number of operators increases (Karimi-Mamaghan, Mohammadi, Meyer, Karimi-Mamaghan, & Talbi, 2022). We use the following three perturbative operators in this study:

Memory Best Operator (memOp). memOp is a binary operator that generates a new solution based on the best positions for different individuals. It uses bitwise operations, XOR and AND, and their notations are $\otimes$ and $\odot$, respectively. φ is a random binary number and the equation of memOp is as follows (Sonuç, 2021):

$$x_i = x_i^{best} \otimes (\varphi \odot (x_i^{best} \otimes x_j^{best})) \tag{6}$$

Dissimilarity Based Operator (disOp). Similarity mechanisms play a critical role to measure between binary vectors in many problems (Choi et al., 2010). Jaccard's similarity is mainly used to generate a new solution in BOPs (Kashan et al., 2012). In this study, we choose the Dice's similarity as it allows to open more facilities (Baş & Ülker, 2020) and it is shown in Eqs. (7) and (8).

$$Similarity(\mathbf{x}_i, \mathbf{x}_j) = \frac{2 \times M_{11}}{2 \times M_{11} + M_{01} + M_{10}} \tag{7}$$

$$\begin{aligned} Dissimilarity(\mathbf{x}_i, \mathbf{x}_j) &= 1 - Similarity(\mathbf{x}_i, \mathbf{x}_j) \\ &= 1 - \frac{2 \times M_{11}}{2 \times M_{11} + M_{01} + M_{10}} \end{aligned} \tag{8}$$

where M_{11} represents the number of bits that are equal to 1 at the same positions in both $\mathbf{x}_i$ and $\mathbf{x}_j$, while M_{01} and M_{10} are determined in the same way.

A new neighbour solution is generated using Eq. (9). X_{new} is the new individual and it is generated using similarity mechanism with existing solutions X_i and X_j. ϕ is random scaling factor for normalisation and Eq. (10) balances the similarities in the Eq. (9) (Kashan et al., 2012).

$$Dissimilarity(\mathbf{X}_{new}, \mathbf{x}_i) \approx \phi \times Dissimilarity(\mathbf{x}_i, \mathbf{x}_j) \tag{9}$$

$$\min \ | \ Dissimilarity(\mathbf{X}_{new}, \mathbf{x}_i) \ - \ \phi \times Dissimilarity(\mathbf{x}_i, \mathbf{x}_j)| \tag{10}$$

s.t.

$$\begin{aligned} &M_{11} + M_{01} = n_1 \\ &M_{10} \le n_0 \\ &\{M_{10}, M_{11}, M_{01}\} \ge 0 \text{ and } \in \mathbb{Z} \end{aligned} \tag{11}$$

where n_0 and n_1 represent the number of bits with a value of 0 and 1 in $\mathbf{x}_i$. The model aims to achieve best value for ϕ to normalise $Dissimilarity(\mathbf{x}_i, \mathbf{x}_j)$. Further information can be found in the original study (Kashan et al., 2012).

Elitist Operator (eltOp). eltOp is an unary operator takes some positions from the neighbour's solution (if it has a better solution than the current one) and copies them to the current solution. The number of the positions decreases during the iterations to balance exploration.

$$x_i^d = x_j^d \tag{12}$$

where d is the vector that declares which positions are copied from the neighbour solution. In this study, the d is linearly decreased from 0.8 to 0.2 throughout iterations (Durgut, 2021; Hakli, 2020).

3.3.2. Credit assignment mechanism

According to the reward strategy, each operator has a credit updated during the search process. It guides the selection process in which the operator is selected and employed. We use objective function value (OFV) as a performance criterion for the reward ($r_i(t)$), and it computes at time t as follows:

$$r_i(t) = \frac{f(x_{current})}{g_{best}}(f(x_{current}) - f(x_{new})) \tag{13}$$

where $f(x_{current})$ and $f(x_{new})$ are the existing solution and the new solution generated by the selected operator, respectively. g_{best} is the best OFV so far, and it helps to normalise the reward value (Durgut & Aydin, 2021). AOS presents different reward strategies with respect to the window size (W): *instant*, *average* and *extreme*. *Instant* considers the current(instantaneous) reward, which means W = 1. According to W, *extreme* and *average* means the best reward value and the mean reward value, respectively. After this, credit assignment is performed using the formula given below (Fialho, 2010):

$$q_i(t+1) \ = \ (1-\alpha) \ q_i(t) \ + \ \alpha \ r_i(t), \tag{14}$$

where $q_i(t)$ and $q_i(t+1)$ are current and updated credits for the operator i and $\alpha \in [0, 1]$ is the adaptation rate. As a result, the credit assignment mechanism measures how well the operator's performance for the current instance or landscape.

3.3.3. Operator selection strategy

We perform Probability Matching (PM) approach to update the selection probability of each operator. PM uses a minimum selection probability score, $p_{min} \in (0, 1)$, to assign all operators a chance to be selected regardless of their credits. It calculates the probabilities as is follows:

$$p_i(t) = p_{min} + (1 \ - \ K \ . \ p_{min}) \ \frac{q_i(t)}{\sum_{j=1}^{K} q_j(t)} \tag{15}$$

where $p_i(t)$ is the selection probability of operator i and K is the number of operators. After this, we use a $\varepsilon - greedy$ policy to ensure the intensification and it can be defined as below (dos Santos, de Melo, Neto, & Aloise, 2014).

$$i^* = \begin{cases} \arg\max_{i=1,\ldots,K} \ q_i(t), & \text{with probability } 1 \text{ - } \varepsilon, \\ \text{any other operator}, & \text{with probability } \varepsilon. \end{cases} \tag{16}$$

4. Experimental results

We have conducted a set of experiments to evaluate the performance of ABPEA. In the first set of experiments, we have carried out parameter tuning. Then the effect of the number of threads on the performance of ABPEA is analysed. The best configuration our of the first two sets of experiments are used in our last set of experiments comparing our proposed approach to the state-of-the-art methods. Computational tests were performed on a computer with a quad-core processor (Intel(R) Core(TM) i7-4790 CPU @ 3.60 GHz), with 16 GB of RAM, running the Windows 10 64-bit operating system. ABPEA is implemented in standard C.[1] using gcc compiler with OpenMP version 4.5.

To test the proposed method, we used two different benchmark data sets for the computational experiments: ORLib (Beasley, 1990) and M*(Kratica, Tošic, Filipović, & Ljubić, 2001). ORLib is the most well-known benchmark in the area, which includes 15 problem instances, and M* contains 20 problem instances. Each problem instance is denoted as *iname_n_m*, where *iname*, n, and m indicate the instance label, number of facilities, and number of customers, respectively. The value of n changes between 16 to 500 and m changes between 50 to 1000 for all instances.

Each experiment is run for 30 and 100 trials for each instance of the ORLib and M*, respectively. Each run terminates when the maximum number of objective function evaluations ($MaxNFEs$), that is 80,000 is exceeded. If no improvement is achieved for 25 iterations, restart is employed (see line 19 in Algorithm 1). The minimum selection probability of $p_{min} = 0.1$ is used for operator selection, and adaptation rate is fixed as $\alpha = 0.9$, as suggested in Durgut and Aydin (2021). Unless mentioned otherwise, the number of threads is set to 512 in an experiment. The following performance measures are used based on the results from the trials: best, worst and mean objective values along with the standard deviation, *gap* and *hit*, where hit indicates the number of times that the perfect solution (optimal) was achieved out of all trials, and gap is the ratio of mean to the distance between the mean and optimum in percentage. Time is given in seconds and the computational results of ABPEA on benchmark instances are presented in Tables A.1 and A.2 in Appendix.

4.1. Tuning ε−greedy policy

In this set of experiments, we investigated the settings of three different algorithmic components for the reinforcement learning scheme: greedy policy threshold (ε), reward strategy (*RWS*), and window size (W). Tables 1–3 presents the tuning results for two challenging instances in M* which are MQ5_300_300 and MR2_500_500 for different values of ε. For each ε value, different reward strategies and W values are analysed, and the number of threads is chosen as 512 for the proposed method. We tune three different values for ε which are 0.05, 0.1 and 0.2 and the results are shown in Tables 1–3, respectively. We use a ranking scheme to clearly understand the effectiveness of the parameters that average rank and overall rank are denoted as A.R. and O.R.

It is observed in Tables 1–3 that the worst results for MQ5_300_300 instance are obtained with *instant RWS* in all ε values. Similarly, the results are not satisfactory with *instant RWS* on MR2_500_500 instances. Therefore, *average* and *extreme RWS* using sliding window approach works better compared to *instant RWS*.

We have investigated four different window size values set of {5, 10, 20, 50} for *average* and *extreme RWS*. When *average* and *extreme RWS* are considered in Table 1 (ε = 0.05), *extreme RWS* ranks worse than *average RWS* for both MQ5_300_300 and MR2_500_500 instances.

[1] The code of our approach is available at: https://github.com/3mrullah/ABPEA.

Table 1
Parametric exploration for W and RWS under $\varepsilon = 0.05$.

Parameters		MQ5_300_300			MR2_500_500			Total rank	
W	*RWS*	Mean	Std.Dev.	Rank	Mean	Std.Dev.	Rank	A.R.	O.R.
1	Instant	4088.48	9.640	7	2656.87	8.140	3	5.0	4
5	Average	4084.62	3.530	1	2656.17	7.830	2	1.5	1
5	Extreme	4087.94	9.310	6	2657.24	9.620	4	5.0	4
10	Average	4087.55	8.780	5	2655.80	5.860	1	3.0	2
10	Extreme	4090.10	9.690	9	2663.15	20.700	8	8.5	6
20	Average	4087.07	7.980	2	2655.80	5.860	1	1.5	1
20	Extreme	4087.39	8.230	4	2657.60	10.890	5	4.5	3
50	Average	4087.26	7.700	3	2658.30	11.020	6	4.5	3
50	Extreme	4088.73	9.680	8	2659.85	13.470	7	7.5	5

Table 2
Parametric exploration for W and RWS under $\varepsilon = 0.1$.

Parameters		MQ5_300_300			MR2_500_500			Total rank	
W	*RWS*	Mean	Std.Dev.	Rank	Mean	Std.Dev.	Rank	A.R.	O.R.
1	Instant	4088.70	9.010	6	2657.94	9.790	4	5.0	5
5	Average	4084.84	6.052	1	2654.74	0.000	1	1.0	1
5	Extreme	4087.20	8.950	3	2660.57	15.140	6	4.5	4
10	Average	4089.10	8.870	8	2655.80	5.860	2	5.0	5
10	Extreme	4087.55	8.780	4	2659.46	15.150	5	4.5	4
20	Average	4087.20	8.950	3	2657.24	9.620	3	3.0	2
20	Extreme	4085.77	7.600	2	2657.94	9.790	4	3.0	2
50	Average	4088.21	9.400	5	2655.80	5.860	2	3.5	3
50	Extreme	4088.85	9.280	7	2657.24	9.620	3	5.0	5

Table 3
Parametric exploration for W and RWS under $\varepsilon = 0.2$.

Parameters		MQ5_300_300			MR2_500_500			Total rank	
W	*RWS*	Mean	Std.Dev.	Rank	Mean	Std.Dev.	Rank	A.R.	O.R.
1	Instant	4089.10	9.300	7	2656.15	7.730	3	5.0	4
5	Average	4085.43	6.970	1	2654.74	0.000	1	1.0	1
5	Extreme	4088.07	9.120	5	2659.74	13.120	6	5.5	5
10	Average	4090.26	9.930	9	2654.74	0.000	1	5.0	4
10	Extreme	4086.23	8.460	2	2656.87	8.140	5	3.5	3
20	Average	4088.53	10.130	6	2656.87	8.140	5	5.5	5
20	Extreme	4087.93	8.650	4	2661.78	16.630	7	5.5	5
50	Average	4087.61	9.520	3	2655.80	5.860	2	2.5	2
50	Extreme	4089.83	10.060	8	2656.17	7.830	4	6.0	6

The top three scores are achieved by *average RWS*, and the best result is obtained with W = 5 for MQ5_300_300 instance and W = 10 and W = 20 for MR2_500_500 instance. In Table 2 (ε = 0.1), although *extreme RWS* is at the second of the list, *average RWS* with W = 5 is in the first place for both MQ5_300_300 and MR2_500_500 instances. Similarly, *extreme RWS* does not produce satisfactory results under ε = 0.2 as seen in Table 3. In conclusion, *average RWS* is clearly better than *extreme RWS* for various W values.

The best result for MQ5_300_300 instance is obtained with *average RWS* using settings ε = 0.05, W = 5 as seen in Table 1. Also, all first ranks utilise *average RWS* where W = 5 for all ε settings so it can be proved that *average RWS* is the most efficient *RWS* for our method. In addition, the best optimal is found in Tables 2 (ε = 0.1) and 3 (ε = 0.2) for MR2_500_500 instance with *average RWS* and W = 5. From this analysis, we can conclude that *average RWS* is the most efficient approach as a reward strategy since it improves the OFV with a few last iteration knowledge (W = 5). Furthermore, it can be inferred that using *extreme RWS* is not an effective approach according to the standard deviations. As a result of all our initial parameter tuning experiments, we have fixed *average RWS* for our method with the configuration of parameters ε = 0.1, W = 5.

4.2. Effect of varying the number of threads

In this set of experiments, we have investigated the effect of increasing the number of threads on the performance of ABPEA using the

Table 4
Gaps and hits for ORLib instances by using different numbers of threads on ABPEA.

Instance	32 Threads		64 Threads		128 Threads		256 Threads		512 Threads	
	Gap	Hit	Gap	Hit	Gap	Hit	Gap	Hit	Gap	Hit
cap71_16_50	0.000	30	0.000	30	0.000	30	0.000	30	0.000	30
cap72_16_50	0.000	30	0.000	30	0.000	30	0.000	30	0.000	30
cap73_16_50	0.000	30	0.000	30	0.000	30	0.000	30	0.000	30
cap74_16_50	0.000	30	0.000	30	0.000	30	0.000	30	0.000	30
cap101_25_50	0.000	30	0.000	30	0.000	30	0.000	30	0.000	30
cap102_25_50	0.000	30	0.000	30	0.000	30	0.000	30	0.000	30
cap103_25_50	0.000	30	0.000	30	0.000	30	0.000	30	0.000	30
cap104_25_50	0.000	30	0.000	30	0.000	30	0.000	30	0.000	30
cap131_50_50	0.000	30	0.000	30	0.000	30	0.000	30	0.000	30
cap132_50_50	0.001	29	0.000	30	0.000	30	0.000	30	0.000	30
cap133_50_50	0.007	25	0.000	30	0.000	30	0.000	30	0.000	30
cap134_50_50	0.000	30	0.000	30	0.000	30	0.000	30	0.000	30
capa_100_1000	0.313	20	0.000	30	0.000	30	0.000	30	0.000	30
capb_100_1000	0.591	4	0.156	17	0.111	22	0.044	25	0.026	28
capc_100_1000	0.686	0	0.184	2	0.023	14	0.018	25	0.000	30

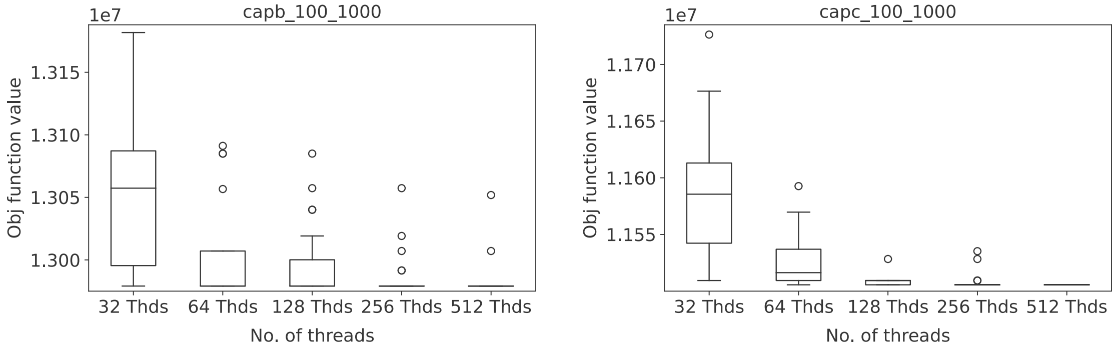

Fig. 1. Boxplots of objective values obtained in 30 runs for a different number of threads on capb_100_1000 and capc_100_1000 instances.

ORLib instances. Increasing the number of threads implies increasing the diversity/spread of the initial population in the landscape, as each thread starts the search using a different solution. The *gap* and *hit* values for 32, 64, 128, 256 and 512 threads are shown in Table 4. According to the results, the best solutions were obtained for the instances with relatively smaller size ($n = 50$, $m = 50$) and increasing the number of threads does not have much influence, except for cap132_50_50 and cap133_50_50. As for the other instances, it can be seen that ABPEA can escape from the local optima when the number of threads increases. ABPEA with fewer threads could miss the global optima even if the size of the problem is medium. Table 4 also reports that ABPEA with 256 and 512 threads obtained the best solution more than 20 times for the most challenging instance group (capx_100_1000) in ORLib. It can be concluded that the proposed approach may stuck local optima when it uses fewer threads. The best method using 512 threads hits the best for all instances in all trials except for capb_100_1000. It misses to detect the optimal solution only 2 out of 30 trials for that instance.

Fig. 1 shows the box plots of capb_100_1000 and capc_100_1000 instances under the different numbers of threads. The box plot has less height when the number of threads is equal to 256 and 512 as compared to the others. Even if ABPEA with 128 threads has less height for the capc_100_1000 instance, it is not as good enough for capb_100_1000 instance. It can be inferred that ABPEA has become more robust, efficient and effective with 512 threads since it obtained satisfactory results as compared to the others.

Fig. 2 compares the execution time in seconds according to the number of threads. We use average times for four different instance groups according to their sizes. Average execution times for 32, 64 and 128 threads are very close for all groups. It is observed that the execution times slightly increase when the number of threads increases to 256 and 512. There is a 0.5 s difference between 32 threads and 512 threads for all groups. The main reason for this difference is latency caused by waiting for all threads to complete their operations

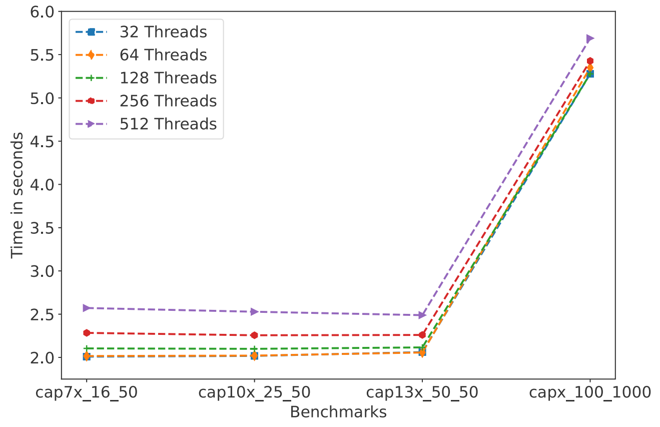

Fig. 2. Average time cost in seconds among 30 runs for each group in ORLib instances.

in order to continue for the next iteration. In other words, threads must synchronise with *#omp barrier* before assigning credits and calculating probabilities in AOS strategy (see line 14 in Algorithm 1). Besides, each thread has its own memory space. Thus, the method needs larger memory space, and this also increases the run time due to latency caused by more memory accesses.

4.3. Performance comparison of ABPEA to the other methods

In this set of experiments, we discuss the performance of ABPEA with a comparison of state-of-the-art methods on instances in both ORLib and M*. Also, we analyse the execution time of ABPEA in terms of speedup against memOp-based sequential EA. As mentioned earlier,

Table 5
A comparison with state-of-the-art methods on ORLib instances.

Instance	EGTOA		JayaX		BinCSA		T-NBDE		BABC-AP		ABPEA	
	Gap	Hit	Gap	Hit	Gap	Hit	Gap	Hit	Gap	Hit	Gap	Hit
cap71_16_50	0.000	30	0.000	30	0.000	30	0.000	30	0.000	30	0.000	30
cap72_16_50	0.000	30	0.000	30	0.000	30	0.000	30	0.000	30	0.000	30
cap73_16_50	0.000	30	0.000	30	0.000	30	0.000	30	0.000	30	0.000	30
cap74_16_50	0.000	30	0.000	30	0.000	30	0.000	30	0.000	30	0.000	30
cap101_25_50	0.000	30	0.000	30	0.000	30	0.000	30	0.000	30	0.000	30
cap102_25_50	0.000	30	0.000	30	0.000	30	0.000	30	0.000	30	0.000	30
cap103_25_50	0.000	30	0.000	30	0.000	30	0.000	30	0.000	30	0.000	30
cap104_25_50	0.000	30	0.000	30	0.000	30	0.000	30	0.000	30	0.000	30
cap131_50_50	0.000	30	0.000	30	0.000	30	0.000	30	0.000	30	0.000	30
cap132_50_50	0.000	30	0.000	30	0.000	30	0.000	30	0.000	30	0.000	30
cap133_50_50	0.000	30	0.000	30	0.000	30	0.000	30	0.000	30	0.000	30
cap134_50_50	0.000	30	0.000	30	0.000	30	0.000	30	0.000	30	0.000	30
capa_100_1000	0.000	30	0.000	30	0.000	30	0.000	30	0.000	30	0.000	30
capb_100_1000	0.121	24	0.079	26	0.006	28	0.000	30	0.000	30	0.026	28
capc_100_1000	0.091	9	0.022	17	0.021	11	0.014	17	0.004	26	0.000	30

each algorithm uses the same maximum number of evaluations for termination to fairly compare their performances to ABPEA.

Gap and hit values on ORLib instances are tabulated in Table 5 to compare ABPEA with state-of-the-art methods. Nevertheless, we ignored poor methods and chose five binary swarm intelligence algorithms that produced competitive results against the other methods in the literature. The results of the state-of-the-art methods were taken directly from the relevant studies. EGTOA (Zhang et al., 2023) is an enhanced group-theory optimisation algorithm based on two strategies which are one direction mutation operator (ODMP) and redundant checking strategy (RCS). ODMP decides whether to close an open facility by comparing a random value in (0,1) with the mutation probability. RCS checks whether an open facility serves any customer or not. If so, then the facility will be closed to reduce the total cost caused by the opening cost. JayaX (Aslan et al., 2019) is an EA based on XOR logic operator, and it uses a local search module to get better intensification in the search space. First, the XOR operator generates a temporary solution using the best and worst solutions. Then XOR operation performs to the temporary solution and current solution. After these two steps, a new candidate solution is generated. Later, the local search module searches for a better solution using the existing one with a greedy approach. BinCSA (Sonuç, 2021) is another XOR-based EA and it uses a single operator, which is included as memOp in the operator pool of ABPEA. While memOP intensifies to a region, CSA has a control parameter that diversifies the search space by generating a new random solution.

T-NBDE (He et al., 2022) is a binary variant of the differential evolution (DE) algorithm. Since DE handles continuous optimisation problems, it needs to map solutions from continuous to the binary domain. For this reason, transfer functions called tapper-shaped were used in T-NBDE to deal with UFLP. In each iteration, DE performs the mutation and crossover operations for each solution (defined as a real vector). Then a transfer function maps the real vector into the binary domain. After, T-NBDE decides which individuals are selected for the next generation. ABABC-AP (Durgut & Aydin, 2021) is an adaptive population-based metaheuristic and it has three perturbative operators. These operators taken from different studies and selection process is performed by different adaptation schemes which are Probability Matching and Adaptive Pursuit. Each operator generates a candidate solution to intensify in a local region. If an individual in a population cannot improve its solution throughout predetermined iterations, then ABABC-AP uses a mechanism that generates a new random solution for escaping from local optima.

Based on the results in Table 5, all methods hit the optimal solution on small-size (cap7x_16_50), medium-size (cap10x_25_50 and cap13x_25_50) and large-size (capx_100_1000) instances. However, the main difference between the methods is on two hard instances which are capb_100_1000 and capc_10_1000. T-NBDE and ABABC-AP obtained

Table 6
Average rankings of algorithms on ORLib instances.

Order	Algorithm	Rank score
1	BABC-AP	3.27
2	ABPEA	3.33
3	T-NBDE	3.37
4	BinCSA	3.60
5	Jayax	3.60
6	EGTOA	3.83

the optimal solution on capb_100_1000 instances for all runs. BinCSA and ABPEA have 28 hits out of 30; however, BinCSA is slightly better than ABPEA in terms of the gap score. That means BinCSA has more promising results than ABPEA on this instance. JayaX is slightly worse than ABPEA, but it can be said that it is an efficient method since it solves the instance optimally 26 times out of 30. ISS is the worst method since it can obtain optimal solution of capb_100_1000 by a probability of 60%.

Considering the results on capc_100_1000 instance, same as on capb_100_1000, ISS is the worst method with 5 hits out of 30. BinCSA cannot obtain satisfactory results on this instance as on capb_100_1000. It reaches the optimal solution 11 times out of 30 which means it can obtain optimal solution of capc_100_1000 by a probability of 37%. JayaX and T-NBDE solve capc_100_1000 optimally 17 times out of 30; however, JayaX performs slightly worse than T-NBDE in terms of the gap score. ABABC-AP and ABPEA can obtain optimal solution 26 and 30 times out of 30, respectively. Even if ABPEA misses the optima 2 times out of 30 on capb_100_1000, it solves capc_100_1000 optimally for all runs. It is clear that ABPEA outperforms all other methods on capc_100_1000 instance. Table 6 presents average rankings calculated by Friedman's test using average hit scores. BABC-AP is in the first place and ABPEA ranks second with a slighty difference. These scores prove that ABABC-AP and ABPEA are highly competitive and efficient methods for solving ORLib instances.

Table 7 reports mean and gap scores on M* instances for state-of-the-art methods to demonstrate the performance of ABPEA. The best gap scores for each instance are shown in bold. The results in the table were taken directly from the relevant studies. PLS (Cura, 2010) is a parallel local search that develops for a multi-thread system. It also uses the multi-search mechanism to get better diversification in the search space. BinSSA (Baş & Ülker, 2020) is a binary EA and generates a new solution using two perturbative operators based on similarity measure and XOR. BinGSO (Kaya, 2022) is a hybrid population-based metaheuristic and developed on two steps. In the first step, BinGSO generates new solutions for each subpopulation using a perturbative operator to diversify the search space. Later, it combines the solutions for intensifying at promising solution regions.

Table 7
A comparison with state-of-the-art methods on M* instances.

Instance	PLS		BinCSA		BinSSA		BinGSO		ABPEA	
	Mean	Gap	Mean	Gap	Mean	Gap	Mean	Gap	Mean	Gap
MO1_100_100	1305.95	**0.000**	1305.95	**0.000**	1305.95	**0.000**	1305.95	**0.000**	1305.95	**0.000**
MO2_100_100	1432.49	0.009	1432.36	**0.000**	1432.36	**0.000**	1432.36	**0.000**	1432.36	**0.000**
MO3_100_100	1520.27	0.230	1516.77	**0.000**	1516.77	**0.000**	1516.77	**0.000**	1516.77	**0.000**
MO4_100_100	1442.24	**0.000**	1442.24	**0.000**	1442.24	**0.000**	1442.24	**0.000**	1442.24	**0.000**
MO5_100_100	1409.17	0.028	1408.77	**0.000**	1408.77	**0.000**	1408.77	**0.000**	1408.77	**0.000**
MP1_200_200	2688.50	0.075	2686.66	0.007	2687.66	0.044	2687.40	0.034	2686.48	**0.000**
MP2_200_200	2904.86	**0.000**	2904.86	**0.000**	2904.86	**0.000**	2904.86	**0.000**	2904.86	**0.000**
MP3_200_200	2624.77	0.040	2623.71	**0.000**	2624.00	0.011	2623.71	**0.000**	2623.71	**0.000**
MP4_200_200	2939.53	0.026	2938.83	0.003	2940.50	0.060	2939.44	0.013	2938.75	**0.000**
MP5_200_200	2933.46	0.038	2932.33	**0.000**	2932.50	0.006	2932.33	**0.000**	2932.33	**0.000**
MQ1_300_300	4091.01	**0.000**	4091.01	**0.000**	4091.01	**0.000**	4091.01	**0.000**	4091.01	**0.000**
MQ2_300_300	4028.33	**0.000**	4028.33	**0.000**	4028.33	**0.000**	4028.33	**0.000**	4028.33	**0.000**
MQ3_300_300	4275.43	**0.000**	4275.43	**0.000**	4275.43	**0.000**	4275.43	**0.000**	4275.43	**0.000**
MQ4_300_300	4235.47	0.008	4235.15	**0.000**	4236.46	0.031	4235.42	0.006	4235.15	**0.000**
MQ5_300_300	4086.53	0.142	4087.95	0.177	4086.45	0.140	4086.62	0.144	4086.11	**0.132**
MR1_500_500	2608.24	0.004	2619.04	0.418	2610.24	0.080	2609.30	0.044	2608.21	**0.002**
MR2_500_500	2654.73	**0.000**	2718.65	2.408	2655.73	0.038	2661.15	0.242	2655.81	0.041
MR3_500_500	2789.04	0.028	2791.34	0.111	2790.14	0.068	2793.47	0.187	2788.60	**0.013**
MR4_500_500	2756.04	**0.000**	2785.53	1.070	2756.04	**0.000**	2768.34	0.446	2756.32	0.010
MR5_500_500	2505.48	0.017	2510.50	0.218	2505.40	0.014	2510.25	0.208	2505.31	**0.010**
Avg. Gaps		*0.032*		*0.221*		*0.025*		*0.066*		*0.010*

When the results in Table 7 are examined, all methods except PLS can obtain optimal solutions on MOx_100_100 group instances. PLS can solve only MO1_100_100 and MO4_100_100 instances optimally. The gap score on MO3_100_100 is 0.230 which is unexpectedly the worst score among all instances for PLS. For the second group of instances in M*, named MPx_200_200, PLS is the worst method as on MOx_100_100 instances. Also BinSSA cannot achieve promising results like PLS, since both can solve only MP2_200_200 optimally. It can be observed that MP2_200_200 is the easiest instance to solve for the second group because all methods can obtain the best solution for all runs. BinCSA and BinGSO can achieve the optimal solution for 3 out of 5 instances; however, BinCSA is slightly better than BinGSO in terms of gap scores. As in MOx_100_100 instances, ABPEA can reach the optimal solutions in all runs for MPx_200_200 instances. For the first three instances in the third group (MQx_300_300), all methods can obtain optimal solutions. Although only BinCSA and ABPEA can solve MQ4_300_300 instance optimally, PLS and BinGSO have promising gap scores. It can be said that MQ5_300_300 instance is one of the hardest instances in this benchmark since no method can solve it optimally, moreover, all of the gap scores are greater than 0.1. For MRx_500_500 BinCSA and BinGSO cannot obtain optimal solutions for any instances. BinGSO is slightly better than BinCSA since it has better gap scores except for MR3_500_500. BinSSA and PLS can obtain optimal solution on MR4_500_500 instance. Furthermore, only PLS can solve MR2_500_500 optimally. Despite this, ABPEA outperforms all methods for the other three instances in the group in terms of gap score.

According to the average gaps in Table 7, BinCSA is the worst method among others despite it can obtain optimal solutions for 12 out of 20 instances in M*. It can be said BinCSA stucks in local optima on some MRx_500_500 instances since its' gap scores are greater than 1.0. BinGSO has a similar performance as BinCSA since it cannot obtain promising results on MRx_500_500 instances. Although PLS has satisfactory performance on MRx_500_500 instances, its performance in MOx_100_100 can be expressed as proof that it leads to poor intensification. Overall, PLS, BinSSA and ABPEA can obtain promising results for solving M* instances that their average gap scores are 0.032, 0.025 and 0.010, respectively. Consequently, average gap scores confirm that ABPEA is the most effective method against state-of-the-art algorithms.

Table 8 compares the proposed approach (ABPEA) against random selection (non-adaptive), labelled as Random BPEA. As in the previous experiments, each metric is calculated over 30 and 100 repetitions for ORLib and M*, respectively. Furthermore, the performance of ABPEA is analysed using the Wilcoxon signed-rank test, a commonly used non-parametric statistical test. The following notation is adopted. For a given instance, > (<) denotes that ABPEA performs better (worse) than Random BPEA and this performance difference is statistically significant within a confidence interval of 95%. ⩾ denotes that ABPEA performs slightly better than Random BPEA but no statistical significance, while ≈ denotes that both methods perform similarly on average. Even though Non-Adaptive BPEA can easily solve most of the benchmark problems on ORLib, it is not competitive for most challenging problems labelled as capb_100_1000 and capc_100_1000. It can be seen from the table that the performance of ABPEA significantly differs from Random BPEA for these instances according to the statistical tests. Likewise, Random BPEA cannot achieve promising results on hardest problem instances (e.g. MQ5_300_300, MR2_500_500.) on M*. 5 of 20 instances on M* are statistically significant considering the performance differences between ABPEA and Random BPEA.

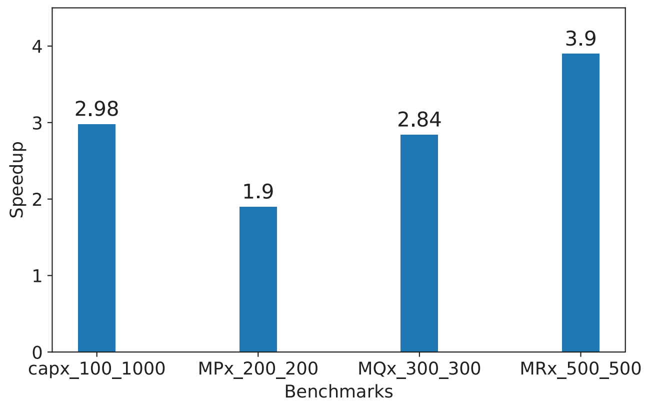

Fig. 3. Comparison between BinCSA and ABPEA in terms of average speedup for benchmark groups.

Fig. 3 shows average speedups for benchmark groups in ORLib and M*. We examined only hard instances since no acceleration was gained for other group instances and BinCSA also solved them optimally. The speedup is up to 3.9x for the hardest instances (MRx_500_500) in this study. Thus, this acceleration is capable of making ABPEA more efficient against BinCSA.

Table 8
A comparison of the proposed approach to the random selection scheme.

Instance	ABPEA			vs.	Random BPEA		
	Mean	Std.Dev.	Gap		Mean	Std.Dev.	Gap
cap71_16_50	932,615.75	0.0000	0.0000	≈	932,615.75	0.0000	0.0000
cap72_16_50	977,799.40	0.0000	0.0000	≈	977,799.40	0.0000	0.0000
cap73_16_50	1,010,641.45	0.0000	0.0000	≈	1,010,641.45	0.0000	0.0000
cap74_16_50	1,034,976.98	0.0000	0.0000	≈	1,034,976.98	0.0000	0.0000
cap101_25_50	796,648.44	0.0000	0.0000	≈	796,648.44	0.0000	0.0000
cap102_25_50	854,704.20	0.0000	0.0000	≈	854,704.20	0.0000	0.0000
cap103_25_50	893,782.11	0.0000	0.0000	≈	893,782.11	0.0000	0.0000
cap104_25_50	928,941.75	0.0000	0.0000	≈	928,941.75	0.0000	0.0000
cap131_50_50	793,439.56	0.0000	0.0000	≈	793,439.56	0.0000	0.0000
cap132_50_50	851,495.33	0.0000	0.0000	≈	851,495.33	0.0000	0.0000
cap133_50_50	893,076.71	0.0000	0.0000	⩾	893,122.10	172.8528	0.0051
cap134_50_50	928,941.75	0.0000	0.0000	≈	928,941.75	0.0000	0.0000
capa_100_1000	17,156,454.48	0.0000	0.0000	⩾	17,160,340.27	21,283.3860	0.0226
capb_100_1000	12,982,431.85	14,074.2596	0.0259	>	13,045,440.32	59,298.7680	0.5114
capc_100_1000	11,505,594.33	0.0000	0.0000	>	11,604,035.61	120,521.2587	0.8556
MO1_100_100	1305.95	0.0000	0.0000	≈	1305.95	0.0000	0.0000
MO2_100_100	1432.36	0.0000	0.0000	≈	1432.36	0.0000	0.0000
MO3_100_100	1516.77	0.0000	0.0000	≈	1516.77	0.0000	0.0000
MO4_100_100	1442.24	0.0000	0.0000	≈	1442.24	0.0000	0.0000
MO5_100_100	1408.77	0.0000	0.0000	≈	1408.77	0.0000	0.0000
MP1_200_200	2686.48	0.0000	0.0000	≈	2686.48	0.0000	0.0000
MP2_200_200	2904.86	0.0000	0.0000	≈	2904.86	0.0000	0.0000
MP3_200_200	2623.71	0.0000	0.0000	≈	2623.71	0.0000	0.0000
MP4_200_200	2938.75	0.0000	0.0000	⩾	2938.83	0.5457	0.0026
MP5_200_200	2932.33	0.0000	0.0000	⩾	2932.38	0.4612	0.0016
MQ1_300_300	4091.01	0.0000	0.0000	≈	4091.01	0.0000	0.0000
MQ2_300_300	4028.33	0.0000	0.0000	≈	4028.33	0.0000	0.0000
MQ3_300_300	4275.43	0.0000	0.0000	≈	4275.43	0.0000	0.0000
MQ4_300_300	4235.15	0.0000	0.0000	≈	4235.15	0.0000	0.0000
MQ5_300_300	4086.11	7.4803	0.1316	>	4097.17	9.0717	0.4027
MR1_500_500	2608.21	0.2449	0.0025	>	2609.29	2.2333	0.0438
MR2_500_500	2655.81	6.1846	0.0405	>	2663.65	17.2981	0.3358
MR3_500_500	2788.60	1.2934	0.0126	>	2789.84	2.4506	0.0570
MR4_500_500	2756.32	2.0663	0.0103	>	2758.10	5.8027	0.0748
MR5_500_500	2505.31	1.5783	0.0104	⩾	2505.96	4.7954	0.0365

5. Conclusion

In this study, we introduce an adaptive binary parallel evolutionary algorithm, referred to as ABPEA implemented using OpenMP for solving UFLP. ABPEA exploits the strengths of multiple operators (memOp, disOp, eltOp) in a multi-threaded environment. Moreover, a reinforcement learning approach is used to maintain a credit score for each operator based on their performance during the search process. We have developed three different credit rewarding strategies, including *instant*, *average*, and *extreme* with the latter two having a certain window size. Each thread selects and executes an operator at a decision point based on the credit scores of those operators using Greedy policy, located in a shared memory. To ensure that different parts of the search landscape are explored, each thread is seeded with a different starting solution.

We have conducted a significant amount of experiments on the 35 UFLP instances in the benchmarks of ORLib and M* to investigate the performance of our approach. First, we have investigated the influence of using different algorithmic components/choices and number of threads. The algorithm configuration and parameter tuning experiments reveal that the adaptive *average* rewarding strategy with a window size of 5 is the best component using 512 threads. ABPEA speeds up the sequential algorithm using the single operator of memOp, up to 3.9x. ABPEA turns out to be an effective and efficient approach for solving UFLP, considering that it is indeed capable of obtaining the optimal solutions for all benchmark instances in ORLib and M*, rapidly. The experimental results also confirm that ABPEA either outperforms or is competitive with the recently proposed state-of-the-art methods for UFLP based on the gap scores.

It has been observed that different operator selection and credit assignment strategies could yield different performances for the overall approach (Drake et al., 2020). Hence, a trivial future work would be to study different AOS mechanisms, such as, Modified Choice Function (Drake, Özcan, & Burke, 2015) or Upper Confidence Bound (Fialho, 2010) in combination with different credit assignment schemes. Additionally, ABPEA framework can be extended, or its GPU implementation could be developed for solving the instances of other computationally hard real-world combinatorial optimisation problems.

CRediT authorship contribution statement

Emrullah Sonuç: Conceptualization, Methodology, Software, Visualization, Writing – original draft, Writing – review & editing. **Ender Özcan:** Validation, Supervision, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

This work was supported by the Scientific and Technological Research Council of Türkiye (TÜBİTAK) under the BIDEB-2219 International Postdoctoral Research Fellowship Programme grant number 1059B192001306.

Table A.1
Optimal, best, worst, and mean objective values along with their standard deviation, average gap, hit rate, and average execution time (in seconds) of ABPEA based on 30 trials for the ORLib instances.

Instance	Optimal	Best	Worst	Mean	Std.Dev.	Gap	Hit	Time (s)
cap71_16_50	932,615.75	932,615.75	932,615.75	932,615.75	0.000	0.000	30/30	2.57
cap72_16_50	977,799.40	977,799.40	977,799.40	977,799.40	0.000	0.000	30/30	2.56
cap73_16_50	1,010,641.45	1,010,641.45	1,010,641.45	1,010,641.45	0.000	0.000	30/30	2.57
cap74_16_50	1,034,976.98	1,034,976.98	1,034,976.98	1,034,976.98	0.000	0.000	30/30	2.58
cap101_25_50	796,648.44	796,648.44	796,648.44	796,648.44	0.000	0.000	30/30	2.50
cap102_25_50	854,704.20	854,704.20	854,704.20	854,704.20	0.000	0.000	30/30	2.53
cap103_25_50	893,782.11	893,782.11	893,782.11	893,782.11	0.000	0.000	30/30	2.52
cap104_25_50	928,941.75	928,941.75	928,941.75	928,941.75	0.000	0.000	30/30	2.57
cap131_50_50	793,439.56	793,439.56	793,439.56	793,439.56	0.000	0.000	30/30	2.47
cap132_50_50	851,495.33	851,495.33	851,495.33	851,495.33	0.000	0.000	30/30	2.49
cap133_50_50	893,076.71	893,076.71	893,076.71	893,076.71	0.000	0.000	30/30	2.48
cap134_50_50	928,941.75	928,941.75	928,941.75	928,941.75	0.000	0.000	30/30	2.52
capa_100_1000	17,156,454.48	17,156,454.48	17,156,454.48	17,156,454.48	0.000	0.000	30/30	5.61
capb_100_1000	12,979,071.58	12,979,071.58	13,051,859.33	12,982,431.85	14,074.260	0.026	28/30	5.69
capc_100_1000	11,505,594.33	11,505,594.33	11,505,594.33	11,505,594.33	0.000	0.000	30/30	5.77

Table A.2
Optimal, best, worst, and mean objective values along with their standard deviation, average gap, hit rate, and average execution time (in seconds) of ABPEA based on 100 trials for M* instances.

Instance	Optimal	Best	Worst	Mean	Std.Dev.	Gap	Hit	Time (s)
MO1_100_100	1305.95	1305.95	1305.95	1305.95	0.000	0.0000	100/100	3.10
MO2_100_100	1432.36	1432.36	1432.36	1432.36	0.000	0.0000	100/100	3.10
MO3_100_100	1516.77	1516.77	1516.77	1516.77	0.000	0.0000	100/100	3.09
MO4_100_100	1442.24	1442.24	1442.24	1442.24	0.000	0.0000	100/100	3.12
MO5_100_100	1408.77	1408.77	1408.77	1408.77	0.000	0.0000	100/100	3.10
MP1_200_200	2686.48	2686.48	2686.48	2686.48	0.000	0.0000	100/100	3.85
MP2_200_200	2904.86	2904.86	2904.86	2904.86	0.000	0.0000	100/100	3.87
MP3_200_200	2623.71	2623.71	2623.71	2623.71	0.000	0.0000	100/100	3.89
MP4_200_200	2938.75	2938.75	2938.75	2938.75	0.000	0.0000	100/100	3.88
MP5_200_200	2932.33	2932.33	2932.33	2932.33	0.000	0.0000	100/100	3.87
MQ1_300_300	4091.01	4091.01	4091.01	4091.01	0.000	0.0000	100/100	5.56
MQ2_300_300	4028.33	4028.33	4028.33	4028.33	0.000	0.0000	100/100	5.57
MQ3_300_300	4275.43	4275.43	4275.43	4275.43	0.000	0.0000	100/100	5.54
MQ4_300_300	4235.15	4235.15	4235.15	4235.15	0.000	0.0000	100/100	5.57
MQ5_300_300	4080.74	4080.74	4103.755	4086.110	7.480	0.1316	49/100	5.48
MR1_500_500	2608.15	2608.15	2609.133	2608.215	0.245	0.0025	93/100	10.97
MR2_500_500	2654.73	2654.73	2697.648	2655.806	6.185	0.0405	97/100	10.95
MR3_500_500	2788.25	2788.25	2794.408	2788.601	1.293	0.0126	93/100	10.96
MR4_500_500	2756.04	2756.04	2773.889	2756.323	2.066	0.0103	98/100	10.94
MR5_500_500	2505.05	2505.05	2516.794	2505.311	1.578	0.0104	97/100	11.01

Appendix. Experimental results of ABPEA for all benchmark instances

See Tables A.1 and A.2.

References

Akan, T., Agahian, S., & Dehkharghani, R. (2022). Binbro: Binary battle royale optimizer algorithm. *Expert Systems with Applications*, *195*, Article 116599.

Akinc, U., & Khumawala, B. M. (1977). An efficient branch and bound algorithm for the capacitated warehouse location problem. *Management Science*, *23*, 585–594.

Alba, E., Luque, G., & Nesmachnow, S. (2013). Parallel metaheuristics: recent advances and new trends. *International Transactions in Operational Research*, *20*, 1–48.

Aslan, M., Gunduz, M., & Kiran, M. S. (2019). Jayax: Jaya algorithm with xor operator for binary optimization. *Applied Soft Computing*, *82*, Article 105576.

Balinski, M. L. (1964). *On finding integer solutions to linear programs: Technical report mathematica Princeton NJ*.

Banos, R., Ortega, J., Gil, C., de Toro, F., & Montoya, M. G. (2016). Analysis of openmp and mpi implementations of meta-heuristics for vehicle routing problems. *Applied Soft Computing*, *43*, 262–275.

Baş, E., & Ülker, E. (2020). A binary social spider algorithm for uncapacitated facility location problem. *Expert Systems with Applications*, *161*, Article 113618.

Beasley, J. E. (1990). Or-library: distributing test problems by electronic mail. *Journal of the Operational Research Society*, *41*, 1069–1072.

Choi, S.-S., Cha, S.-H., & Tappert, C. C. (2010). A survey of binary similarity and distance measures. *Journal of Systemics, Cybernetics and Informatics*, *8*, 43–48.

Cinar, A. C., & Kiran, M. S. (2018). Similarity and logic gate-based tree-seed algorithms for binary optimization. *Computers & Industrial Engineering*, *115*, 631–646.

Cornuéjols, G., Nemhauser, G., & Wolsey, L. (1983). *The uncapicitated facility location problem: Technical report*, Cornell University Operations Research and Industrial Engineering.

Cowling, P., Kendall, G., & Soubeiga, E. (2000). A hyperheuristic approach to scheduling a sales summit. In *International conference on the practice and theory of automated timetabling* (pp. 176–190). Springer.

Crainic, T. (2019). Parallel metaheuristics and cooperative search. In *Handbook of metaheuristics* (pp. 419–451). Springer.

Cura, T. (2010). A parallel local search approach to solving the uncapacitated warehouse location problem. *Computers & Industrial Engineering*, *59*, 1000–1009.

Di Gaspero, L., & Urli, T. (2012). Evaluation of a family of reinforcement learning cross-domain optimization heuristics. In *International conference on learning and intelligent optimization* (pp. 384–389). Springer.

Drake, J. H., Kheiri, A., Özcan, E., & Burke, E. K. (2020). Recent advances in selection hyper-heuristics. *European Journal of Operational Research*, *285*, 405–428.

Drake, J. H., Özcan, E., & Burke, E. K. (2015). A modified choice function hyper-heuristic controlling unary and binary operators. In *2015 IEEE congress on evolutionary computation* (pp. 3389–3396). IEEE.

Durgut, R. (2021). Improved binary artificial bee colony algorithm. *Frontiers of Information Technology & Electronic Engineering*, *22*, 1080–1091.

Durgut, R., & Aydin, M. E. (2021). Adaptive binary artificial bee colony algorithm. *Applied Soft Computing*, *101*, Article 107054.

Efroymson, M., & Ray, T. (1966). A branch-bound algorithm for plant location. *Operations Research*, *14*, 361–368.

Erlenkotter, D. (1978). A dual-based procedure for uncapacitated facility location. *Operations Research*, *26*, 992–1009.

Fialho, Á. (2010). *Adaptive operator selection for optimization* (Ph.D. thesis), Université Paris Sud-Paris XI.

Galvão, R. D., & Raggi, L. A. (1989). A method for solving to optimality uncapacitated location problems. *Annals of Operations Research*, *18*, 225–244.

García, J., Crawford, B., Soto, R., & Astorga, G. (2019). A clustering algorithm applied to the binarization of swarm intelligence continuous metaheuristics. *Swarm and Evolutionary Computation*, *44*, 646–664.

Glover, F., Hanafi, S., Guemri, O., & Crevits, I. (2018). A simple multi-wave algorithm for the uncapacitated facility location problem. *Frontiers of Engineering Management*, *5*, 451–465.

Gmys, J., Carneiro, T., Melab, N., Talbi, E.-G., & Tuyttens, D. (2020). A comparative study of high-productivity high-performance programming languages for parallel metaheuristics. *Swarm and Evolutionary Computation*, *57*, Article 100720.

Gong, W., Fialho, Á., Cai, Z., & Li, H. (2011). Adaptive strategy selection in differential evolution for numerical optimization: an empirical study. *Information Sciences*, *181*, 5364–5386.

Hakli, H. (2020). Bineho: a new binary variant based on elephant herding optimization algorithm. *Neural Computing and Applications*, *32*, 16971–16991.

Hakli, H., & Ortacay, Z. (2019). An improved scatter search algorithm for the uncapacitated facility location problem. *Computers & Industrial Engineering*, *135*, 855–867.

Harada, T., & Alba, E. (2020). Parallel genetic algorithms: a useful survey. *ACM Computing Surveys*, *53*, 1–39.

He, Y., Zhang, F., Mirjalili, S., & Zhang, T. (2022). Novel binary differential evolution algorithm based on taper-shaped transfer functions for binary optimization problems. *Swarm and Evolutionary Computation*, *69*, Article 101022.

Hoefer, M. (2003). Experimental comparison of heuristic and approximation algorithms for uncapacitated facility location. In *Experimental and efficient algorithms: second international workshop, WEA 2003, Ascona, Switzerland, May (2003) 26–28 proceedings, Vol. 2* (pp. 165–178). Springer.

Husseinzadeh Kashan, M., Husseinzadeh Kashan, A., & Nahavandi, N. (2013). A novel differential evolution algorithm for binary optimization. *Computational Optimization and Applications*, *55*, 481–513.

Karimi-Mamaghan, M., Mohammadi, M., Meyer, P., Karimi-Mamaghan, A. M., & Talbi, E.-G. (2022). Machine learning at the service of meta-heuristics for solving combinatorial optimization problems: A state-of-the-art. *European Journal of Operational Research*, *296*, 393–422.

Kashan, M. H., Nahavandi, N., & Kashan, A. H. (2012). Disabc: a new artificial bee colony algorithm for binary optimization. *Applied Soft Computing*, *12*, 342–352.

Kaya, E. (2022). Bingso: galactic swarm optimization powered by binary artificial algae algorithm for solving uncapacitated facility location problems. *Neural Computing and Applications*, 1–20.

Kennedy, J., & Eberhart, R. C. (1997). A discrete binary version of the particle swarm algorithm. In *1997 IEEE international conference on systems, man, and cybernetics. computational cybernetics and simulation, Vol. 5* (pp. 4104–4108). IEEE.

Kiran, M. S. (2021). A binary artificial bee colony algorithm and its performance assessment. *Expert Systems with Applications*, *175*, Article 114817.

Korkmaz, S., & Kiran, M. S. (2018). An artificial algae algorithm with stigmergic behavior for binary optimization. *Applied Soft Computing*, *64*, 627–640.

Kratica, J., Tošic, D., Filipović, V., & Ljubić, I. (2001). Solving the simple plant location problem by genetic algorithm. *RAIRO-Operations Research*, *35*, 127–142.

Luh, G.-C., Lin, C.-Y., & Lin, Y.-S. (2011). A binary particle swarm optimization for continuum structural topology optimization. *Applied Soft Computing*, *11*, 2833–2844.

Luke, S. (2013). *Essentials of metaheuristics* (2nd ed.). Lulu: Available for free at http://cs.gmu.edu/∼sean/book/metaheuristics/.

Monabbati, E. (2014). An application of a lagrangian-type relaxation for the uncapacitated facility location problem. *Japan Journal of Industrial and Applied Mathematics*, *31*, 483–499.

Monabbati, E., & Kakhki, H. T. (2015). On a class of subadditive duals for the uncapacitated facility location problem. *Applied Mathematics and Computation*, *251*, 118–131.

Özcan, E., Misir, M., Ochoa, G., & Burke, E. K. (2012). A reinforcement learning: great-deluge hyper-heuristic for examination timetabling. In *Modeling, analysis, and applications in metaheuristic computing: advancements and trends* (pp. 34–55). IGI Global.

Peng, B., Zhang, Y., Gajpal, Y., & Chen, X. (2019). A memetic algorithm for the green vehicle routing problem. *Sustainability*, *11*(6055).

dos Santos, J. P. Q., de Melo, J. D., Neto, A. D. D., & Aloise, D. (2014). Reactive search strategies using reinforcement learning, local search algorithms and variable neighborhood search. *Expert Systems with Applications*, *41*, 4939–4949.

Schrage, L. (1975). Implicit representation of variable upper bounds in linear programming. In *Computational practice in mathematical programming* (pp. 118–132). Springer.

Sevaux, M., Sörensen, K., & Pillay, N. (2018). Adaptive and multilevel metaheuristics. In *Handbook of heuristics* (pp. 3–21). Springer.

Song, H., Triguero, I., & Özcan, E. (2019). A review on the self and dual interactions between machine learning and optimisation. *Progress in Artificial Intelligence*, *8*, 143–165.

Sonuç, E. (2021). Binary crow search algorithm for the uncapacitated facility location problem. *Neural Computing and Applications*, *33*, 14669–14685.

Sonuc, E., Sen, B., & Bayir, S. (2018). A cooperative gpu-based parallel multistart simulated annealing algorithm for quadratic assignment problem. *Engineering Science and Technology, An International Journal*, *21*, 843–849.

Sörensen, K., & Glover, F. (2013). Metaheuristics. *Encyclopedia of Operations Research and Management Science*, *62*, 960–970.

Stollsteimer, J. F. (1961). *The effect of technical change and output expansion on the optimum number, size, and location of pear marketing facilities in a california pear producing region*. Berkeley: University of California.

Sutton, R. S., & Barto, A. G. (2018). *Reinforcement learning: an introduction*. MIT Press.

Swan, J., Adriaensen, S., Brownlee, A. E., Hammond, K., Johnson, C. G., Kheiri, A., et al. (2022). Metaheuristics in the large. *European Journal of Operational Research*, *297*, 393–406.

Talbi, E.-G. (2009). *Metaheuristics: From design to implementation, Vol. 74*. John Wiley & Sons.

Talbi, E.-G. (2015). Parallel evolutionary combinatorial optimization. In *Springer handbook of computational intelligence* (pp. 1107–1125). Springer.

Talbi, E.-G. (2021). Machine learning into metaheuristics: A survey and taxonomy. *ACM Computing Surveys*, *54*, 1–32.

Tohyama, H., Ida, K., & Matsueda, J. (2011). A genetic algorithm for the uncapacitated facility location problem. *Electronics and Communications in Japan*, *94*, 47–54.

Tsuya, K., Takaya, M., & Yamamura, A. (2017). Application of the firefly algorithm to the uncapacitated facility location problem. *Journal of Intelligent & Fuzzy Systems*, *32*, 3201–3208.

Wang, D., Wang, D., Yan, Y., & Wang, H. (2008a). An adaptive version of parallel Mpso with Openmp for uncapacitated facility location problem. In *2008 Chinese control and decision conference* (pp. 2387–2391). IEEE.

Wang, D., Wu, C.-H., Ip, A., Wang, D., & Yan, Y. (2008b). Parallel multi-population particle swarm optimization algorithm for the uncapacitated facility location problem using openmp. In *2008 IEEE congress on evolutionary computation (IEEE world congress on computational intelligence)* (pp. 1214–1218). IEEE.

Wauters, T., Verbeeck, K., Causmaecker, P. D., & Berghe, G. V. (2013). Boosting metaheuristic search using reinforcement learning. In *Hybrid metaheuristics* (pp. 433–452). Springer.

Zhang, F., He, Y., Ouyang, H., & Li, W. (2023). A fast and efficient discrete evolutionary algorithm for the uncapacitated facility location problem. *Expert Systems with Applications*, *213*, Article 118978.