The Journal of Supercomputing (2022) 78:4799–4826
https://doi.org/10.1007/s11227-021-04050-6

Discrete hybrid cuckoo search and simulated annealing algorithm for solving the job shop scheduling problem

Faisal Alkhateeb[1] · **Bilal H. Abed-alguni**[1] · **Mohammad Hani Al-rousan**[1]

Accepted: 25 August 2021 / Published online: 10 September 2021

Abstract

The job shop scheduling problem (JSSP) is a popular NP-hard scheduling problem that simulates the scheduling problems of different real-world applications in fields such as manufacturing, engineering, and planning. In general, many optimization-based scheduling algorithms are guaranteed to provide an optimal solution for small instances of the JSSP, but do not perform well with large instances of the JSSP. The hybrid cuckoo search and simulated annealing (CSA) algorithm is a new hybrid optimization algorithm for solving continuous optimization problems. It incorporates the optimization operators of the simulated annealing algorithm into the cuckoo search algorithm. In this paper, we present discrete CSA (DCSA) which is a discrete variation of CSA for solving the JSSP. DCSA incorporates four modifications into CSA. First, it uses the opposition-based learning method in the initialization step to produce diverse candidate solutions. Second, it uses a combination of the variable neighborhood search (VNS) and Lévy-flight methods for better exploration of the search space. Third, it uses the elite opposition-based learning method before the abandon step of CSA to jump out of local optima. Finally, it uses the smallest position value with the JSSP to convert the continuous candidate solutions generated by CSA into discrete ones. DCSA was examined and compared to six popular optimization-based scheduling algorithms (DABC (Yin et al. in Sci Res Essays 6:2578–2596, 2011), PSO-VNS (Tasgetiren et al. in Int J Oper Res 3:120–135, 2006), DCS (Quaarab et al., in: Yang (ed) Recent advances in swarm intelligence and evolutionary computation, Springer, New York, 2015), GWO (Jiang and Zhang in IEEE Access 6:26231–26240, 2018), DWPA (Wang et al. in 2019 5th International Conference on Control, Automation and Robotics ICCAR 2019 581–585, 2019) and DGA (Kalshetty et al. in J Sci Res 64:310–321, 2020) using 34 JSSP instances from the OR-Library: Fisher and Thompson (3 instances), and Lawrence (31 instances). The experimental and statistical results indicate that DCSA converges faster to the Best-Known Solution for 29 instances out of the 34 tested JSSP instances and requires less computational time for the 34 tested instances than the computational times of the other algorithms.

Extended author information available on the last page of the article

Keywords Cuckoo search · Simulated annealing · Job shop scheduling problem · Opposition-based learning · Hybrid cuckoo search and simulated annealing algorithm

1 Introduction

The job shop scheduling problem (JSSP) is a well-known NP-hard combinatorial problem that simulates the scheduling problems in industrial and decision-making applications [7–10]. Several heuristic and optimization-based scheduling algorithms have been proposed to solve the JSSP with the objectives of increasing the efficiency of manufacturing and reducing its cost [1, 3, 5–15]. However, many optimization-based scheduling algorithms are guaranteed to provide optimal schedules for small instances of the JSSP, but do not provide efficient schedules for large instances of the JSSP. This is possible due to three reasons. First, many optimization algorithms such as genetic algorithms and cuckoo search (CS) easily fall into local optima [3, 16–18]. Second, some optimization algorithms such as grey wolf optimizer (GWO), whale optimization algorithm (WOA), and differential evolution algorithm suffer from the premature convergence problem [19–25]. Third, the optimization operators in continuous optimization algorithms may not be successful in solving discrete problems when converted to discrete operators [17, 26].

Alkhateeb and Abed-Alguni [27] proposed the hybrid cuckoo search and simulated annealing (CSA) algorithm, which is a new hybrid optimization algorithm for solving continuous optimization problems. CSA incorporates the optimization operators of the simulated annealing (SA) algorithm [28] into the improvement loop of the cuckoo search (CS) algorithm [29]. The experimental and statistical results in [27] suggest that CSA performs better than the particle swarm optimization (PSO) algorithm [30], bat Algorithm (BA) [31], Boltzman CS algorithm, and ε-greedy CS algorithm [32]. Besides, the results indicate that CSA has a stable reliable performance with various problem sizes (small, medium, and large) compared to the other algorithms. This is because CSA has several advantages compared to CS. First, the optimization operators of the SA algorithm can explore the solution space by replacing the current solutions by the worse solutions with a decreasing probability. Second, the integration of the CS and SA operators helps the CSA algorithm to solve the premature convergence problem.

Variable neighborhood search (VNS) is an effective metaheuristic used for solving combinatorial optimization problems [33]. The main objective of using the VNS algorithm is to improve the current solution by applying a set of operations (e.g., mutation) to the solution to generate a neighboring solution that may help to jump out of local optima [27]. The basic idea of VNS is to iteratively move from one neighboring solution to another one in an attempt to find the best candidate solution [34].

In this paper, we introduce a discrete variation of CSA by the name discrete CSA (DCSA) for solving the JSSP. DCSA uses the opposition-based learning (OBL) in its initialization step and elite opposition-based learning (EOBL) in its final steps to improve its exploration abilities. It also uses a combination of the VNS and

Lévy-flight methods to explore the search space. The generated solutions using the original CSA algorithm are continuous solutions; therefore, DCSA uses the smallest position value (SPV) method with the JSSP to convert the continuous candidate solutions generated by CSA into discrete ones.

We are encouraged to use the OBL method in the initialization step of DCSA because it improves the diversity of candidate solutions. We are also interested to use EOBL in DCSA because it can speed up the convergence speed to optimality by exploring the opposite solutions of the best-known candidate solutions. Besides, we are interested to use the combination of the VNS and Lévy-flight methods in the DCSA algorithm because this combination has been used successfully to explore the search space of discrete optimization problems [2].

We experimentally evaluated and compared DCSA to six popular optimization-based scheduling algorithms (the discrete artificial bee colony-based memetic algorithm (DABC) [1], hybrid particle swarm optimization and variable neighborhood search (PSO-VNS) algorithm [2], discrete cuckoo search (DCS) algorithm [3], grey wolf optimizer (GWO) [4], discrete wolf pack algorithm (DWPA) [5] and discrete genetic algorithm (DGA) [6]) using 34 instances from two benchmark sets: Fisher and Thompson (3 instances) and Lawrence (31 instances). The experimental and statistical results suggest that DCSA converges to the best-known solutions faster than the other algorithms. The main contributions of this paper are as follows:

1. We propose discrete CSA (DCSA), which is a discrete variation of CSA for solving the JSSP.
2. We use the OBL method in the initialization step of DCSA to produce a diverse population of candidate solutions.
3. We use a combination of the VNS and Lévy-flight methods to explore the discrete search space of the JSSP in an efficient manner.
4. We employ the EOBL method before the abandon step of DCSA to increase the likelihood of finding an optimal solution.
5. We apply the SPV method in the DCSA algorithm to convert the continuous candidate solutions generated by DCSA into discrete ones.
6. We conducted several experiments to evaluate and compare DCSA to six popular optimization-based scheduling algorithms (DABC, PSO-VNS, DCS, GWO, DWPA and DGA) using 34 instances of two benchmark sets from the OR-Library: Fisher and Thompson (3 instances), and Lawrence (31 instances). The overall experimental results show that the DCSA algorithm outperformed the other algorithms for 29 out of 34 instances with less computational time compared to the other tested algorithms.
7. We conducted an experimental comparison between the exploration methods used in DCSA using the 34 JSSP instances from the OR-Library. The results clearly show that DCSA combined with all its exploration methods is the best performing algorithm. It achieved the BKS for 33 out of the 34 tested instances. The results also show that the order of the best exploration methods that were used in DCSA individually is as follows: DCSA with EOBL (24 BKS), DCSA with VNS and Lévy flight (23 BKS), and finally DCSA with OBL (18 BKS).

The rest of the paper is organized as follows: In Sect. 2, we critically review recent research studies that attempted to solve the JSSP using optimization-based scheduling algorithms. In Sect. 3, we discuss the preliminaries of the research. In Sect. 4, we present the discrete cuckoo search and simulated annealing (DCSA) algorithm. In Sect. 5, we present the experimental results. Finally, we present the conclusion of this paper in Sect. 6.

2 Literature review

The JSSP is an NP-hard problem that is commonly used to evaluate the performance of discrete optimization algorithms using benchmarks sets such as Taillard's set, Fisher and Thompson's set [35], and Lawrence's set [36]. In this section, we critically review recently proposed discrete optimization-based scheduling algorithms for solving the JSSP.

Nazif [37] used the ant colony algorithm (ACA) to solve the JSSP with the objective of minimizing the completion time. ACA uses a new method to initialize the pheromone trails using an initial sequence of operations. Besides, ACA iteratively uses a local search method to improve its exploration capabilities. Hence, it requires high number of computations at each iteration compared to classical optimization algorithms.

Ouaarab et al. [3] presented the discrete cuckoo search (DCS), which aims to minimize the makespan of the JSSP. DCS simulates the Lévy-flight mutation method using three discrete operators: swapping, insertion, and inversion. However, the DCS algorithm was experimentally compared in [3] to only the particle swarm optimization (PSO) algorithm using 16 benchmarks from the OR-Library. Therefore, we will further investigate the performance of DCS in the experimental section of our paper.

Banharnsakun et al. [11] used a variation of the artificial bee colony algorithm called the best-so-far ABC algorithm to minimize the overall makespan of the JSSP. This algorithm iteratively moves the candidate solutions toward the best-so-far solution in the population of candidate solutions instead of moving them toward a neighboring solution as done in the original ABC algorithm. However, moving the candidate solutions repeatedly toward the best solutions causes imbalance between exploration and exploitation of the search space, which may degrade the performance of the algorithm [32].

Asadzadeh [12] proposed an agent-based local-search genetic algorithm for solving the JSSP. This algorithm uses various search agents, where each agent can independently perform the genetic algorithm and then refine its solutions using two-stage local search. This means that the agent-based local-search genetic algorithm requires more computation than the basic genetic algorithm (GA). In addition, the agent-based local-search genetic algorithm should be executed in parallel machines to reduce its computational complexity.

Kalshetty et al. [6] proposed a discrete variation of GA (DGA) for solving the JSSP. DGA uses the generalized order crossover and swap methods to explore the

search space. The performance of DGA was evaluated in [6] using 64 benchmark datasets from the OR-library, but it was not compared to state-of-the-art algorithms.

Singh et al. [38] suggested cuckoo search optimization (CSO) with assorted individual enhancement scheme to solve the JSSP. The assorted individual enhancement scheme is a discrete mutation method that uses four mutation operators: swapping, insertion, inversion, and wide movement. This method is used to improve the quality of the solutions generated by the Lévy-flight-based mutation method in CS. The CSO was evaluated in [38] to one algorithm (basic CS) and found to be more efficient than CS. Thus, further investigation and comparisons with popular optimization-based scheduling algorithms are required to evaluate the performance of CSO.

Tasgetiren et al. [2] hybridized the particle swarm optimization (PSO) [39] algorithm with the variable neighborhood search (VNS) in a new algorithm called PSO-VNS. This algorithm was developed for solving the JSSP. PSO-VNS uses the SPV method for mapping the real values in particles (candidate solutions) to discrete values. It uses the VNS algorithm inside PSO to prevent it from stucking in local optimal solutions and improve its solutions. The experimental results in [2] using widely used benchmarks from the OR-library suggest that PSO-VNS is better than PSO and VNS. Therefore, we will examine the performance of PSO-VNS compared to DCSA in the experimental section of our paper to gain a better insight into the performance of PSO-VNS.

Wang et al. [5] presented a discrete wolf pack algorithm (DWPA) to solve the JSSP. DWPA uses encoding and decoding methods to represent the processing time and machine order for each operation in the schedule. In [4, 40], the GWO algorithm was used to minimize the completion time of the JSSP. This new GWO variation uses the crossover and mutation operators to improve and keep the candidate solutions around the global optima. Furthermore, it applies the VNS algorithm using three operators (swapping, insertion, inversion) to the best three solutions in the population.

Alawad et al. [17] introduced a discrete Jaya algorithm by the name discrete Jaya with refraction learning and three mutation method (DJRL3M) for solving the permutation flow shop scheduling problem. DJRL3M uses refraction learning (RL) (i.e., a special type of opposition-based learning that is based on the phenomenon of refraction) to improve the diversity of its initial population of solutions. It also utilizes RL in its optimization loop to improve the best and worst solutions. It uses three mutation methods (DJaya mutation [41], highly disruptive polynomial mutation [17] and Pitch Adjustment mutation [42]) to globally explore the search space of an optimization problem. DJRL3M uses the abandon method generate random solutions instead of the worst solution in a population at each iteration of it optimization loop. Finally, it uses the smallest position value (SPV) as a discretization method for the permutation flow shop scheduling problem. DJRL3M was compared to well-known optimization-based scheduling algorithms such as discrete cuckoo search [41] and generalized accelerations for insertion-based [43] using a set of Taillard's benchmark instances. The results indicate that DJRL3M outperforms all the tested algorithms.

In summary, many optimization algorithms in this section such as ABC, ACA, GA, PSO, BA, DGA and GWO have internal parameters that require fine tuning.

Other optimization algorithms are hybrid algorithms (e.g., PSO-VNS, ACA with local search, agent-based local-search GA) that require heavy computations compared to classical algorithms. In this article, we employ CSA as the foundation for our novel discrete scheduling algorithm to address the limits and drawbacks of several current optimization-based scheduling algorithms. This is because CSA has faster convergence rate than well-known algorithms such as PSO, BA, SA, Boltzman CS, and ε-greedy CS [27]. Also, CSA has no parameters that require fine tuning. In addition, CSA can be converted easily to a discrete optimization algorithm using the SPV method as described in Sect. 3.2.

3 Preliminaries

This section is divided into five subsections. Section 3.1 describes the job shop scheduling problem (JSSP), Sect. 3.2 describes the smallest position value (SPV) method, Sect. 3.3 describes the opposite-based learning (OBL) method, Sect. 3.4 discusses the cuckoo search (CS) algorithm, and finally Sect. 3.5 describes the hybrid cuckoo search and simulated annealing (CSA) algorithm.

3.1 Job shop scheduling problem

The JSSP is a combinatorial optimization problem that has various applications in manufacturing and commerce. The JSSP is basically composed of a sequence of n jobs ($J = \{J_1, J_2, \ldots, J_n\}$), and a sequence of m machines ($M = \{M_1, M_2, \ldots, M_m\}$). Each job is composed of m operations to be processed ($J_i = \{O_{i1}, O_{i2}, \ldots, O_{im}\}$). Each machine M_k can process only one operation at a time, and the machine or the operation cannot be interrupted. In other words, an operation must be processed on only one machine for one time in the same order of operations specified in the job and a machine cannot process more than one operation at a time. Therefore, the total number of operations required is $(n \times m)$. Simply, the main goal of solving JSSP is to produce an optimal schedule from $(n!)^m$ possible schedules [3]. In the JSSP, two conditions must be met before processing an operation:

- A machine should be available and ready before processing an operation.
- The previous operation $O_{i\,(j\text{-}1)}$ in job J_i must finish executing before executing the current operation O_{ij}.

Therefore, the completion of execution time is called makespan ($C_{\max}$). The optimization function of the JSSP is as follows:

$$C_{\max} = \min_{1 \le i \le N} C_i \tag{1}$$

where C_i is the total execution time of the ith candidate solution.

3.2 Smallest position value method

The smallest position value (SPV) method is a heuristic technique, which was presented by Tasgetiren et al. [44]. SPV is used to convert candidate solutions with real numbers (continuous candidate solutions) into candidate solutions with integer numbers (discrete candidate solutions) [45]. More precisely, the SPV method sorts the continuous position values in a candidate solution in ascending order to determine a sequence of operations. The SPV method was experimentally proven in [16, 17] that it improves the diversity of the population. Assume that w is the number of candidate solutions of $X = \langle x_1, x_2, \ldots, x_w \rangle$, and the ith candidate solution consists of h decision variables $x_i = \langle \vec{x}_{i1}, \vec{x}_{i2}, \ldots, \vec{x}_{ih} \rangle$. These decision variables consist of continuous values with lower and upper bounds to represent each operation in a scheduling problem. So, SPV sorts the operations in another list y_{ij} by sorting the continuous values of decision variables from the smallest to the largest value. Table 1 demonstrates an example of the SPV method.

3.3 Opposition-based learning methods

Opposition-based learning (OBL) is a relatively new algorithm in machine learning that simulates the opposite relationships between the components of a system. The integration of OBL into an optimization algorithm is beneficial when the current solution is relatively far from the optimal solution. This is clear when the optimal solution is in the opposite direction of the current solution. It has been demonstrated and mathematically proven in [46, 47] that searching for better candidate solutions with pure random method along with the OBL method increases the likelihood of finding the optimal solution compared to pure random search.

According to Mahdavi et al. [47], the OBL algorithm can be applied to single and multiple dimensions variables. The definitions below show how OBL can be applied to single-dimensional variables and multi-dimensional variables.

Definition 1 Let $x \in \mathbb{R}$ be a single-dimensional variable within the interval $[a, b]$. The opposite variable of x ($\check{x}$) is defined as follows:

$$\check{x} = a + b - x \tag{2}$$

Definition 2 Let $q = (q_1, q_2, \ldots, q_d)$ be a multi-dimensional variable with d dimensions, where $q_i \in [a_i, b_i]$, $\forall\ i \in \{1, 2, \ldots, d\}$. The opposite variable of q, which is denoted as $\check{q} = (\check{q}_1, \check{q}_2, \ldots, \check{q}_d)$, is defined as follows:

$$\check{q}_i = a_i + b_i - q_i \tag{3}$$

Table 1 An example of the SPV method

Dimensions	1	2	3	4	5	6	7
x_{ij}	3.74	−1.64	0.97	−3.25	−2.45	5.23	1.41
y_{ij}	4	5	2	3	7	1	6

Algorithm 1: OBL Algorithm

1: Input: solution S
2: Create the opposite $\check{S}$
3: Calculate the fitness value for $f(S)$ and $f(\check{S})$
4: if $f(S) > f(\check{S})$ then
5: replace S with $\check{S}$
6: end if

Elite opposition-based learning (EOBL) is a variation of OBL [48]. It is based on the idea that the opposite solutions of the best solutions (elite solutions) have higher probability to be closer to the global optima than the other candidate solution. In the EOBL method, a fraction of the best solutions is selected and then OBL is applied to the selected solutions.

3.4 Cuckoo search algorithm

In the last decade, Yang and Deb [27, 29] presented a metaheuristic called cuckoo search (CS). This algorithm follows a simulation model that is based on the parasitic breeding behavior (i.e., the behavior of cuckoo birds that trick other birds (host birds) to take care of their eggs) of cuckoo birds. In nature, the host bird may discover that its eggs have been replaced. In this situation, the host bird may either throw the suspected eggs or abandon its nest. The CS algorithm can find solutions to optimization problems by simulating two bird behaviors: the exploitative breeding behavior of some cuckoo species and the Lévy-flight behavior of some bird species. The CS algorithm consists of three main rules:

- For each generation, each cuckoo drops only one egg in a randomly chosen bird nest to represent the candidate solution.
- The nests of the best quality (best solutions) will be kept to the reproduction of the next generation.
- The number of nests is fixed, and only a few nests are abandoned (worst solutions) with probability ($P_a \in [0, 1]$).

In the CS algorithm, the Lévy flight is used in the mutation function to generate a new nest (candidate solution) as follows:

$$x_i^{t+1} = x_i^t + \alpha \oplus \text{lévy}(\beta) \tag{4}$$

where x_i^t represent the ith candidate solution at t generation, α is the step value related to the size of the determined problem, $\oplus$ is the entry wise product, and lévy(β) is a random walk with a step length generated from a Lévy-flight distribution as follows:

$$\text{lévy} \sim u = S^{-\beta} \tag{5}$$

where S is the step size and $\beta \in (1, 3]$ is a variable for fractal dimension. In detail, β is a ratio that measures the complexity by comparing how detail in a pattern varies with scale. Note that $\beta \leq 1$ cannot normalized, and when $\beta \geq 3$, the Lévy distribution turns into a Gaussian distribution. This means that the lengths of a Lévy flight follow the power-law distribution with $1 < \beta \leq 3$ [49]. It is important to know that the Lévy-flight distribution has an exponential variation with infinite mean, which means that the Lévy flights will get longer over the course of its simulation process.

Algorithm 2: Cuckoo Search Algorithm

1: Begin
2: Objective function $f(x)$, $x = (x_1, x_2, \ldots, x_m)$
3: Generate initial population of N host nests x_i (i = 1, 2, ..., N)
4: while ($t <$ *maxIterations*) or (*stop criterion*) do
5: Get a cuckoo randomly by Lévy flights
6: Evaluate its quality/fitness $f(x_i)$
7: Choose a nest among N (*say* j) randomly
8: if $f(x_i) > f(xj)$ then
9: replace j by the new solution;
10: end if
11: A fraction (P_o) of worse nests are abandoned and new ones are built;
12: Keep the best solutions (or nests with quality solutions);
13: Rank the solutions and find the current best;
14: end while
15: End

3.5 Hybrid cuckoo search and simulated annealing

The simulated annealing (SA) is a probabilistic, optimization method that was proposed by Kirkpatrick et al. [27, 28]. SA simulates the annealing process of metal that requires balancing between the heating and cooling processes of metal to make it into a desired shape. SA is used to approximate the global optimal solution of an optimization problem by iteratively generating a neighboring solution around the current solution and then deciding to keep the current solution or replace it with the neighboring solution. This means that SA gives a chance for the worst solution to be improved, which is accomplished probabilistically using a decreasing parameter called temperature (T). In the beginning, SA randomly generates a candidate solution (say A) and then randomly generates another solution (say B). Afterward, it evaluates the fitness values of A and B. If the fitness value of B ($f(B)$) is better than the fitness value of A ($f(A)$), then SA accepts B. Otherwise, SA accepts B based on a decreasing probability ($p(B)$):

$$\Delta E = f(B) - f(A) \tag{6}$$

$$p(B) = \exp^{\Delta E/T} \tag{7}$$

where ΔE is the difference between the fitness values of A and B, T is the current temperature value, and exp is the exponential value.

Algorithm 3: Simulated Annealing Algorithm

```
1: Function SIMULATED-ANNEALING (problem, schedule, iterations, LB, UB) return a solution
state
2: input: problem: a problem. schedule: a mapping from time to temperature. iterations: is the
number of iterations. LB: the lower bound of the search range of the problem. UB: the upper
bound of the search range of the problem.
3: local variables: current: a node. next: a node. T: a temperature controlling the prob. of
downward steps
4: Begin
5: current ← Make Node (initial-State[problem])
6: for t ← 1 to iterations do
7:        T ← schedule[t]
8:        if T = 0 then
9:                return current
10:       end if
11:       next ← a randomly selected solution with [LB, UB]
12:       ΔE ← VALUE [next] - VALUE [current]
13:       if ΔE < 0 then
14:               current ← next
15:       else
16:               current ← next, only with probability exp^(ΔE/T)
17:       end if
18: end for
19: End
```

The standard CS algorithm does not perform well and suffers from the premature convergence problem when it is applied to solve optimization problems with high dimensions. This is because its evolutionary operators (i.e., Lévy flight, abandon method, random selection method) may not be able to explore the search space of large optimization problems efficiently. On the other hand, SA has efficient evolutionary operators that balance between exploration and exploitation of solutions. Therefore, Alkhateeb and Abed-Alguni [27] integrated CS and SA in a new hybrid algorithm called the hybrid CS and SA (CSA) algorithm. CSA aims to improve the performance of CS by applying the evolutionary operators of SA to refine the solutions generated in CS using Lévy flights (Lines 5–18 in Algorithm 4). SA is used in CSA to increase the likelihood of finding a global optimal solution and avoid being trapped within a sub-optimal value (Fig. 1).

Initial representation of solution x_i

Dimension	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
x_{ij}	1.37	-5.3	-0.36	6.3	5.24	-1.9	1.1	-3.7	-0.17	2.7	-7.5	-2.13	2.1	3.15	8.57	-1.9

Applying the SPV on solution x_i

x_{ij}	-7.5	-5.3	-4.9	-3.7	-2.13	-1.9	-0.36	-0.17	1.1	1.37	2.1	2.7	3.15	5.24	6.3	8.57
S_{ij}	10	1	15	7	11	5	2	8	6	0	12	9	13	4	3	14

Job permutation of solution S_i

S_{ij}	10	1	15	7	11	5	2	8	6	0	12	9	13	4	3	14
Job Permutation	J_3	J_2	J_4	J_4	J_4	J_2	J_3	J_1	J_3	J_1	J_1	J_2	J_2	J_1	J_4	J_3

Fig. 1 An example of initial representation of solution x_i

Algorithm 4: Hybrid Cuckoo Search and Simulated Annealing Algorithm (CSA)

```
Begin
Objective function f(x), x = (x1,x2,...,xm)
Generate initial population of N host nests xi (i = 1, 2, ..., N)
while (t < maxIterations) or (stop criterion) do
    for T ← (maxIterations - t) to 0 do
        T ← schedule[t]
        current ← get a random solution from population
        if T = 0 then
            return current
        end if
        next ← generate a random solution by Lévy flights
        ΔE ← VALUE [next] - VALUE [current]
        if ΔE < 0 then
            current ← next
        else
            current ← next, only with probability exp^(ΔE/T)
        end if
    end for
    A fraction (fa) of worse nests are abandoned and new ones are built;
    Keep the best solutions (or nests with quality solutions);
    Rank the solutions and find the current best;
end while
End
```

4 Proposed algorithm: discrete CSA

In this section, we describe a new discrete optimization algorithm called discrete CSA for solving the JSSP. The flowchart of the DCSA algorithm is shown in Fig. 2. First, DCSA generates an initial population for a given JSSP instance using a random generation method and the OBL method as discussed in detail in Sect. 4.1. Second, DCSA applies the SPV method to the generated continuous candidate

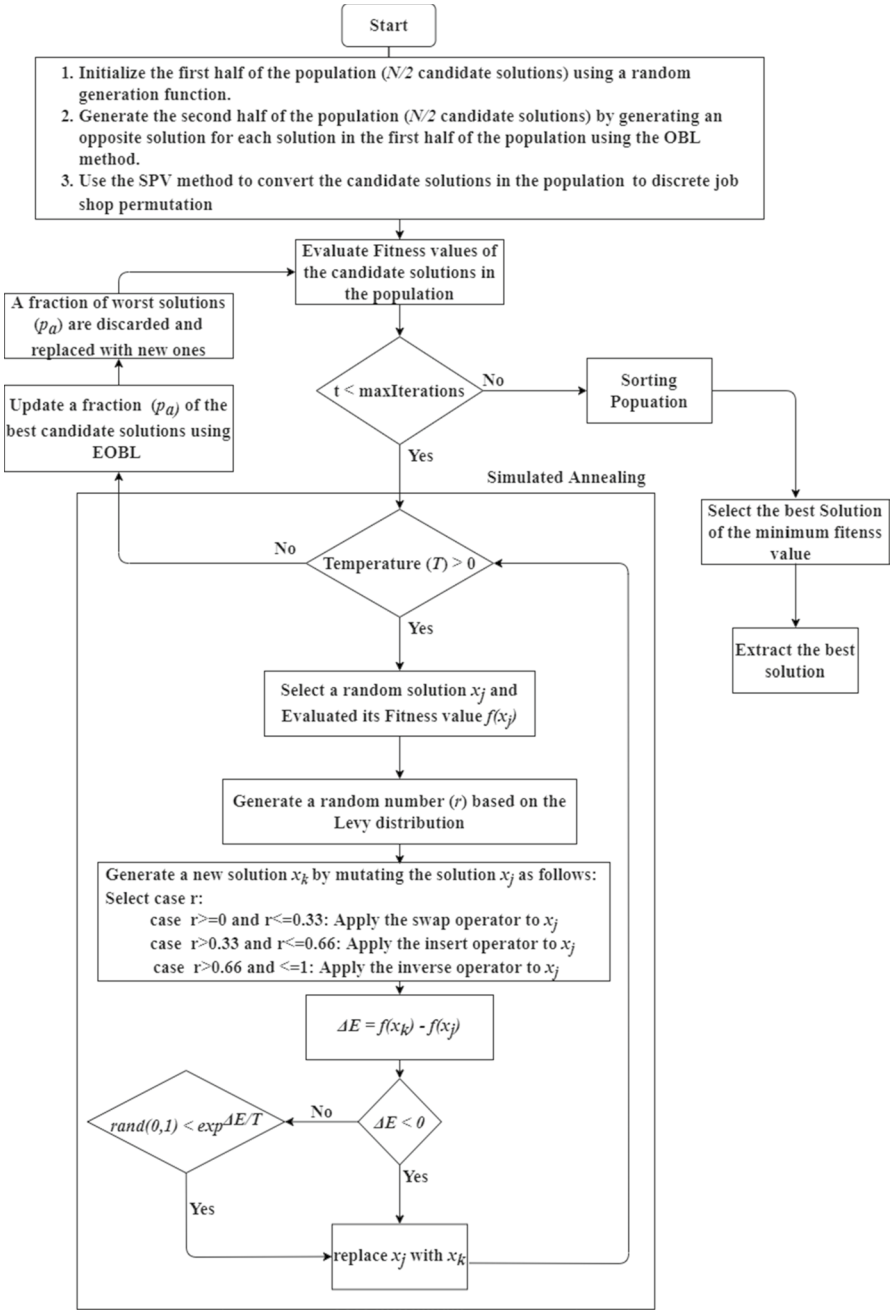


Fig. 2 Flowchart of DCSA for solving JSSP

Table 2 Machine order for JSSP

Jobs	Machine order			
J_1	1	2	3	4
J_2	2	1	4	3
J_3	1	3	4	2
J_4	3	2	1	4

Table 3 Processing time for JSSP

Jobs	Processing time			
J_1	5	3	5	2
J_2	4	3	5	7
J_3	2	4	6	5
J_4	3	3	4	2

solutions to convert them to discrete job shop permutations as discussed in the end of Sect. 4.1. Third, DCSA calculates the fitness values for the discrete job permutations using Eq. 1 in Sect. 3. Fourth, DCSA attempts to improve the current population by applying the SA algorithm to a randomly selected solution (x_j) from the population (Sect. 4.2). Fifth, DCSA generates a random number (r) between 0 and 1 based on the Lévy distribution. The random number r is used as a probability to choose among three moves: swap with probability 33.3%, insert with probability 33.3%, or inverse with probability 33.3% (Sect. 4.2). The new generated solution is called x_k. Sixth, if the new solution x_k is worse than the current solution x_j, then DCSA accepts x_k to replace x_j according to exponentially decreasing probability (Sect. 3.5). Finally, DCSA attempts to improve a fraction (P_a) of the best solutions using EOBL and then it replaces a fraction (P_a) of the worst solutions with new random solutions. Upon the end of the optimization loop of DCSA, it sorts the candidate solutions according to their fitness values and then returns the solution with the best fitness values.

4.1 Initialization of population

The first step of DCSA, as shown in Fig. 2, is to generate the initial population for a given JSSP. The first half of the population is generated using a random generation function, while the second half is generated by applying the OBL to the first half of the population. The details of the initialization step are provided in this section.

First, we start by describing the representation of a candidate solution. The ith candidate solution (x_i) consists of a set of ($n \times m$) decision variables (dimensions) that represents a sequence of operations $x_i = \langle \vec{x}_{i1}, \vec{x}_{i2}, \ldots, \vec{x}_{i(n\times m)} \rangle$, where n is the number of jobs and m is the number of machines. Tables 2 and 3 represent an example of a 4×4 JSSP benchmark (16 operations), which represent the machine order and processing time, respectively, for each operation.

The first half of the population is generated using a random generation function. The following equation describes how to generate a random value for each operation in solution x_i:

$$x_{ij} = \text{LB} + (\text{UB} - \text{LB}) \times \text{rand} \tag{8}$$

where rand is a random value between [0,1], LB = − 10, and UB = 10. After that, we use the SPV to produce a list of $S_i = (S_{i1}, S_{i2}, \ldots, S_{i(n\times m)})$ for mapping real numbers to integer numbers of solution x_i in ascending order. Each integer number (S_{ij}) represents one operation in the sequence S_i. A job permutation can be produced for all integer values in the sequence $S_{i,}$ as follows [3]:

$$S_{ij} \bmod n \tag{9}$$

where n is the number of jobs. Figure 1 shows an example of the SPV method. First, a random solution $x_i = \langle 1.37, -5.3, \ldots, 8.57, -1.9 \rangle$ is generated using Eq. 8. Second, the SPV method is used to produce a list of $Si = (10, 1, \ldots, 3, 14)$ where each integer S_{ij} represents one operation in the sequence S_i. The values in S_i represent the indices that arrange x_i in ascending order. A job permutation $< 10, 1, \ldots, 3, 14 >$ is produced for all integer values in the sequence S_i by calculating S_{ij} mod 16.

Then, we use OBL to generate an opposite solution according to [47, 50] using the flowing equation:

$$\check{j}_i = 1 + n - J_i \tag{10}$$

where J_i is the job permutation of the current task and $\check{j}_i$ is opposite for it and n is the number of jobs for each benchmark instance.

4.2 Improving solutions using DCSA

As shown in Fig. 2, the DCSA algorithm attempts to improve its population of candidate solutions through three stages: improving solutions using VNS and Lévy flight, acceptance of solutions using SA, and improving abandon step using EOBL.

4.2.1 Improving solutions using VNS and Lévy flight

Variable neighborhood search (VNS) is an effective metaheuristic used for solving combinatorial optimization problems [33]. The main objective of using the VNS algorithm is to improve the current solution by applying a set of operations (e.g., mutation) to the solution to generate a neighboring solution that may help to jump out of local optima [27]. The basic idea of VNS is to iteratively move from one neighboring solution to another one in an attempt to find the best candidate solution [34].

In this stage, DCSA randomly chooses a candidate solution from the current population of solutions. Afterward, the algorithm uses a combination of the VNS method and the Lévy flight method to explore the neighborhood of the selected solution as suggested in [3]. In detail, the Lévy flight method generates a random

number r between 0 and 1, which is used as a probability to choose among three moves: swap, insert, or inverse.

$$x_k = \begin{cases} 0 \le r < 0.33 & \text{Apply Swap Operator to} x_j \\ 0.33 \le r < 0.66 & \text{Apply Insert Operator to } x_j \\ 0.66 \le r < 1 & \text{Apply Inverse Operator to} x_j \end{cases} \tag{11}$$

In the swap operator, two positions are selected randomly in the candidate solution (w and q) and then the two operations in these positions are exchanged as shown in Fig. 3.

The insert operator randomly selects two positions (w and q), and then it removes the first operation from the first position and add it into the second position (Fig. 4).

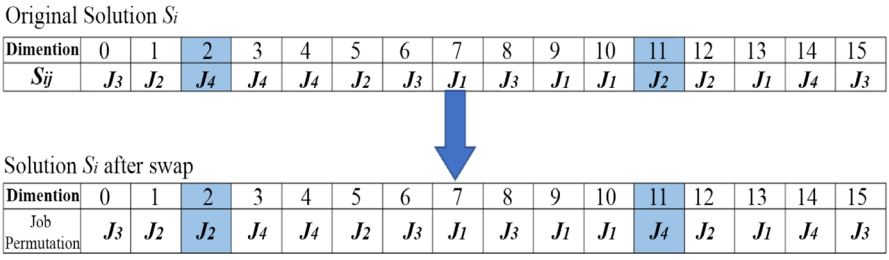


Fig. 3 Swap operator between position 2 and position 11

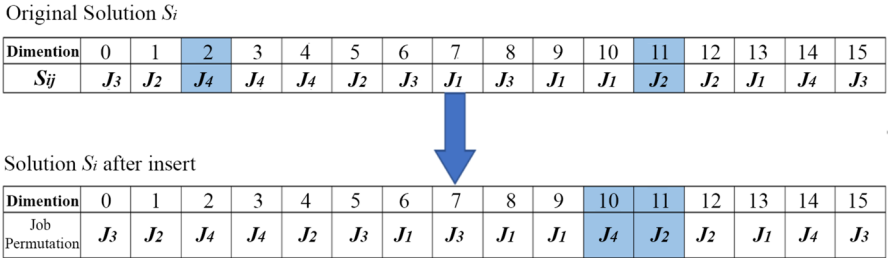


Fig. 4 Insert operator between position 2 and position 11

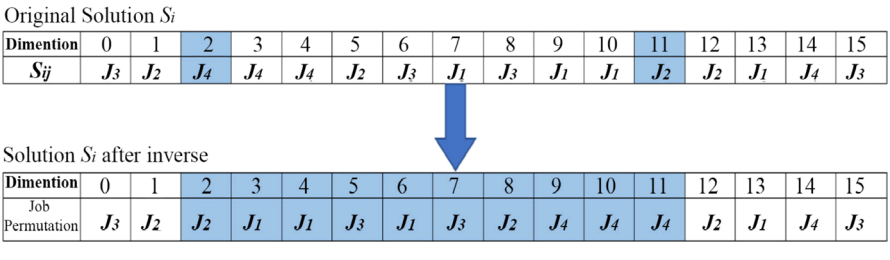


Fig. 5 Inverse operator between position 2 and position 11

As shown in Fig. 5, the inverse operator selects two random positions (w and q) and then it inverses the order of operations between the two positions.

4.2.2 *Acceptance of solutions using SA*

In the beginning of this stage, the difference between x_k and xj is calculated ($\Delta E = f(x_k) - f(x_j)$). Then, the SA algorithm accepts the new solution x_k and replaces it with the current solution x_j in two cases. First, if x_k is better than x_j ($\Delta E > 0$ for minimization problems), x_k is accepted to replace x_j. Second, if x_k is worse than x_j, then x_k is accepted to replace x_j according to exponentially decreasing probability (rand $(0, 1) > e^{\Delta E/T}$, Sect. 3.5).

4.2.3 Improved abandon step using EOBL

Finally, in the third stage, DCSA attempts to improve a fraction (P_a) of the best solutions using EOBL. DCA applies the inverse operator to each elite solution to explore the search area around it. The inverse operator is applied as described at the end of Sect. 4.2.3.

Finally, a fraction P_a of the worst solutions is replaced with new random solutions.

5 Experimental results

We compared the performance of the DCSA algorithm to the performance of three popular optimization-based scheduling algorithms (the discrete artificial bee colony-based memetic algorithm (DABC) [1], hybrid particle swarm optimization and variable neighborhood search (PSO-VNS) algorithm [2], and discrete cuckoo search (DCS) [3] algorithm) using 34 benchmark instances from the OR-Library [51]: Fisher and Thompson (1963) [35], and Lawrence (1984) [36]. The selected 34 benchmark instances have been used extensively in the literature to evaluate the performance of many scheduling algorithms in solving the JSSP [1–4, 6, 7, 12, 52].

We also compared the performance of DCSA to the performance of three powerful optimization algorithms (GWO [4], DWPA [5] and DGA [6]) using the 34 selected JSSP from the OR-Library. The simulation results of GWO, DWPA and DGA were taken from their original papers.

The algorithms were implemented using Java and the experiments were executed on a computer with the following specifications: Intel(R) Core (TM) i7-7500U CPU @ 2.70 GHz 2.90 GHz, and 8 GB RAM running Windows 10. Each benchmark instance was executed independently for 10 times. The parameters settings that was used with DCSA are described in [27, 29]. Table 4 presents the parameter settings of DCSA.

The parameter settings of DABC, PSO-VNS and DCS were adopted from [1, 2] and [3], respectively. The best results in the tables in this section are marked with Bold Font.

Table 4 Parameter settings of DCSA algorithm

Parameter	Value
Number of iterations (maxIter)	1500
Population size (N)	50
Temperature (T)	1500
Cooling rate (c)	0.1
Fraction of abandon solutions (p_a)	0.25
Step size of the Lévy flight (L)	1
Lower bound (LB)	− 10.0
Upper bound (UB)	10.0

5.1 Comparison between DCSA and other algorithms

Table 5 shows the experimental results of DCSA vs the experimental results of DABC, PSO-VNS and DCS. In the table, the first column represents the instance name, and the second column represents the Best-Known Solution (BKS). The best fitness value (Best) obtained over 1500 iterations and the relative percent error (RPE%) are shown in the table for each algorithm. RPE% is the difference between the Best and BKS. RPE% is computed as follows:

$$\mathrm{RPE} = \frac{\mathrm{Best} - \mathrm{BKS}}{\mathrm{BKS}} \times 100 \tag{12}$$

The experimental results in Table 5 show that DCSA achieved the BKS for 29 out of the 34 instances. DCS is the second-best performing algorithm with 23 out of the 34 instances, followed by DABC which achieved the BKS for 12 instances. The worst performing algorithm is PSO-VNS which achieved the BKS for 11 instances. The RPE% of DCSA is the same as DABC, PSO-VNS, and DCS for FT06, LA01, LA05, LA06, LA08 to LA14. This is maybe because these JSSP instances are small and medium instances. In addition, all the tested algorithms use efficient exploration methods: DABC uses three mutation methods and a local search method, DCS uses random walks based on Lévy-flight, and PSO-VNS uses a combination of the VNS and Lévy-flight methods. The RPE% of DCSA and DCS are the same for LA26 to LA35. This is possibly because both algorithms are based in the CS algorithm and use the Lévy-flight as an exploration method. DCSA did not achieve the BKS for FT20 because the number of jobs in FT20 is 20 which is considered relatively high.

Figure 6 represents the RPE% values of the compared algorithms. The RPE% values of the DCSA algorithm are the lowest for all the algorithms. The efficiency of DCSA was four times better than DCS. The superior performance of DCSA may be because it integrates SA into CS and uses different exploration methods (VNS with Lévy flight, OBL and EOBL). These methods increase the ability of DCSA to find better solutions in the search space.

Table 5 Computational comparison between DCSA, DABC, PSO-VNS, and DCS

Instance	BKS	DCSA		DABC		PSO-VNS		DCS	
		Best	RPE%	Best	RPE%	Best	RPE%	Best	RPE%
FT06	55	**55**	**0**	**55**	**0**	**55**	**0**	**55**	**0**
FT10	930	**930**	**0**	1005	8.065	1054	13.333	943	1.398
FT20	1165	**1174**	**0.773**	1300	11.588	1253	7.554	1178	1.116
LA01	666	**666**	**0**	**666**	**0**	**666**	**0**	**666**	**0**
LA02	655	**655**	**0**	669	2.137	685	4.580	**655**	**0**
LA03	597	**597**	**0**	615	3.015	615	3.015	604	1.173
LA04	590	**590**	**0**	594	0.678	612	3.729	**590**	**0**
LA05	593	**593**	**0**	**593**	**0**	**593**	**0**	**593**	**0**
LA06	926	**926**	**0**	**926**	**0**	**926**	**0**	**926**	**0**
LA07	890	**890**	**0**	**890**	**0**	920	3.371	**890**	**0**
LA08	863	**863**	**0**	**863**	**0**	**863**	**0**	**863**	**0**
LA09	951	**951**	**0**	**951**	**0**	**951**	**0**	**951**	**0**
LA10	958	**958**	**0**	**958**	**0**	**958**	**0**	**958**	**0**
LA11	1222	**1222**	**0**	**1222**	**0**	**1222**	**0**	**1222**	**0**
LA12	1039	**1039**	**0**	**1039**	**0**	**1039**	**0**	**1039**	**0**
LA13	1150	**1150**	**0**	**1150**	**0**	**1150**	**0**	**1150**	**0**
LA14	1292	**1292**	**0**	**1292**	**0**	**1292**	**0**	**1292**	**0**
LA15	1207	**1207**	**0**	1239	2.651	1281	6.131	**1207**	**0**
LA16	945	**945**	**0**	978	3.492	990	4.762	946	0.106
LA17	784	**784**	**0**	806	2.806	817	4.209	792	1.020
LA18	848	**848**	**0**	882	4.009	889	4.835	861	1.533
LA19	842	**842**	**0**	884	4.988	917	8.907	863	2.494
LA20	902	907	0.554	933	3.437	939	4.102	907	0.554
LA21	1046	1061	1.434	1150	9.943	1240	18.547	1085	3.728
LA22	927	935	0.863	1032	11.327	1092	17.799	937	1.079
LA23	1032	1032	0	1083	4.942	1157	12.112	1032	0
LA24	935	946	1.176	1047	11.979	1108	18.503	981	4.920
LA26	1218	**1218**	**0**	1363	11.905	1466	20.361	**1218**	**0**
LA30	1355	**1355**	**0**	1502	10.849	1615	19.188	**1355**	**0**
LA31	1784	**1784**	**0**	1814	1.682	2048	14.798	**1784**	**0**
LA32	1850	**1850**	**0**	1934	4.541	2170	17.297	**1850**	**0**
LA33	1719	**1719**	**0**	1774	3.200	1927	12.100	**1719**	**0**
LA34	1721	**1721**	**0**	1826	6.101	2003	16.386	**1721**	**0**
LA35	1888	**1888**	**0**	1980	4.873	2104	11.441	**1888**	**0**
Average of RPE (ARPE%)			0.141		3.771		7.266		0.562

5.2 Comparison between DCSA and three new algorithms

Table 6 shows the experimental results of DCSA vs the experimental results of GWO [4], DWPA [5] and DGA [6]. The simulation results of GWO, DWPA and

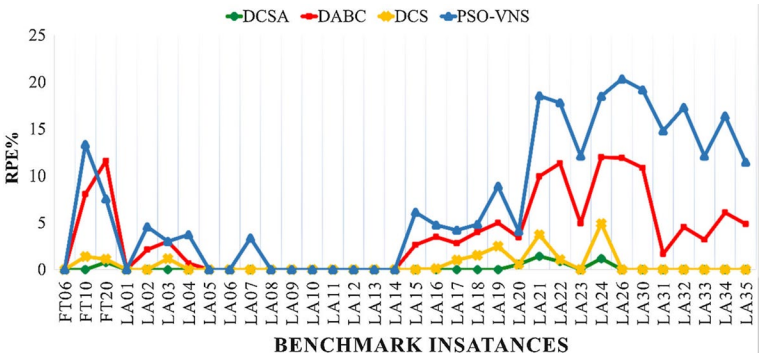


Fig. 6 Average of RPE for DCSA, PSO, BCO, and DCS

DGA were taken from their original papers. The symbol * indicates that the results of the tested algorithms were not found in the original papers.

The experimental results in Table 6 show that DCSA outperforms GWO, DWPA and DGA. DCSA achieved the BKS for 29 out of the 34 instances. GWO is the second-best performing algorithm with 23 out of the 34 instances, followed by DWPA which achieved the BKS for 15 instances and finally DGA is in the last position by achieving the BKS for 11 instances.

5.3 Comparison between the exploration methods used in DCSA

Table 7 shows a comparison between the exploration methods used in DCSA using 34 benchmark instances for the JSSP from the OR-Library. The results clearly show that DCSA combined with all its exploration methods is the best performing algorithm. It achieved the BKS for 33 out of the 34 tested instances.

However, the order of the best exploration methods that were used in DCSA individually is as follows: DCSA with EOBL (24 BKS), DCSA with VNS and Lévy flight (23 BKS), and finally DCSA with OBL (18 BKS). This means that DCSA with all its exploration methods is the best algorithm; however, the most effective exploration method is EOBL.

5.4 Computational time of DCSA compared to the other algorithms

Table 8 shows the average time in seconds over 10 runs to reach the BKS for DCSA, DABC, PSO-VNS and DCS. The results clearly show that DCSA is faster than the other algorithms for 27 instances out of the 34 instances. The results also show that DCS is the second fastest algorithm. Note that the symbol "_" indicates that an algorithm did not converge to the BKS for a JSSP instance within a reasonable time. DCSA is the only algorithm that converged in a reasonable time to the BKS for the JSSP instances. Although the RPE% of DCSA in Table 5 is the same as DABC, PSO-VNS, and DCS for FT06, LA01, LA05, LA06, LA08 to LA14, DCSA reaches

Table 6 Computational comparison between DCSA, GWO, DWPA, and GA

Instance	BKS	DCSA		GWO		DWPA		DGA	
		Best	RPE%	Best	RPE%	Best	RPE%	Best	RPE%
FT06	55	**55**	**0**	**55**	**0**	*	*	**55**	**0**
FT10	930	**930**	**0**	940	1.075	*	*	1009	8.495
FT20	1165	**1174**	**0.773**	1178	1.116	*	*	1263	8.412
LA01	666	**666**	**0**	**666**	**0**	**666**	**0**	666	0.000
LA02	655	**655**	**0**	**655**	**0**	**655**	**0.000**	663	1.221
LA03	597	**597**	**0**	**597**	**0**	614	2.848	603	1.005
LA04	590	**590**	**0**	**590**	**0**	598	1.356	604	2.373
LA05	593	**593**	**0**	**593**	**0**	**593**	**0**	**593**	**0**
LA06	926	**926**	**0**	**926**	**0**	**926**	**0**	**926**	**0**
LA07	890	**890**	**0**	**890**	**0**	**890**	**0**	**890**	**0**
LA08	863	**863**	**0**	**863**	**0**	**863**	**0**	**863**	**0**
LA09	951	**951**	**0**	**951**	**0**	**951**	**0**	**951**	**0**
LA10	958	**958**	**0**	**958**	**0**	**958**	**0**	**958**	**0**
LA11	1222	**1222**	**0**	**1222**	**0**	**1222**	**0**	**1222**	**0**
LA12	1039	**1039**	**0**	**1039**	**0**	**1039**	**0**	**1039**	**0**
LA13	1150	**1150**	**0**	**1150**	**0**	**1150**	**0**	**1150**	**0**
LA14	1292	**1292**	**0**	**1292**	**0**	**1292**	**0**	**1292**	**0**
LA15	1207	**1207**	**0**	**1207**	**0**	1273	5.468	1216	0.746
LA16	945	**945**	**0**	956	1.164	993	5.079	973	2.963
LA17	784	**784**	**0**	790	0.765	793	1.148	797	1.658
LA18	848	**848**	**0**	859	1.297	861	1.533	869	2.476
LA19	842	**842**	**0**	845	0.356	888	5.463	883	4.869
LA20	902	907	0.554	937	3.880	934	3.548	918	1.774
LA21	1046	1061	1.434	1090	4.207	1105	5.641	1159	10.803
LA22	927	935	0.863	970	4.639	989	6.688	1029	11.003
LA23	1032	1032	0	**1032**	**0**	1051	1.841	1091	5.717
LA24	935	946	1.176	982	5.027	988	5.668	1032	10.374
LA26	1218	**1218**	**0**	1239	1.724	1303	6.979	1354	11.166
LA30	1355	**1355**	**0**	**1355**	**0**	1389	2.509	1481	9.299
LA31	1784	**1784**	**0**	**1784**	**0**	**1784**	**0**	1842	3.251
LA32	1850	**1850**	**0**	**1850**	**0**	**1850**	**0**	1951	5.459
LA33	1719	**1719**	**0**	**1719**	**0**	**1719**	**0**	1789	4.072
LA34	1721	**1721**	**0**	**1721**	**0**	1788	3.893	1850	7.496
LA35	1888	**1888**	**0**	**1888**	**0**	1947	3.125	1962	3.919
Average of RPE (ARPE%)			0.141		3.771		0.789		2.025

the BKS in Table 8 for these JSSP instances in much less CPU time than the other algorithms. Another issue, the RPE% of DCSA and DCS in Table 5 are the same for LA26 to LA35. But DCS is faster than DCSA, which is maybe because DCSA requires in each one of its iterations more computations than DCS.

Table 7 Comparison between the exploration methods used in DCSA

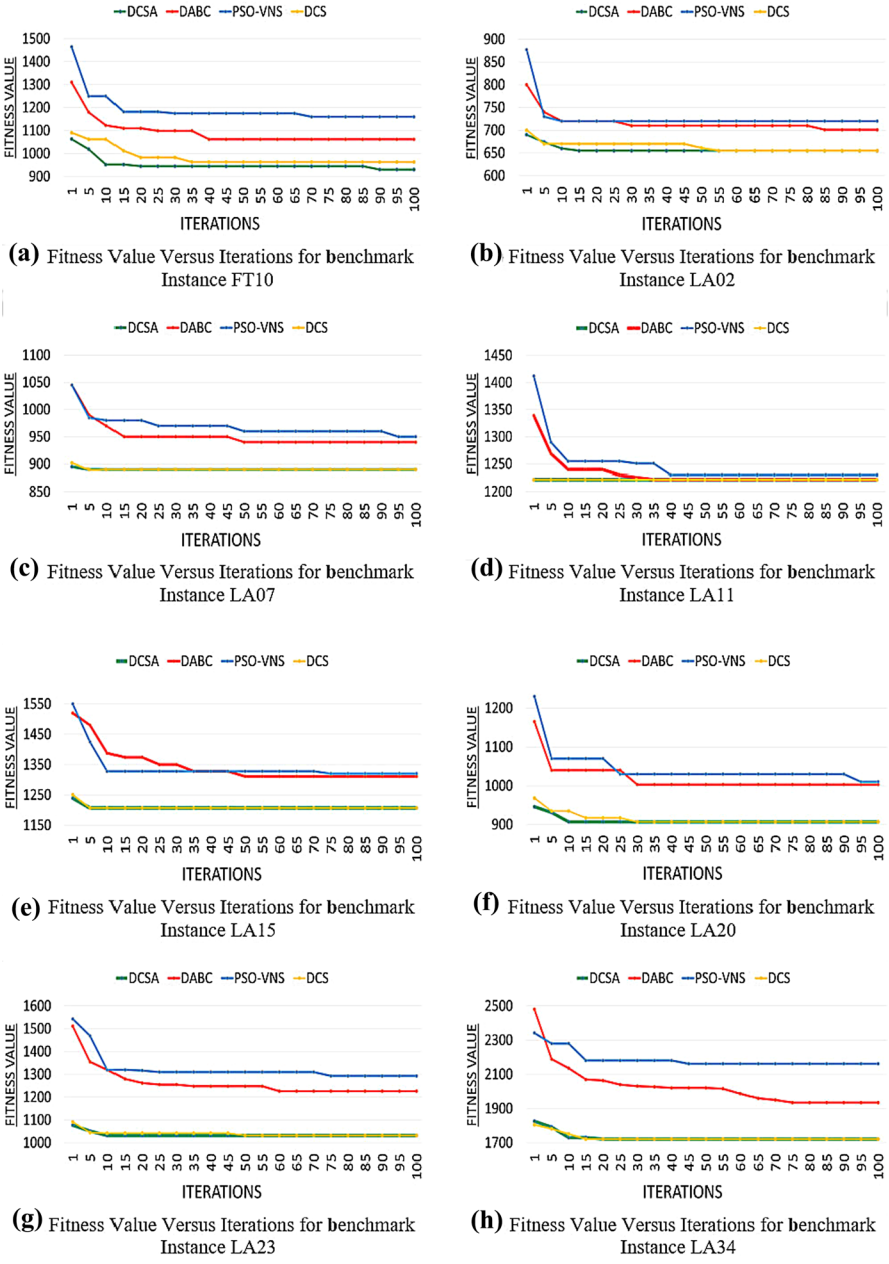
Instance	BKS	DCSA with OBL		DCSA with VNS and Lévy flight		DCSA with EOBL		DCSA with all exploration methods	
		Best	RPE%	Best	RPE%	Best	RPE%	Best	RPE%
FT06	55	**55**	**0**	**55**	**0**	**55**	**0**	**55**	**0**
FT10	930	978	5.161	942	1.290	966	3.871	**930**	**0**
FT20	1165	1217	4.464	1191	2.232	1193	2.403	**1174**	**0.773**
LA01	666	**666**	**0**	**666**	**0**	**666**	**0**	**666**	**0**
LA02	655	**655**	**0**	**655**	**0**	**655**	**0**	**655**	**0**
LA03	597	601	0.670	601	0.670	602	0.838	**597**	**0**
LA04	590	**590**	**0**	**590**	**0**	**590**	**0**	**590**	**0**
LA05	593	598	0.843	**593**	**0**	**593**	**0**	**593**	**0**
LA06	**926**	**926**	**0**	**926**	**0**	**926**	**0**	**926**	**0**
LA07	**890**	**890**	**0**	**890**	**0**	**890**	**0**	**890**	**0**
LA08	**863**	**863**	**0**	**863**	**0**	**863**	**0**	**863**	**0**
LA09	**951**	**951**	**0**	**951**	**0**	**951**	**0**	**951**	**0**
LA10	**958**	**958**	**0**	**958**	**0**	**958**	**0**	**958**	**0**
LA11	**1222**	**1222**	**0**	**1222**	**0**	**1222**	**0**	**1222**	**0**
LA12	**1039**	**1039**	**0**	**1039**	**0**	**1039**	**0**	**1039**	**0**
LA13	**1150**	**1150**	**0**	**1150**	**0**	**1150**	**0**	**1150**	**0**
LA14	1292	1319	2.090	1298	0.464	1295	0.232	**1292**	**0**
LA15	1207	1221	1.160	1213	0.497	1217	0.829	**1207**	**0**
LA16	945	963	1.905	946	0.106	946	0.106	**945**	**0**
LA17	784	794	1.276	791	0.893	792	1.020	**784**	**0**
LA18	848	853	0.590	**848**	**0**	**848**	**0**	**848**	**0**
LA19	842	873	3.682	**842**	**0**	**842**	**0**	**842**	**0**
LA20	902	921	2.106	**907**	**0.554**	**907**	**0.554**	**907**	**0.554**
LA21	1046	1112	6.310	1075	2.772	1068	2.103	**1061**	**1.434**
LA22	927	947	2.157	937	1.079	939	1.294	**935**	**0.863**
LA23	1032	**1032**	**0**	**1032**	**0**	**1032**	**0**	**1032**	**0**
LA24	935	961	2.781	949	1.497	**943**	**0.856**	946	1.176
LA26	1218	1262	3.612	1221	0.246	1226	0.657	**1218**	**0**
LA30	1355	1390	2.583	**1355**	**0**	**1355**	**0**	**1355**	**0**
LA31	1784	**1784**	**0**	**1784**	**0**	**1784**	**0**	**1784**	**0**
LA32	1850	**1850**	**0**	**1850**	**0**	**1850**	**0**	**1850**	**0**
LA33	1719	**1719**	**0**	**1719**	**0**	**1719**	**0**	**1719**	**0**
LA34	1721	**1721**	**0**	**1721**	**0**	**1721**	**0**	**1721**	**0**
LA35	1888	**1888**	**0**	**1888**	**0**	**1888**	**0**	**1888**	**0**
Average of RPE (ARPE%)			1.217		0.362		0.434		0.141

Table 8 Comparison of CPU time between DCSA, DABC, PSO-VNS, and DCS

Instance	DCSA	DABC	PSO-VNS	DCS
FT06	**1**	5.6	3.2	**1**
FT10	**459.2**	–	–	–
FT20	**540.4**	–	–	–
LA01	**1**	4.8	9.8	**1**
LA02	**1.8**	–	–	4.2
LA03	**45.4**	–	–	230
LA04	**18.6**	–	–	54.2
LA05	**1**	1.6	1	**1**
LA06	**1**	3.2	4.8	**1**
LA07	**1**	39.6	466.2	**1**
LA08	**1**	21.8	40.4	**1**
LA09	**1**	2.8	6.6	**1**
LA10	**1**	1.8	1.6	**1**
LA11	**1**	6.8	30.2	**1**
LA12	**1**	3	11.4	**1**
LA13	**1**	5	33.2	**1**
LA14	**1**	1.6	1.6	**1**
LA15	**5.4**	–	–	3.2
LA16	**114.6**	–	–	–
LA17	**47.4**	–	–	–
LA18	**127.4**	–	–	–
LA19	**110.4**	–	–	–
LA20	**9.4**	–	–	23.8
LA21	**1466.8**	–	–	–
LA22	**2510.8**	–	–	4305
LA23	**37.2**	–	–	112.2
LA24	**2103.2**	–	–	–
LA26	3810.6	–	–	**2029.6**
LA30	1867.4	–	–	**358.2**
LA31	160.8	–	–	**51**
LA32	291	–	–	**57.2**
LA33	262	–	–	**193.6**
LA34	1746.2	–	–	**392.4**
LA35	653.8	–	–	**149.4**

5.5 Convergence speed of DCSA compared to the other algorithms

In this section, we compared the convergence speed of the algorithms over the first 100 iterations. Figure 7 illustrates the convergence behavior of the algorithms using eight benchmark instances with different sizes (FT10, LA02, LA07, LA11, LA15, LA20, LA23, and LA34). As shown in Figs. 6, DCSA converges faster than the other algorithms to the BKS of each benchmark instances. However, DCS exhibits



(a) Fitness Value Versus Iterations for benchmark Instance FT10

(b) Fitness Value Versus Iterations for benchmark Instance LA02

(c) Fitness Value Versus Iterations for benchmark Instance LA07

(d) Fitness Value Versus Iterations for benchmark Instance LA11

(e) Fitness Value Versus Iterations for benchmark Instance LA15

(f) Fitness Value Versus Iterations for benchmark Instance LA20

(g) Fitness Value Versus Iterations for benchmark Instance LA23

(h) Fitness Value Versus Iterations for benchmark Instance LA34

Fig. 7 Comparison of convergence speed between DCSA, DABC, PSO-VNS, and DCS

very competitive behavior compared to DCSA. The convergence behavior of the DCSA algorithm is possibly because it utilizes different exploration methods (SA, VNS with Lévy flight, OBL, and EOBL).

Table 9 Friedman test

Algorithm	Rank	Chi-square	*df*	Asymp. Sig
DCSA	1.69	64.662	3	0.000
DABC	2.85			
PSO-VNS	3.47			
DCS	1.99			

Table 10 Friedman test between DCSA and new algorithms

Algorithm	Rank	Chi-square	*df*	Asymp. Sig
DCSA	62	24.9794	3	0.00002
GWO	75			
DWPA	90.5			
GA	112.5			

5.6 Friedman test

In this section, the Friedman test was applied to the experimental results in Tables 5 and 6 to validate the significance of their results using the level of significance of 0.05. Table 9 shows the results of the Friedman test that was applied to the simulation results in Table 5, while Table 10 shows the results of Friedman test that was applied to the simulation results in Table 6.

Table 9 shows that the DCSA achieved the first rank, followed by DCS. Note that the Asymp. Sig (short for asymptotic significance) is zero. Table 10 also shows that the DCSA achieved the first rank, followed by GWO. Note that the Asymp. Sig is zero. The overall results in Tables 9 and 10 indicate that DCSA is an efficient and reliable algorithm for solving the JSSP problem.

5.7 Discussion of the overall experimental results and limitations

The superior performance of DCSA in solving the 34 JSSP instances (Sect. 5.1 and 5.2) is possibly because it integrates SA into CS and uses different exploration methods (VNS with Lévy flight, OBL and EOBL). These methods increase the ability of DCSA to find better solutions in the search space. In Sect. 5.3, the simulation of the exploration methods used in DCSA using 34 benchmark instances for the JSSP from the OR-Library clearly shows that DCSA combined with all its exploration methods is the best-performing algorithm; however, the most effective exploration method is EOBL.

The convergence analysis in Sect. 5.5 indicates that DCSA converges faster than the other algorithms to the BKS of each of the 34 benchmark instances. However, DCS exhibits a very competitive behavior compared to DCSA. This is possibly because both algorithms are based on CS and use the Lévy-flight method to explore the search space. Besides, the results of the Friedman test (Sect. 5.6) clearly indicate

that the performance of DCSA is stable and reliable for the 34 tested instances of JSSP compared to the other tested algorithms.

The computational complexity of DCSA is $O(m.n)$, where m is the maximum number of iterations and n is the number of candidate solutions. This complexity is the same as the computational complexity of the original CS algorithm. However, the comparison of computational times of the tested algorithms in Sect. 5.2 clearly shows that DCSA is faster than the other algorithms for all the 34 tested JSSP instances.

6 Conclusions

The current paper presented an optimization-based scheduling algorithm by the name discrete cuckoo search and simulated annealing (DCSA) algorithm for solving the job shop scheduling problem (JSSP). DCSA is a discrete variation of the cuckoo search and simulated annealing (CSA) algorithm. DCSA uses the OBL method to increase the diversity of the initial population, and the EOBL method at the end of its improvement loop to explore the surrounding of the best candidate solutions. Besides, it uses the SPV method to convert the continuous candidate solutions generated in its initialization step into discrete ones, and then it utilizes the VNS and Lévy-flight methods to explore the search space.

DCSA algorithm was experimentally evaluated and compared DCSA to six popular optimization-based scheduling algorithms (DABC, PSO-VNS, DCS, GWO, DWPA, DGA) using 34 instances of two benchmark sets from the OR-Library: Fisher and Thompson (3 instances), and Lawrence (31 instances). The experimental results show that the DCSA algorithm outperformed the other algorithms for 29 out of 34 instances with less computational time compared to the other tested algorithms. This is possibly because DCSA uses the SA and VNS for the exploration of the search space, and OBL and EBOL to jump out of local optima.

In the future, we will incorporate a population structured model called the island model [25, 53–55] into DCSA to further improve its population diversity, performance, and convergence behavior. It will be also interesting to incorporate the SA algorithm in new innovative algorithms such as the Spotted Hyena Optimizer [56], Harris Hawk Optimization [57] and Farmland Fertility [58], Human Mental Search [59] algorithms. Finally, we will apply CSA as a selection method for actions in the Q-learning algorithm as described in [60–62].

References

1. Yin M, Li X, Zhou J (2011) An efficient job shop scheduling algorithm based on artificial bee colony. Sci Res Essays 6:2578–2596
2. Tasgetiren MF, Sevkli M, Liang Y-C, Yenisey MM (2006) A particle swarm optimization and differential evolution algorithms for job shop scheduling problem. Int J Oper Res 3:120–135
3. Ouaarab A, Ahiod B, Yang X-S (2015) Discrete cuckoo search applied to job shop scheduling problem. In: Yang X-S (ed) Recent advances in swarm intelligence and evolutionary computation. Springer, Cham, pp 121–137. https://doi.org/10.1007/978-3-319-13826-8_7

4. Jiang T, Zhang C (2018) Application of grey wolf optimization for solving combinatorial problems: job shop and flexible job shop scheduling cases. IEEE Access 6:26231–26240. https://doi.org/10.1109/ACCESS.2018.2833552
5. Wang F, Tian Y, Wang X (2019) A discrete wolf pack algorithm for job shop scheduling problem. In: 2019 5th International Conference on Control, Automation and Robotics ICCAR 2019. pp 581–585. https://doi.org/10.1109/ICCAR.2019.8813444
6. Kalshetty YR, Adamuthe AC, Kumar SP (2020) Genetic algorithms with feasible operators for solving job shop scheduling problem. J Sci Res 64:310–321. https://doi.org/10.37398/jsr.2020.640157
7. Asadzadeh L (2016) A parallel artificial bee colony algorithm for the job shop scheduling problem with a dynamic migration strategy. Comput Ind Eng 102:359–367. https://doi.org/10.1016/j.cie.2016.06.025
8. Chakraborty S, Bhowmik S (2013) Job shop scheduling using simulated annealing. In: First International Conference on Computation and Communication Advancement. pp 69–73
9. Chaouch I, Driss OB, Ghedira K (2017) A modified ant colony optimization algorithm for the distributed job shop scheduling problem. Procedia Comput Sci 112:296–305. https://doi.org/10.1016/j.procs.2017.08.267
10. Zaher H, El-sherbieny M (2017) Bat algorithm for job shop scheduling problem. J Multidiscip Eng Sci Technol 4:6758–6763
11. Banharnsakun A, Sirinaovakul B, Achalakul T (2012) Job shop scheduling with the best-so-far ABC. Eng Appl Artif Intell 25:583–593
12. Asadzadeh L (2015) A local search genetic algorithm for the job shop scheduling problem with intelligent agents. Comput Ind Eng 85:376–383. https://doi.org/10.1016/j.cie.2015.04.006
13. Khadwilard A, Pongcharoen P (2013) Job shop scheduling optimisation using harmony search algorithm. Naresuan Univ J Sci Technol pp 8–14
14. Rameshkumar K, Rajendran C (2018) A novel discrete PSO algorithm for solving job shop scheduling problem to minimize makespan. IOP Conf Ser Mater Sci Eng. https://doi.org/10.1088/1757-899X/310/1/012143
15. Udaiyakumar KC, Chandrasekaran M (2014) Application of firefly algorithm in job shop scheduling problem for minimization of makespan. Procedia Eng 97:1798–1807. https://doi.org/10.1016/j.proeng.2014.12.333
16. Wang H, Wang W, Sun H et al (2017) A new cuckoo search algorithm with hybrid strategies for flow shop scheduling problems. Soft Comput 21:4297–4307. https://doi.org/10.1007/s00500-016-2062-9
17. Alawad NA, Abed-alguni BH (2021) Discrete island-based cuckoo search with highly disruptive polynomial mutation and opposition-based learning strategy for scheduling of workflow applications in cloud environments. Arab J Sci Eng 46:3213–3233. https://doi.org/10.1007/s13369-020-05141-x
18. Alkhateeb F, Abed-Alguni BH (2017) A hybrid cuckoo search and simulated annealing algorithm. Intell Syst 28(4):683–698
19. Mousavirad SJ, Rahnamayan S (2020) One-array differential evolution algorithm with a novel replacement strategy for numerical optimization. In: 2020 IEEE International Conference on Systems, Man, and Cybernetics (SMC). IEEE, pp 2514–2519
20. Mousavirad SJ, Rahnamayan S (2019) Differential evolution algorithm based on a competition scheme. In: 2019 14th International Conference on Computer Science & Education (ICCSE). IEEE, pp 929–934
21. Mousavirad SJ, Rahnamayan S (2020) A novel center-based differential evolution algorithm. In: 2020 IEEE Congress on Evolutionary Computation (CEC). IEEE, pp 1–8
22. Abed-Alguni BH, Klaib AF (2020) Hybrid whale optimisation and β-hill climbing algorithm for continuous optimisation problems. Int J Comput Sci Math. https://doi.org/10.1504/IJCSM.2020.112650
23. Zhu J, Shao ZH, Chen C (2019) An improved whale optimization algorithm for job-shop scheduling based on quantum computing. Int J Simul Model 18:521–530. https://doi.org/10.2507/ijsimm18(3)co13
24. Liu M, Yao X, Li Y (2020) Hybrid whale optimization algorithm enhanced with Lévy flight and differential evolution for job shop scheduling problems. Appl Soft Comput J 87:105954. https://doi.org/10.1016/j.asoc.2019.105954
25. Abed-Alguni BH, Klaib AF, Nahar KMO (2019) Island-based whale optimisation algorithm for continuous optimisation problems. Int J Reason Intell Syst 11:319–329
26. Alawad NA, Abed-alguni BH (2021) Discrete Jaya with refraction learning and three mutation methods for the permutation flow shop scheduling problem. J Supercomput pp 1–22

27. Alkhateeb F, Abed-Alguni BH (2019) A hybrid cuckoo search and simulated annealing algorithm. J Intell Syst 28:683–698
28. Kirkpatrick S, Gelatt CD, Vecchi MP (1983) Optimization by simulated annealing optimization by simulated annealing. Science 80(220):671–680
29. Yang XS, Deb S (2009) Cuckoo search via Lévy flights. In: 2009 World Congress on Nature and Biologically Inspired Computing, NABIC 2009—Proceedings. pp 210–214
30. Kennedy J, Eberhart R (1995) Particle swarm optimization. In: Proceedings of ICNN'95-International Conference on Neural Networks. IEEE, pp 1942–1948
31. Yang X, Gandomi AH (2012) Bat algorithm: a novel approach for global engineering optimization. Eng Comput 29:464–483
32. Abed-alguni BH, Alkhateeb F (2017) Novel selection schemes for cuckoo search. Arab J Sci Eng. https://doi.org/10.1007/s13369-017-2663-3
33. Mladenović N, Hansen P (1997) Variable neighborhood search. Comput Oper Res 24:1097–1100
34. Huang KW, Girsang AS, Wu ZX, Chuang YW (2019) A hybrid crow search algorithm for solving permutation flow shop scheduling problems. Appl Sci. https://doi.org/10.3390/app9071353
35. Fisher H, Thompson GL (1963) Probabilistic learning combinations of local job-shop scheduling rules. Ind Sched pp 225–251
36. Lawrence S (1984) Resource constrained project scheduling: an experimental investigation of heuristic scheduling techniques (Supplement). Grad Sch Ind Adm Carnegie-Mellon Univ
37. Nazif H (2015) Solving job shop scheduling problem using an ant colony algorithm. J Asian Sci Res 5:261–268. https://doi.org/10.18488/journal.2/2015.5.5/2.5.261.268
38. Singh S, Singh KP (2015) Cuckoo search optimization for job shop scheduling problem. Adv Intell Syst Comput 335:99–111. https://doi.org/10.1007/978-81-322-2217-0_9
39. Zaman HRR, Gharehchopogh FS (2021) An improved particle swarm optimization with backtracking search optimization algorithm for solving continuous optimization problems. Eng Comput pp 1–35
40. Jiang T (2018) A hybrid grey wolf optimization for job shop scheduling problem. Int J Comput Intell Appl 17:1–12. https://doi.org/10.1142/S1469026818500165
41. Gao K, Yang F, Zhou M et al (2018) Flexible job-shop rescheduling for new job insertion by using discrete Jaya algorithm. IEEE Trans Cybern 49:1944–1955
42. Abed-Alguni BH, Paul DJ (2020) Hybridizing the cuckoo search algorithm with different mutation operators for numerical optimization problems. J Intell Syst. https://doi.org/10.1515/jisys-2018-0331
43. Fernandez-Viagas V, Molina-Pariente JM, Framinan JM (2020) Generalised accelerations for insertion-based heuristics in permutation flowshop scheduling. Eur J Oper Res 282:858–872
44. Tasgetiren MF, Sevkli M, Liang YC, Gencyilmaz G (2004) Particle swarm optimization algorithm for permutation flowshop sequencing problem. In: Lect Notes Comput Sci (including Subser Lect Notes Artif Intell Lect Notes Bioinformatics) 3172 LNCS. pp 382–389. https://doi.org/10.1007/978-3-540-28646-2_38
45. Gharehchopogh FS, Maleki I, Dizaji ZA (2021) Chaotic vortex search algorithm: metaheuristic algorithm for feature selection. Evol Intell pp 1–32
46. Gupta S, Deep K (2019) A hybrid self-adaptive sine cosine algorithm with opposition based learning. Expert Syst Appl 119:210–230
47. Mahdavi S, Rahnamayan S, Deb K (2018) Opposition based learning: a literature review. Swarm Evol Comput 39:1–23. https://doi.org/10.1016/j.swevo.2017.09.010
48. Zhou X, Wu Z, Wang H et al (2013) Elite opposition-based particle swarm optimization. Acta Electron Sin 41:1647–1652
49. Singh MK (2013) Evaluating levy flight parameters for random searches in a 2D space
50. Wang K, Ma WQ, Luo H, Qin H (2016) Coordinated scheduling of production and transportation in a two-stage assembly flowshop. Int J Prod Res 54:6891–6911
51. Beasley JE (1990) OR-Library: distributing test problems by electronic mail. J Oper Res Soc 41:1069–1072
52. Bouzidi A, Riffi ME (2014) Cat swarm optimization to solve job shop scheduling problem. In: 2014 Third IEEE International Colloquium in Information Science and Technology (CIST). IEEE, pp 202–205
53. Abed-alguni BH, Barhoush M (2018) Distributed grey wolf optimizer for numerical optimization problems. Jordanian J Comput Inf Technol 4:130–149
54. Abed-alguni BH, Alawad NA (2021) Distributed grey wolf optimizer for scheduling of workflow applications in cloud environments. Appl Soft Comput 102:107113

55. Abed-alguni BH (2019) Island-based cuckoo search with highly disruptive polynomial mutation. Int J Artif Intell 17:57–82
56. Ghafori S, Gharehchopogh FS (2021) Advances in spotted hyena optimizer: a comprehensive survey. Arch Comput Methods Eng pp 1–22
57. Gharehchopogh FS, Abdollahzadeh B (2021) An efficient harris hawk optimization algorithm for solving the travelling salesman problem. Cluster Comput pp 1–25
58. Gharehchopogh FS, Farnad B, Alizadeh A (2021) A farmland fertility algorithm for solving constrained engineering problems. Concurr Comput Pract Exp p e6310
59. Mousavirad SJ, Schaefer G, Esmaeili L, Korovin I (2020) On improvements of the human mental search algorithm for global optimisation. In: 2020 IEEE Congress on Evolutionary Computation (CEC). IEEE, pp 1–8
60. Abed-alguni BH (2018) Action-selection method for reinforcement learning based on cuckoo search algorithm. Arab J Sci Eng 43:6771–6785
61. Abed-alguni BH (2017) Bat Q-learning algorithm. Jordan J Comput Inf Technol 3:56–77
62. Abed-alguni BH, Ottom MA (2018) Double delayed Q-learning. Int J Artif Intell 16:41–59

Authors and Affiliations

Faisal Alkhateeb[1] · Bilal H. Abed-alguni[1] · Mohammad Hani Al-rousan[1]

✉ Bilal H. Abed-alguni
bilal.h@yu.edu.jo

Faisal Alkhateeb
alkhateebf@yu.edu.jo

Mohammad Hani Al-rousan
malrousan77@gmail.com

[1] Department of Computer Sciences, Yarmouk University, Irbid, Jordan