

A Block-Guided Artificial Bee Colony Algorithm for Distributed Assembly Flowshop Scheduling with Sequence-Dependent Setup Times

Ji-Feng Xu

Zhejiang University of Finance and Economics

Hong-Bo Song

hbsong@zjsru.edu.cn

Zhejiang Shuren University

Article

Keywords:

Posted Date: October 30th, 2026

DOI: <https://doi.org/10.21203/rs.3.rs-11546208/v1>

License:   This work is licensed under a Creative Commons Attribution 4.0 International License.

[Read Full License](#)

Additional Declarations: No competing interests reported.

A Block-Guided Artificial Bee Colony Algorithm for Distributed Assembly Flowshop Scheduling with Sequence-Dependent Setup Times

Ji-Feng Xu¹ and Hong-Bo Song^{2*}

^{1*}School of Information Technology and Artificial Intelligence, Zhejiang University of Finance & Economics, Xueyuan Street, Hangzhou, 310018, Zhejiang, China.

^{2*}Department of Artificial Intelligence, Zhejiang Shuren University, Shuren Street, Hangzhou, 310015, Zhejiang, China.

*Corresponding author(s). E-mail(s): hbsong@zjsru.edu.cn;
Contributing authors: xjf@zufe.edu.cn;

Abstract

The Distributed Assembly Permutation Flowshop Scheduling Problem with Sequence-Dependent Setup Times (DAPFSP-SDST) arises in many modern manufacturing supply chains. This paper proposes a Block-Guided Artificial Bee Colony (BG-ABC) algorithm for this problem with the aim of minimizing the makespan. An idle-aware block compression mechanism is developed to lock adjacent product pairs with minimal setup time and zero assembly idle time into atomic blocks, reducing the search space for global exploration. To further enhance search adaptability, a Sliding Window Upper Confidence Bound 1 (SW-UCB1) scheme is embedded in the employed bee phase to dynamically select neighborhood operators based on their recent performance. In the onlooker bee phase, local search is restricted to the Critical Job Subsequence so that computational effort is concentrated on the bottleneck region of the schedule. Moreover, a refinement-and-reconstruction mechanism is proposed in the scout bee phase to counteract the stagnation risk induced by block compression. Finally, extensive experiments on benchmark instances demonstrate that BG-ABC significantly outperforms state-of-the-art algorithms, verifying its effectiveness in solving DAPFSP-SDST.

Keywords: Distributed Assembly Scheduling; Artificial Bee Colony; Idle-Aware Block Compression; Adaptive Neighborhood Selection

1 Introduction

1.1 Background

With the deepening of economic globalization and industrial specialization, distributed assembly manufacturing has become a mainstream production strategy in industries such as automotive, semiconductors, and furniture [1, 2]. In this paradigm, jobs are processed across multiple geographically dispersed factories and subsequently transported to a centralized assembly facility, where they are assembled into final products according to predefined product structures. This model offers several well-recognized advantages, such as higher product quality, decreased management risks, and the ability to produce a large variety of final products at a controlled cost [3, 4]. In the context of supply chain management, this distributed assembly architecture constitutes a multi-stage supply chain network, where coordinating geographically dispersed factories is critical for fulfilling customer orders efficiently.

Despite these advantages, scheduling such distributed assembly systems is far from trivial, as it requires tight coordination of production and assembly decisions within a multi-stage supply chain. In practice, each factory in these systems typically operates as a permutation flowshop, where jobs are processed through a series of machines in a fixed order. This gives rise to the Distributed Assembly Permutation Flowshop Scheduling Problem (DAPFSP), which involves three sets of strongly coupled decisions: assigning jobs to factories, sequencing jobs within each factory, and determining the assembly order of products [5]. Owing to its practical relevance and computational intractability, the DAPFSP has attracted extensive attention from researchers, and a variety of effective constructive heuristics and metaheuristics have been developed [6].

In practical manufacturing, setup operations—such as machine cleaning, tool replacement, and fixture adjustment—are usually required both before processing a job and assembling a product, and the corresponding time is referred to as the setup time. Setup time commonly depends on the pair of consecutive jobs or products on a machine, and this type of setup time is known as the Sequence-Dependent Setup Time (SDST). Since SDSTs couple adjacent jobs and products in a nonseparable manner, any decision on job or product sequencing immediately affects the setup times of the preceding and succeeding operations, thereby making the scheduling problem substantially more difficult to solve. Incorporating SDSTs into both the processing and assembly stages of the DAPFSP gives rise to the DAPFSP-SDST [7], which is a strongly NP-hard problem to be addressed in this paper.

In recent years, a number of constructive heuristics, metaheuristics, and hyperheuristics have been developed for the DAPFSP-SDST. Among them, Song et al. [8] revealed a fundamental property of the problem: minimizing the makespan is equivalent to minimizing the sum of assembly setup times and assembly idle times. They proposed a Two-Stage Heuristic Algorithm (TSHA) that first constructs promising product sequences and then optimizes these two components using neighborhood descent algorithms. Although TSHA achieves both remarkable computational efficiency and acceptable performance, its solution quality is inherently constrained by the single-descent nature of its local search. To overcome this limitation, metaheuristics

offer a promising alternative, as their population-based search and flexible neighborhood structures can explore a much larger solution space than a single-descent heuristic. However, the solution space of the DAPFSP-SDST can be viewed as $d!$ regions, each defined by a distinct product sequence, where d is the number of products. When d is large, even identifying which regions are worth exploring becomes a challenging task, let alone optimizing the job sequences within them.

Interestingly, observations on high-quality solutions reveal that a large number of adjacent product pairs are both SDST-minimal and assembled with zero idle time. Such pairs achieve the minimum possible contribution to the objective and are therefore expected to appear frequently in elite solutions. This suggests that locking these pairs into blocks during the search can substantially reduce the effective search space and improve search efficiency. However, such compression also carries a risk of premature convergence, as locked blocks may prevent the algorithm from exploring alternative structural configurations. This is precisely where the Artificial Bee Colony (ABC) algorithm offers a natural advantage. Its scout bee mechanism can periodically abandon stagnated solutions and reconstruct the search structure, thereby compensating for the risk introduced by compression. Moreover, at the algorithmic level, balancing search depth and computational cost during exploitation remains a general challenge. This involves two critical aspects: dynamically selecting suitable neighborhood operators as their effectiveness changes during the search, and precisely confining the local search to the bottleneck region to avoid wasting evaluations on non-critical parts of the schedule.

Motivated by the above observations and challenges, we propose a Block-Guided Artificial Bee Colony (BG-ABC) algorithm that embeds problem-specific structural insights into the ABC framework. The above challenges are addressed through four coordinated designs: an idle-aware block compression mechanism that reduces the search space, a Sliding Window Upper Confidence Bound 1 (SW-UCB1) scheme for adaptive operator selection, a critical-job-subsequence-based local search with two-level evaluation for focused exploitation, and a scout bee phase that dissolves locked blocks to escape local optima. The main contributions are summarized as follows:

- (1) An idle-aware block compression mechanism is proposed to lock adjacent product pairs that are both SDST-minimal and assembled with zero idle time into atomic blocks. Since such pairs frequently appear in elite solutions, this mechanism reduces the dimensionality of the search space while preserving promising structural patterns.
- (2) A SW-UCB1 scheme is embedded into the employed bee phase for adaptive operator selection. By dynamically adjusting the selection probabilities based on the recent rewards of each operator, SW-UCB1 enables the algorithm to adapt its search strategy as the block structure evolves.
- (3) In the onlooker bee phase, local search is restricted to the Critical Job Subsequence (CJS), and a two-level evaluation strategy is adopted to reduce the cost of evaluating candidate moves. This focuses computational effort on the bottleneck region of the schedule and significantly reduces the number of redundant evaluations.
- (4) To prevent the premature convergence induced by block compression, a refinement-and-reconstruction mechanism is designed in the scout bee phase. When an

individual stagnates, a targeted local refinement is first attempted; if no improvement is found, the existing blocks are dissolved and the product sequence is reconstructed to restore structural diversity.

In summary, by addressing the structural bottlenecks in distributed assembly scheduling, the proposed BG-ABC algorithm provides a practical and robust intelligent optimization tool for improving operational efficiency and decision-making within multi-stage supply chain environments. The remainder of this paper is organized as follows. Section 2 reviews the related literature. Section 3 introduces the notations. Section 4 describes the DAPFSP-SDST, solution encoding and decoding. Section 5 presents the proposed BG-ABC algorithm in detail. Section 6 reports the experimental results and statistical analyses. Finally, Section 7 concludes the paper and discusses future research directions.

2 Literature Review

2.1 The Distributed Assembly Permutation Flowshop Scheduling Problem

The DAPFSP with makespan criterion was formally introduced by Hatami et al. [5], who developed a mixed integer linear programming model, constructive heuristics, and a variable neighborhood descent (VND) algorithm. Since then, a collection of metaheuristics and hyper-heuristics has been proposed to improve solution quality and computational efficiency in distributed manufacturing environments. Wang and Wang [9] proposed an estimation of distribution algorithms-based memetic algorithm (EDAMA), which incorporates a selective-enhancing sampling mechanism and a critical-path-based local search to balance exploration and exploitation. Zhang et al. [10] introduced MCEDA, a matrix-cube-based EDA that uses a three-dimensional probabilistic model to preserve superior block and order information from elite solutions. Song et al. [11] proposed a hyper-heuristic-based memetic algorithm (HHMA) that uses an EDA-based hyper-heuristic for global search and a critical-products-based referenced local search for exploitation. Other contributions include a biased-randomized iterated local search by Ferone et al. [12], a hybrid biogeography-based optimization (BBO) by Lin and Zhang [13], a backtracking search hyper-heuristic by Lin et al. [14], a social spider optimization algorithm by Zhang et al. [15], and a single seekers society algorithm by Hamzadayi et al. [16].

Beyond makespan minimization, alternative objectives have also been considered for the DAPFSP. For the total flowtime criterion, Sang et al. [17] proposed effective discrete invasive weed optimization variants with a two-level solution representation and problem-specific local search operators for product and job sequences. Huang et al. [18] developed a groupthink-based improved iterated greedy algorithm (gIGA), which combines a Pal-NEH initialization, a destruction-and-construction scheme, and separate local searches to enhance solution quality. Qiu et al. [19] designed three evolutionary algorithms, in which Q-learning and Sarsa adaptively select among local search operators. For energy-efficient scheduling, Zhang et al. [20] proposed an MCEDA

that incorporates a machine speed adjustment strategy to reduce energy consumption without sacrificing production efficiency. Yu et al. [21] embedded Q-learning into metaheuristics, enabling adaptive selection of local search operators to dynamically balance energy consumption with other criteria. Cheng et al. [22] developed a BBO algorithm featuring a dual-population heuristic initialization and hybrid migration operators. Sun et al. [23] proposed a multi-objective memetic algorithm that explicitly generates time-dominant and energy-dominant product sub-sequences to balance makespan and energy consumption.

To accommodate practical production constraints, various constrained variants of the DAPFSP have also been investigated. These include the blocking, no-wait, and no-idle flowshop settings, for which Yang and Li [24] proposed a knowledge-driven constructive heuristic, Zhao et al. [25] designed an optimal block-knowledge-driven backtracking search algorithm, and Shao et al. [26] developed local search methods, respectively. Other variants include flexible assembly with batch delivery, studied by Yang and Xu [27]; a distributed assembly hybrid flowshop with transportation resource scheduling, modeled by Luo et al. [28]; and a distributed two-stage assembly flowshop, for which Deng et al. [29] presented a competitive memetic algorithm. Heterogeneous assembly settings have recently attracted attention, with Zhang et al. [30] developing a matheuristic-based self-learning approach and Zhao et al. [31] proposing a knowledge-based two-population optimization algorithm. In addition, Zhao et al. [32] addressed the integrated scheduling of distributed production, assembly, and delivery. For more literature on these DAPFSP variants and optimization objectives, see [1] and references therein.

The considered DAPFSP with sequence-dependent setup times (DAPFSP-SDST) was first studied by Hatami et al. [7], who proposed constructive heuristics, a VND, and iterated greedy (IG) algorithms with two neighborhood structures. Subsequently, Song and Lin [33] designed a genetic programming hyper-heuristic (GP-HH) to automatically select low-level heuristics from a heuristic set. Huang and Gu [34] developed a novel biogeography-based optimization algorithm (NBBO) with an effective initialization scheme and an SDST-specific local search operator. Yu et al. [35] proposed a double-learning-strategy-based evolutionary algorithm for the multi-objective DAPFSP-SDST, where Q-learning is used to guide the selection of machine speed adjustment strategies and neighborhood structures. However, the above results mainly focused on algorithm-level adaptation to the problem and have not fully discovered the essential properties of the DAPFSP-SDST. Song et al. [8] uncovered the structural characteristics of the DAPFSP-SDST and proposed a two-stage heuristic algorithm (TSHA) that efficiently solves the problem, achieving better results than existing approaches. Nevertheless, TSHA is essentially a heuristic algorithm with limited global exploration capability, and the design of a metaheuristic algorithm that can fully leverage the problem-specific structural information of the DAPFSP-SDST remains an open issue.

2.2 Scheduling with Sequence-Dependent Setup Times

Setup time is a critical factor in realistic production environments, and its sequence-dependent nature substantially increases the combinatorial complexity of scheduling

problems. Research on scheduling with setup times can be traced back to the mid-1960s. Early studies concentrated on classical shop environments, including single-machine, parallel-machine, flowshop, and job shop settings, where problem scales were small and exact methods predominated. As practical production environments became more complex and problem scales grew, the steady advancement of computing power and algorithmic techniques enabled a gradual shift toward heuristic and metaheuristic approaches capable of tackling larger and more realistic problems. A series of comprehensive surveys, from the first by Allahverdi et al. [36] to the most recent one [37], have reviewed this development.

For the permutation flowshop with SDST, Ruiz and Stützle [38] proposed an IG algorithm that combines a destruction–construction scheme with a local search, achieving strong performance for both makespan and total weighted tardiness objectives. In assembly flowshop environments with SDST, Navaei et al. [39] considered non-identical assembly machines and developed hybrid metaheuristics integrating simulated annealing and tabu search to minimize the sum of holding and delay costs. For distributed flowshop problems with SDST, Karabulut et al. [40] proposed a MILP model and a constraint programming model for the problem, and further developed a self-adaptive evolution strategy that embeds a ruin-and-recreate procedure to obtain high-quality solutions in short computational times. More recently, He et al. [41] developed a historical information-based iterated greedy algorithm, in which a set-covering model is used to learn effective factory allocation patterns from historical solutions for total flow time minimization.

2.3 Artificial Bee Colony Algorithms in Distributed Assembly Scheduling

The Artificial Bee Colony (ABC) algorithm, originally proposed by Karaboga [42], is a swarm-based metaheuristic that mimics the foraging behavior of honey bee colonies. Owing to its simplicity, few control parameters, and strong global exploration capability, it has been successfully applied to a wide range of scheduling problems, as comprehensively reviewed by Gorkemli et al. [43]. In the context of distributed assembly scheduling, existing ABC-based studies have addressed both problem extensions and algorithmic enhancements. Zuo et al. [44] extended ABC to handle fuzzy processing and assembly times in the standard DAPFSP, while Jiang et al. [45] developed a hybrid ABC-CP algorithm that couples ABC with a constraint programming model for the distributed assembly hybrid flowshop problem.

Beyond extending the problem itself, a common thread in more recent ABC-based studies is the adaptive selection of search operators. Since no single neighborhood operator performs consistently well across problem instances and search stages, adaptive operator selection has become an effective means to enhance algorithm robustness. Wang et al. [46] reinforced ABC with Q-learning for a distributed three-stage assembly problem with maintenance constraints, and Wang et al. [47] embedded Q-learning into ABC to guide operator selection for a DAPFSP variant with factory eligibility, transportation capacity, and sequence-independent setup times. Li et al. [48] and Li

et al. [49] also adopted Q-learning to assist ABC for the DAPFSP with batch delivery. Yu et al. [50] employed Q-learning as well, embedding it into several improved metaheuristics including ABC for distributed assembly permutation flowshops.

3 Notations

The notations used throughout this paper are summarized in Table 1.

Table 1: Notations used in this paper.

Notation	Description
Indices	
i	Machine index, $i = 1, 2, \dots, m$
j, j'	Job index, $j, j' = 1, 2, \dots, n$
k	Factory index, $k = 1, 2, \dots, f$
h, h'	Product index, $h, h' = 1, 2, \dots, d$
l	Position index in the product sequence, $l = 1, 2, \dots, d$
b	Individual index, $b = 1, 2, \dots, d$
Parameters	
n	Total number of jobs
m	Number of machines in each factory
f	Number of identical factories
d	Number of products
p_{ij}	Processing time of job j on machine i
$s_{i,j,j'}^M$	SDST on machine i after job j and before job j'
$s_{i,0,j}^M$	Initial setup time on machine i before the first job j
t_h	Assembly time of product h
$s_{h,h'}^A$	SDST on the assembly machine after product h and before product h'
$s_{0,h}^A$	Initial setup time on the assembly machine before the first product h
Ω_h	Set of jobs belonging to product h
$ \Omega_h $	Number of jobs in product h
Decision Variables	
$C_{i,j}$	Completion time of job j on machine i in the manufacturing stage
C_h^M	Completion time of the jobs belonging to product h
C_h^A	Completion time of product h in the assembly stage
$C_{\max}$	Completion time of the last assembled product
Sequences and Derived Variables	
$X = (\Pi, \sigma)$	A feasible solution, consisting of a product sequence and a job sequence
B	Block sequence obtained by compressing the product sequence of a solution
$\Pi = [\pi_1, \pi_2, \dots, \pi_d]$	Product sequence
π_l	The l -th product in the assembly sequence
σ	Complete job sequence
λ_k	Job sequence of factory k
L_k	Number of jobs assigned to factory k
I_l	Idle time on the assembly machine between the setup and the assembly of the l -th product
TST^A	Total setup time on the assembly machine
TIT^A	Total idle time on the assembly machine
K	Number of products in the CPS
Algorithm Parameters	
W	Sliding window size in SW-UCB1
θ	Stagnation threshold for scout bee
φ	Onlooker bee selection ratio

4 Problem Statement and Solution Representation

4.1 Problem Description

As shown in Fig. 1, the DAPFSP-SDST consists of a distributed manufacturing stage and a centralized assembly stage. Compared with the classical DAPFSP, the DAPFSP-SDST additionally considers SDSTs in both stages, i.e., between consecutive jobs in each factory and between consecutive products on the assembly machine. In the first stage, n jobs are assigned to f identical factories, each containing m machines in series. Within each factory, jobs are processed in the same order on all machines. In the second stage, the completed jobs are assembled into d products on a single assembly machine M_A . Each product h is assembled from a predefined subset of jobs Ω_h , and its assembly can start only after all jobs in Ω_h are completed and M_A is free. Solving the problem requires determining the assignment of jobs to factories, the job sequence within each factory, and the product assembly sequence. The objective is to minimize the makespan $C_{\max}$.

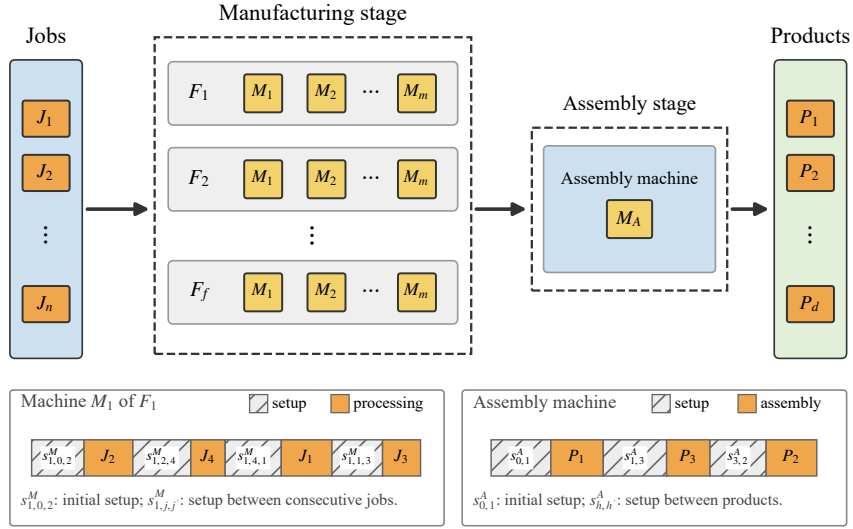


Fig. 1 System architecture of the DAPFSP-SDST.

The standard assumptions are as follows: all jobs are available at time zero; no machine breakdowns or preemption; at any given time, a machine can process at most one job, and a job can be processed on at most one machine; no job transfer between factories; each job belongs to exactly one product, and all jobs are used for product assembly; infinite intermediate storage is available; all the processing, assembly, and setup times are deterministic non-negative integers known in advance.

4.2 Makespan Calculation

For a given solution, the makespan is obtained by decoding the product sequence and the corresponding job sequence into a feasible schedule. First, the jobs are dispatched to the factories according to a given assignment rule, yielding a job sequence

$$\lambda_k = [\lambda_k(1) \cdots \lambda_k(L_k - 1) \lambda_k(L_k)] \quad (1)$$

for factory k , where L_k is the number of jobs assigned to factory k . For the first job in each factory, it can be obtained that

$$C_{1,\lambda_k(1)} = p_{1,\lambda_k(1)} + s_{1,0,\lambda_k(1)}^M \quad (2)$$

$$C_{i,\lambda_k(1)} = \max\{C_{i-1,\lambda_k(1)}, s_{i,0,\lambda_k(1)}^M\} + p_{i,\lambda_k(1)}, \quad i = 2, \dots, m \quad (3)$$

For the remaining jobs in factory k , that is, $u = 2, \dots, L_k$, we have

$$C_{1,\lambda_k(u)} = C_{1,\lambda_k(u-1)} + p_{1,\lambda_k(u)} + s_{1,\lambda_k(u-1),\lambda_k(u)}^M \quad (4)$$

$$C_{i,\lambda_k(u)} = \max\{C_{i,\lambda_k(u-1)} + s_{i,\lambda_k(u-1),\lambda_k(u)}^M, C_{i-1,\lambda_k(u)}\} + p_{i,\lambda_k(u)}, \quad i = 2, \dots, m \quad (5)$$

The setups in Eqs. (3) and (5) are anticipatory, that is, machine i can be set up for a job while that job is still being processed on machine $i - 1$. After all jobs are processed, the completion time of job j on the last machine is $C_{m,j}$. The latest completion time among the constituent jobs within product h is

$$C_h^M = \max_{j \in \Omega_h} C_{m,j}. \quad (6)$$

The assembly completion time of each product is computed according to the assembly sequence Π . For the first product:

$$C_{\pi_1}^A = \max\{C_{\pi_1}^M, s_{0,\pi_1}^A\} + t_{\pi_1} \quad (7)$$

For subsequent products $l = 2, 3, \dots, d$:

$$C_{\pi_l}^A = \max\{C_{\pi_l}^M, C_{\pi_{l-1}}^A + s_{\pi_{l-1},\pi_l}^A\} + t_{\pi_l} \quad (8)$$

and the makespan is

$$C_{\max} = C_{\pi_d}^A. \quad (9)$$

4.3 Solution Encoding and Decoding

A solution in the proposed algorithm is represented by two components: a product sequence Π and a job sequence σ . The product sequence determines the order in which products enter the assembly stage, while the job sequence determines the priority in which individual jobs are processed on the manufacturing machines.

In the decoding process, jobs are dispatched to factories in the order they appear in σ . We adopt the commonly used NR2 rule [51], in which each job is assigned to the factory with the smallest current completion time. If multiple factories tie, the one with the smallest index is chosen. Once all the constituent jobs for a product have been completed and the assembly machine is available, the assembly of the product can be started. It should be pointed out that we make sure that the assembly order follows Π , but the start times depend on both product ready times and machine availability.

As for the encoding scheme, we adopt a stage-dependent representation tailored to the distinct roles of the employed bee and onlooker/scout bee phases. Specifically, since the employed bee phase operates at the product-sequence level and a full job sequence is required to evaluate a solution, σ is constructed by concatenating the job sub-sequences of all products according to Π . This yields the multi-permutation representation (MPR), in which all jobs belonging to the same product remain contiguous. In contrast, the onlooker bee and scout bee phases operate at the job-sequence level, which corresponds to the full permutation representation (FPR). Unlike MPR, jobs belonging to the same product are not necessarily contiguous in FPR and they may be separated by jobs from other products. Nevertheless, the assembly order is still preserved by the stored product sequence.

For instance, if product 1 requires jobs (1, 3, 4) and product 2 requires jobs (2, 5), then for $\Pi = [1, 2]$, the MPR gives $\sigma = (1, 3, 4, 2, 5)$, whereas the FPR may give $\sigma = (1, 2, 3, 5, 4)$. In the FPR, jobs of the same product are allowed to be separated, but the assembly order is still determined by the stored $\Pi = [1, 2]$.

5 Block-Guided Artificial Bee Colony Algorithm

5.1 Algorithm Foundation

The design of the proposed BG-ABC algorithm is based on several interesting observations on the solution structure of the DAPFSP-SDST, which are described in this subsection.

First, as established by Song et al. [8], the assembly Gantt chart in Fig. 2 shows that before a product is assembled, a setup time is required on the assembly machine, and an idle period may follow this setup. The product is then assembled by a process taking its given assembly time. Thus, the makespan can be viewed as the sum of d product-specific segments, each comprising a setup time, an idle period, and an assembly time. Regrouping these elements by type yields Eq. (10):

$$C_{\max}(X) = \sum_{l=1}^d t_{\pi_l} + \text{TST}^A(\Pi) + \text{TIT}^A(\Pi, \sigma), \quad (10)$$

where

$$\text{TST}^A(\Pi) = \sum_{l=1}^d s_{\pi_{l-1}, \pi_l}^A, \quad \text{TIT}^A(\Pi, \sigma) = \sum_{l=1}^d I_l, \quad (11)$$

with $s_{\pi_0, \pi_1}^A \triangleq s_{0, \pi_1}^A$ denoting the initial setup before the first product. The first term is a constant, whereas TST^A and TIT^A are both variables. Consequently, minimizing

the makespan is equivalent to minimizing $TST^A + TIT^A$. The idle time associated with product π_l is given by

$$I_l = \max\left\{0, C_{\pi_l}^M - C_{\pi_{l-1}}^A - s_{\pi_{l-1}, \pi_l}^A\right\}, \quad l = 1, \dots, d, \quad (12)$$

where $C_{\pi_0}^A \triangleq 0$. If all jobs belonging to Ω_{π_l} are completed no later than the previous assembly completion time plus the corresponding SDST, the corresponding idle time is zero, that is,

$$I_l = 0 \quad \text{if} \quad C_{\pi_l}^M \leq C_{\pi_{l-1}}^A + s_{\pi_{l-1}, \pi_l}^A. \quad (13)$$

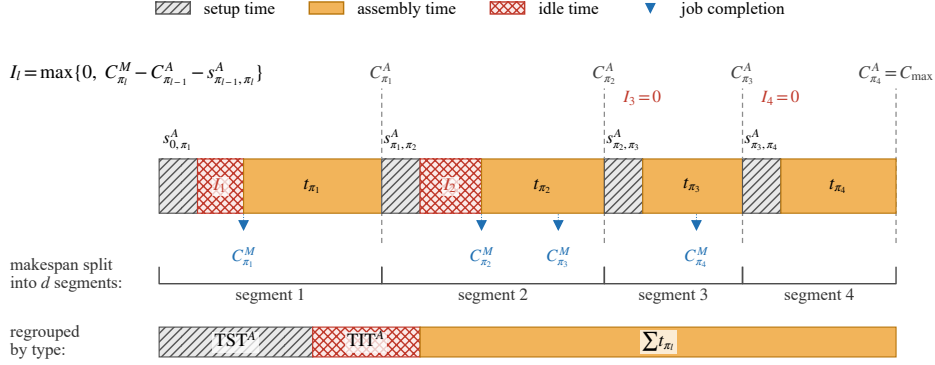


Fig. 2 Assembly machine schedule of an example product sequence $\Pi = [\pi_1, \pi_2, \pi_3, \pi_4]$. A positive idle time I_l appears between the setup and the assembly of π_1 and π_2 , whereas $I_3 = I_4 = 0$.

Second, TST^A depends only on the product sequence Π and can therefore be optimized by changing the order in which the products are assembled. In contrast, TIT^A depends on both the product sequence Π and the job sequence σ . It is obvious that each term I_l is nonnegative and attains its minimum at zero. Then, $I_l = 0$ means that the idle time before the assembly of product π_l has already reached its minimum. This condition yields a useful guideline: for the last subsequence of products whose idle times are all zero, there is no value in further adjusting the processing order of their jobs, since the corresponding idle times cannot be reduced any further; by contrast, the jobs of earlier products are more worthwhile to reorder, as they offer a greater opportunity to reduce the makespan. As shown in Fig. 2, the idle times of the last two products are both zero, so adjusting the orders of the jobs belonging to them cannot reduce TIT^A .

5.2 Population Initialization

We can easily infer that the first product typically incurs idle time, because its assembly operation cannot commence until all of its constituent jobs have completed their

Algorithm 1 Population initialization of BG-ABC.

Input: Assembly SDST matrix s^A , job sets Ω_h , number of products d

Output: Population $\{X_1, X_2, \dots, X_d\}$, where $X_b = (\Pi_b, \sigma_b)$

```
1: for  $b = 1$  to  $d$  do
2:    $\Pi_b \leftarrow [b]$   $\triangleright$  the product sequence of  $X_b$  starts with the  $b$ -th product
3:    $U \leftarrow \{1, 2, \dots, d\} \setminus \{b\}$   $\triangleright$  unsequenced products
4:    $\pi \leftarrow b$ 
5:   while  $U \neq \emptyset$  do
6:      $\pi^* \leftarrow$  an  $h \in U$  minimizing  $s_{\pi, h}^A$ , selected at random if more than one such
        $h$  exists  $\triangleright$  SDST-minimal successor
7:     append  $\pi^*$  to  $\Pi_b$ ;  $U \leftarrow U \setminus \{\pi^*\}$ ;  $\pi \leftarrow \pi^*$ 
8:   end while
9:    $\sigma_b \leftarrow$  concatenation of the default job orders of the products in  $\Pi_b$ 
10:  decode  $X_b$  to obtain its makespan  $C_{\max}(X_b)$ 
11: end for
12: return  $\{X_1, X_2, \dots, X_d\}$ 
```

manufacturing processes, and there is no preceding product that could mask this waiting time through overlapping. The idle time equals zero only when $C_{\pi_1}^M \leq s_{0, \pi_1}^A$, which is rarely encountered in practice, since the initial setup time on the assembly machine is usually much shorter than the time required to process the constituent jobs. This typical idle time implies that the choice of the first product has a decisive impact on the makespan.

The population size is chosen to be equal to the number of products. Each individual is initialized with a unique first product to ensure full diversity, that is, the first product of individual X_b is the b -th product. Starting from this first product, the remaining products are appended greedily by minimizing the local SDST. If more than one successor has the same minimum SDST, then one of them is selected at random. The job sequence σ is constructed by concatenating the default job orders of the products in Π , where the default job order of a product lists its jobs in ascending index order. The overall procedure is summarized in Algorithm 1.

5.3 Idle-Aware SDST-Minimal Block Compression

From the analysis in Section 5.1, we can see that consecutive product pairs with both SDST-minimal setup time and zero idle time are always preferable in high-quality solutions, since such an adjacent pair achieves its minimum possible contribution to the makespan. Therefore, fixing such pairs into atomic blocks offers substantial search value, as it preserves promising structural patterns that are highly likely to appear in high-quality solutions while reducing the dimensionality of the search space. Motivated by this observation, we introduce the following definition.

Definition 1 (Compressible block) A consecutive product pair (π_l, π_{l+1}) in the product sequence Π is called a *compressible block* if the following two conditions hold simultaneously:

- (i) π_{l+1} is the SDST-minimal successor of π_l ;
- (ii) the idle time between their assembly operations is zero, that is, $I_{l+1} = 0$.

Two compressible blocks are merged into a larger block when the last product of the first block is identical to the first product of the second block. In the employed bee phase, these blocks are treated as atomic units, so that the neighborhood operators operate on the block sequence rather than on a single product sequence.

To illustrate this mechanism, consider the product sequence $\Pi = [1, 2, 3, 4]$ shown in Fig. 2. Suppose the SDSTs between these products are given by the following matrix:

$s_{h,h'}^A$	1	2	3	4
1	0	15	20	30
2	10	0	5	25
3	12	8	0	6
4	18	22	9	0

From this matrix, products 3 and 4 are the SDST-minimal successors of products 2 and 3, respectively, since $s_{2,3}^A = 5$ and $s_{3,4}^A = 6$ are the minimum values in their respective rows. Combined with $I_3 = I_4 = 0$, both adjacent pairs $[2, 3]$ and $[3, 4]$ are *compressible blocks*. Since the last product of $[2, 3]$ is the same as the first product of $[3, 4]$, they are merged into a single larger atomic block $[2, 3, 4]$.

Consequently, the original search space of the product sequence, which contains $4! = 24$ permutations, is compressed into $2! = 2$ block-level permutations. This reduction allows the algorithm to focus its computational effort on more critical combinatorial decisions, thereby improving search efficiency. However, this compression introduces an inherent risk of premature convergence. By permanently binding $[2, 3, 4]$ together, the algorithm is prevented from exploring configurations where breaking this block might lead to a globally better schedule. To address this limitation, the scout bee phase (detailed in Section 5.6) will be designed to detect stagnation and dissolve these atomic blocks, thereby reconstructing the search space and escaping local optima.

5.4 Employed Bee Phase

The employed bee phase aims to explore promising regions of the solution space. Since each product sequence defines a distinct region where all solutions share the same assembly order, this phase operates at the product-sequence level. After initialization, the product sequence of each individual is compressed into a block sequence via the idle-aware compression mechanism. The BG-ABC algorithm then performs global exploration directly on this compressed block representation, using three commonly-used neighborhood operators to generate new candidates:

- **Swap:** randomly selects two blocks and exchanges their positions.
- **Insertion:** removes a block and reinserts it at another position.
- **Inversion:** reverses the order of a randomly selected subsequence of blocks.

While swap and insertion perform fine-grained local adjustments, inversion introduces more disruptive changes by reversing entire subsequences. Since no single operator consistently dominates throughout this phase, we adopt the SW-UCB1 algorithm for adaptive operator selection. For each operator $op \in \{\text{swap}, \text{insertion}, \text{inversion}\}$, SW-UCB1 maintains a sliding window of its most recent W rewards. At each selection step, the operator with the highest UCB value is selected:

$$\text{UCB}(op) = \bar{\rho}_{op} + \sqrt{\frac{2 \ln(\min(T, W))}{n_{op}}} \quad (14)$$

where $\bar{\rho}_{op}$ is the average reward within the window, n_{op} is the number of times op has been selected within the window, and T is the total number of selections so far.

The sliding window discards outdated historical information and quickly adapts to the time-varying effectiveness of operators as the block structure evolves. Compared to Q-learning, which requires jointly tuning the learning rate, discount factor, and exploration strategy, SW-UCB1 involves only a single window-size parameter, making it more robust and easier to implement in this context.

Once an operator is applied to the block sequence, the resulting sequence is decompressed into a product sequence, and the job sequence is constructed by concatenating the current job orders of the products in the new product sequence. The full makespan is then evaluated. Based on the improvement achieved, a reward is assigned to the operator according to the scheme defined in Table 2, which is then fed back to update the SW-UCB1 statistics. The selection and feedback procedure is detailed in Algorithm 2.

Table 2 Reward scheme for operator evaluation in the employed bee phase.

Condition	Reward
New global best found	5
New individual best found	2
Improvement achieved (non-best)	1
No improvement, but worsening $< 1\%$	0.3
No improvement, worsening $\geq 1\%$	0

After evaluation, the product sequence is re-compressed into a new block sequence for subsequent operations. The makespan of this solution is used as an indicator of the potential of the entire region defined by the product sequence, since all job sequences within this region share the same TST^A and differ only in TIT^A . A smaller makespan suggests a lower achievable bound for this region, and thus a higher priority for further exploitation. If the makespan improves, the individual is updated and its stagnation counter is reset to zero; otherwise, the stagnation counter is incremented by one. The overall procedure of the employed bee phase is outlined in Algorithm 3.

Algorithm 2 SW-UCB1-based operator selection and reward feedback.

Input: operator set $\mathcal{O} = \{\text{swap}, \text{insertion}, \text{inversion}\}$; window size W ; reward scheme in Table 2

Output: the selected operator op^* and the updated sliding-window statistics

Selection

- 1: $T \leftarrow T + 1$ $\triangleright$ total number of selections so far
- 2: **if** some $op \in \mathcal{O}$ has $n_{op} = 0$ **then**
- 3: $op^* \leftarrow$ such an op $\triangleright$ each operator is tried at least once
- 4: **else**
- 5: **for** each $op \in \mathcal{O}$ **do**
- 6: $UCB(op) \leftarrow \bar{\rho}_{op} + \sqrt{2 \ln(\min(T, W)) / n_{op}}$
- 7: **end for**
- 8: $op^* \leftarrow \arg \max_{op \in \mathcal{O}} UCB(op)$
- 9: **end if**

Feedback $\triangleright$ after the offspring of op^* is evaluated

 - 10: $\rho \leftarrow$ reward determined by Table 2
 - 11: push ρ into the window of op^* ; remove the oldest record if the window size exceeds W
 - 12: update $\bar{\rho}_{op^*}$ and n_{op^*} over the current window

5.5 Onlooker Bee Phase

Following the global exploration by the employed bees, the onlooker bee phase focuses on exploiting the most promising individuals. To balance the computational effort, only a proportion φ of the current population are selected using roulette wheel selection based on makespan. However, within a given region (i.e., a fixed product sequence), not all products require job-order optimization. As already shown in Section 5.1, only the prefix up to the last product with positive idle time requires refinement. Following the notion introduced by Song et al. [8], we refer to this prefix as the Critical Product Sequence (CPS) of length K , and the concatenation of the jobs belonging to these products forms the Critical Job Subsequence (CJS). The onlooker bee therefore performs local search exclusively on the CJS without altering the product sequence. Two neighborhood operators are applied.

- **Intra-product swap:** for each product h in the CPS, one pair of distinct jobs of that product is randomly selected and swapped.
- **Inter-product swap:** two positions are randomly selected from the CJS and the corresponding jobs are swapped.

For each product h in the CPS, the intra-product swap is executed $2|\Omega_h|$ times, whereas the inter-product swap is executed $5K$ times. To screen these candidate moves efficiently, we adopt a two-level evaluation strategy. The first level computes a partial makespan consisting only of the first K products and the computational cost of this partial evaluation is approximately K/d of a full evaluation. The second level computes the complete makespan over all d products and is triggered only when the candidate

Algorithm 3 Employed bee phase.

Input: population $\{X_1, X_2, \dots, X_d\}$ in block representation; SW-UCB1 statistics**Output:** updated population, stagnation counters, personal bests, and global best

```
1: for each individual  $X_b$  do
2:    $op^* \leftarrow$  SW-UCB1 selection
3:    $B' \leftarrow$  apply  $op^*$  to the block sequence of  $X_b$ 
4:    $\Pi' \leftarrow$  decompress( $B'$ );  $\sigma' \leftarrow$  concatenate the current job orders of the
      products in  $\Pi'$ 
5:   evaluate the full makespan of  $(\Pi', \sigma')$ 
6:    $B' \leftarrow$  re-compress  $(\Pi', \sigma')$  into a block sequence
7:   feed the reward of the outcome back to  $op^*$ 
8:   if the makespan is improved then
9:      $X_b \leftarrow (\Pi', \sigma')$  with block representation  $B'$ ; stagnation counter  $st_b \leftarrow 0$ 
10:    update the personal best of  $X_b$ 
11:    update the global best whenever it is improved
12:   else
13:      $st_b \leftarrow st_b + 1$ 
14:   end if
15: end for
16: return the updated population
```

outperforms the incumbent at the first level. The overall procedure is detailed in Algorithm 4.

5.6 Scout Bee Phase

While idle-aware block compression improves search efficiency, it also carries the risk of trapping the search in local optima, as the locked blocks may prevent the algorithm from exploring alternative structural configurations. The scout bee phase addresses this limitation through a refinement-and-reconstruction mechanism that detects stagnation, attempts a targeted refinement of the stagnant solution, and rebuilds the search structure when the refinement fails. Once the stagnation counter exceeds a predefined threshold θ , the individual is considered stagnant and triggers the scout bee procedure.

Before discarding the stagnant solution, an insertion-based neighborhood descent on SDST (SDST-ND) is applied to the non-critical part of its product sequence, followed by an insertion-based descent on the CJS of the new sequence. In the SDST-ND, one product is removed and reinserted at another position of that part, and a move is accepted whenever it reduces the total setup time. In the CJS descent, each job of the CJS is removed and reinserted at up to f positions ahead in the CJS, and the first improving reinsertion is accepted. If either descent yields an improvement, the resulting solution is fully evaluated to update the individual best and the global best if applicable. Afterwards, the following structural reset is triggered.

- (1) A first product is randomly selected from the set of first products that appear in the best solutions of the current population.
- (2) The remaining products are randomly permuted.

Algorithm 4 Onlooker bee phase.

Input: population $\{X_1, X_2, \dots, X_d\}$ with stagnation counters $\{st_1, st_2, \dots, st_d\}$; onlooker bee ratio φ

Output: updated population, personal bests, and global best

```
1: select  $\varphi \cdot d$  individuals by roulette wheel selection according to their makespan
2: for each selected individual  $X_b$  do
3:   extract the CPS of length  $K$  and the corresponding CJS
4:   for each product  $h$  in the CPS do
5:     for  $t = 1$  to  $2|\Omega_h|$  do
6:       swap two randomly chosen distinct jobs of product  $h$  in the CJS
7:       evaluate the move at the two levels
8:       if the move is improved then
9:         update the individual, the personal best of  $X_b$ 
10:        update the global best whenever it is improved
11:         $st_b \leftarrow 0$ 
12:      end if
13:    end for
14:  end for
15:  if  $K > 2$  then
16:    for  $t = 1$  to  $5K$  do
17:      two positions are randomly selected from the CJS and the corresponding
18:      jobs are swapped.
19:      evaluate the move at the two levels
20:      if the move is improved then
21:        update the personal best of  $X_b$ 
22:        update the global best whenever it is improved
23:         $st_b \leftarrow 0$ 
24:      end if
25:    end for
26:  end if
27: return the updated population
```

(3) The job order within each product is randomly shuffled.

This reset destroys all existing blocks and creates a new adjacency structure, which will be re-compressed in the subsequent employed bee phase. The stagnation counter is reset to zero, allowing the algorithm to initiate new space-reduction cycles from various regions of the solution space. The overall procedure of the scout bee phase is outlined in Algorithm 5.

5.7 Overall Framework and Computational Complexity

By integrating the components described in the preceding subsections, the proposed BG-ABC follows the overall procedure summarized in Fig. 3. Starting from the initialized population, the algorithm repeatedly performs the employed, onlooker, and

Algorithm 5 Scout bee phase.

Input: population $\{X_1, X_2, \dots, X_d\}$ with stagnation counters $\{st_1, st_2, \dots, st_d\}$; threshold θ ; number of factories f

Output: updated population with reset stagnation counters, personal bests, and global best

```
1: for each individual  $X_b$  with  $st_b > \theta$  do
2:    $\Pi_b^{\text{nc}} \leftarrow$  the products of  $\Pi_b$  outside the CPS
3:   apply SDST-ND to  $\Pi_b^{\text{nc}}$ , yielding a new product sequence  $\Pi'_b$ 
4:    $accepted \leftarrow \text{false}$ 
5:   if the total SDST of  $\Pi'_b$  is smaller than that of  $\Pi_b$  then
6:     decode  $\Pi'_b$  with the current job orders to obtain  $\sigma'_b$  and evaluate its
       makespan
7:     apply one insertion-based descent to the CJS of  $\Pi'_b$  (updating  $\sigma'_b$  accordingly)
8:     if the makespan is improved then
9:        $X_b \leftarrow (\Pi'_b, \sigma'_b)$ ;  $accepted \leftarrow \text{true}$ 
10:      update the personal best of  $X_b$ 
11:      update the global best whenever it is improved
12:    end if
13:  end if
14:  if not  $accepted$  then
15:    select a first product  $h$  uniformly at random from the first products of the
      best solutions in the population
16:    permute the remaining products randomly, and shuffle the job order within
      each product
17:     $X_b \leftarrow$  the reconstructed solution
18:  end if
19:   $st_b \leftarrow 0$ 
20: end for
21: return the updated population
```

scout bee phases until the termination criterion is satisfied, and finally returns the best solution found.

As for computational complexity, a single full evaluation takes $O(nm)$ time. The initialization phase constructs and evaluates d individuals in $O(d^3 + dnm)$ time. The employed bee phase costs $O(d(nm + q))$ per generation, since it operates directly on the q compressed blocks. The onlooker bee phase generates $O(n + K)$ candidate moves per selected individual, but its first-level partial evaluation costs only about K/d of a full evaluation, filtering out most unpromising candidates before the expensive full evaluation is triggered. The scout bee phase is invoked only for stagnant individuals; its SDST-ND and CJS descent have a worst-case cost of $O(d^3 + nf)$, which has a limited impact on the overall runtime. Overall, under a budget of E full evaluations, the time complexity of BG-ABC is $O(E \cdot nm)$.

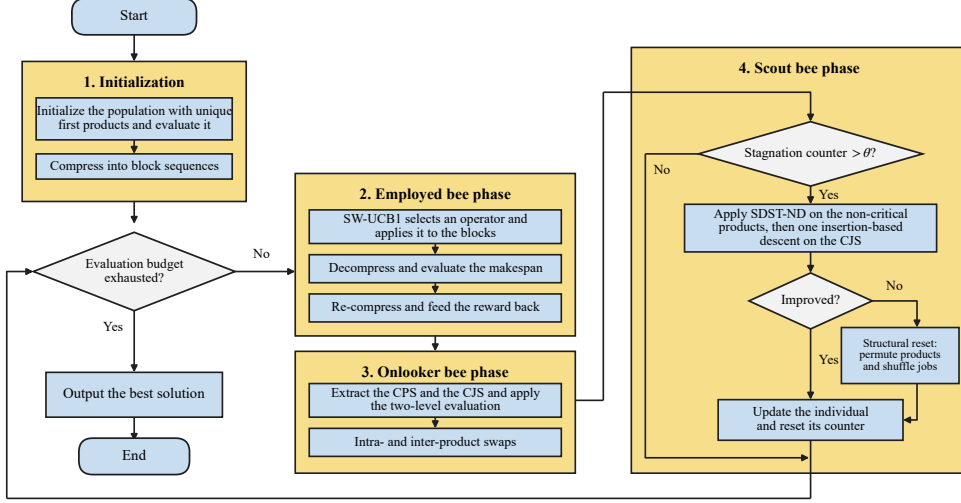


Fig. 3 Overall framework of the proposed BG-ABC.

6 Experimental Study

6.1 Experimental Setup

The performance of the proposed BG-ABC is evaluated on a set of benchmark instances originally generated by Hatami et al. [7] for the DAPFSP-SDST. These instances are constructed from combinations of five problem parameters, as summarized in Table 3. This yields $2 \times 3 \times 3 \times 3 \times 2 = 108$ parameter combinations.

Table 3 Problem parameters of the benchmark instances.

Parameter	Levels	Values
n	2	100, 200
m	3	5, 10, 20
f	3	4, 6, 8
d	3	30, 40, 50
SDST range	2	[1, 50], [1, 125]

For each combination, five instances are randomly generated, resulting in a total of $108 \times 5 = 540$ instances. All instances and their best-known solutions are publicly available at <http://soa.iti.es>.

The proposed BG-ABC, like other metaheuristics, is a stochastic algorithm whose performance may vary across independent runs due to randomness in the search process. Therefore, in order to obtain a reliable assessment, each instance is run $R = 10$

times independently with different random seeds, and the results are aggregated for evaluation.

To compare the performance of different algorithms, we define the Relative Percentage Deviation (RPD) for a single run of algorithm A on instance $\mathcal{I}$ as:

$$\text{RPD}(A, \mathcal{I}) = \frac{C_{\max}(A, \mathcal{I}) - C_{\max}^*(\mathcal{I})}{C_{\max}^*(\mathcal{I})} \times 100\% \quad (15)$$

where $C_{\max}(A, \mathcal{I})$ is the makespan obtained by algorithm A on instance $\mathcal{I}$, and $C_{\max}^*(\mathcal{I})$ is the best-known makespan for that instance. The best-known values are those originally released with the benchmark [7] and remain fixed throughout this study. Therefore, a negative RPD implies the algorithm outperforms the published best-known value, while a positive RPD means that it is not reached. ARPD is the mean RPD over R runs, reflecting the average performance of the algorithm, which is defined as follows:

$$\text{ARPD}(A, \mathcal{I}) = \frac{1}{R} \sum_{r=1}^R \text{RPD}(A_r, \mathcal{I}) \quad (16)$$

All algorithms are implemented in Python and run on a desktop computer with an Intel i7-12700 processor and 32 GB RAM. The stopping criterion for all algorithms is a maximum number of solution evaluations. Specifically, for an instance with d products, each algorithm is allowed to perform at most $d \times 3000$ evaluations. This evaluation budget is scaled with the problem size to ensure a fair comparison across instances of different scales. For fair comparison, all competing algorithms are executed under the same termination condition and hardware environment.

6.2 Parameter Settings

We conducted a parameter tuning experiment on three key parameters: the stagnation threshold $\theta \in \{25, 50, 75, 100\}$, the window size $W \in \{50, 100, 150\}$, and the onlooker bee ratio $\varphi \in \{5\%, 10\%, 15\%\}$. For each of the 36 parameter combinations, we ran the algorithm on 29 calibration instances covering all instance scales, with 10 independent runs per instance, resulting in a total of $36 \times 29 \times 10 = 10,440$ runs.

We adopt the range analysis [52] to evaluate the parameter effects. For each parameter, the average makespan at each level is computed across all instances and all combinations involving that level. The optimal level for each parameter is the one yielding the smallest average makespan. The range Δ —defined as the difference between the maximum and minimum average makespan across levels of a given parameter—indicates the relative importance of that parameter: a larger range means the parameter has a greater influence on algorithm performance. Table 4 reports the range analysis results. For each parameter, the average makespan at each level, the range, and the optimal level are presented.

First, the stagnation threshold θ exhibits the largest range ($\Delta = 2.4$), indicating that it is the most critical parameter affecting algorithm performance. We can see that a small θ may trigger the scout bee to reset individuals prematurely, cutting off the search along promising directions before they have been sufficiently exploited. Conversely, a large θ may waste too much computational resources on individuals

Table 4 Range analysis of the three key parameters.

Parameter	Level	Average makespan	Range Δ	Optimal Level
Stagnation threshold θ	25	7782.7	2.4	50
	50	7781.2		
	75	7782.1		
	100	7783.6		
Window size W	50	7782.4	1.2	100
	100	7781.8		
	150	7783.0		
Onlooker ratio φ	5%	7782.9	0.9	10%
	10%	7782.0		
	15%	7782.3		

that have already stagnated. The data in Table 4 support this interpretation: the best average makespan is obtained at $\theta = 50$, while both smaller and larger values lead to performance degradation.

A small window makes the SW-UCB1 operator selection overly sensitive to recent fluctuations, while a large window retains outdated historical information that no longer reflects the current search stage. The intermediate value $W = 100$ yields the best performance by maintaining a sufficiently long history for stable reward estimation while remaining responsive to changes in the search dynamics.

The onlooker bee ratio φ controls the balance between global exploration and local exploitation. A small φ implies that only a few top individuals undergo local search, which may leave the promising regions insufficiently exploited. Conversely, a large φ causes too many individuals to be refined, which consumes too much computational effort in this phase. The intermediate value $\varphi = 10\%$ strikes the best balance, focusing computational effort on the elite individuals while maintaining the evolutionary progress of the population.

6.3 Ablation Study

To validate the effectiveness of the four key components in the proposed BG-ABC algorithm, we conduct ablation experiments on four variants:

- **BG-ABC-v1**: removes the idle-aware block compression mechanism and operates directly on the raw product sequence throughout the search. This variant is designed to verify the contribution of the block compression mechanism.
- **BG-ABC-v2**: replaces the SW-UCB1 operator selector with uniform random selection among the three operators (swap, insertion, inversion). This variant is designed to verify the contribution of the adaptive operator selection mechanism.
- **BG-ABC-v3**: replaces the CJS-oriented neighborhood and the two-level evaluation strategy of the onlooker bee phase with a plain neighborhood search over the whole product sequence in which every candidate is fully evaluated. This variant is designed to verify the contribution of the CJS-oriented local search with two-level evaluation.

- **BG-ABC-v4**: replaces the SDST-ND-guided reconstruction and the insertion-based descent of the scout bee phase with a uniform random reinitialization of the stagnating individual. This variant is designed to verify the contribution of the guided reconstruction mechanism.

All variants are evaluated on the same 540 benchmark instances with 10 independent runs per instance, using the same termination criterion and parameter settings as the full BG-ABC. We employ the Wilcoxon signed-rank test [53], with BG-ABC as the control algorithm, to detect significant differences between BG-ABC and each variant. Instances with identical makespan are excluded as ties. Because four comparisons are conducted, the Holm–Bonferroni procedure is applied to control the family-wise error rate, and the adjusted p -values are reported together with the significance decisions at $\alpha = 0.05$ and $\alpha = 0.01$. Table 5 reports the results of the ablation experiments.

Table 5 Wilcoxon signed-rank test results of the ablation study (control algorithm: BG-ABC).

BG-ABC vs.	Wins	Losses	Ties	R^+	R^-	Z	Adj. p	$\alpha = 0.05$	$\alpha = 0.01$
v1	442	92	6	129675.5	13169.5	16.330	1.81×10^{-59}	Yes	Yes
v2	345	192	3	101907.0	44163.0	7.959	3.46×10^{-15}	Yes	Yes
v3	480	57	3	141304.0	4766.0	18.820	2.07×10^{-78}	Yes	Yes
v4	325	211	4	98855.5	45060.5	7.498	6.47×10^{-14}	Yes	Yes

As shown in Table 5, the full BG-ABC significantly outperforms all four variants at both significance levels. The largest performance degradations result from removing the idle-aware block compression and the CJS-oriented local search, indicating that these two mechanisms play the most critical role in exploring and exploiting the search space. Replacing SW-UCB1 with uniform random selection, or replacing the guided reconstruction of the scout bee phase with a random reinitialization, also leads to statistically significant performance losses. Overall, these results demonstrate that none of the four components is redundant.

6.4 Comparison with State-of-the-Art Algorithms

To evaluate the performance of the proposed BG-ABC algorithm, we compare it with four state-of-the-art algorithms from the literature: IG [7], GP-HH [33], NBBO [34] and TSHA [8]. All compared algorithms are re-implemented in Python and executed in the same experimental environment as BG-ABC. The parameters of each algorithm follow the recommended settings from their original publications and all algorithms use the same termination criterion to ensure a fair comparison. Each algorithm is run 10 times independently, and the average results are reported.

Table 6 shows the ARPD values of all compared algorithms, grouped by the number of jobs (n), machines (m), factories (f), products (d), and the SDST range. The negative ARPDs attained by GP-HH, TSHA, NBBO, and BG-ABC indicate that these algorithms improve upon the best-known solutions published for the benchmark. Fig. 4 further illustrates the boxplots of the per-instance ARPD values. The box of BG-ABC lies almost entirely below those of the other algorithms, and its median is the

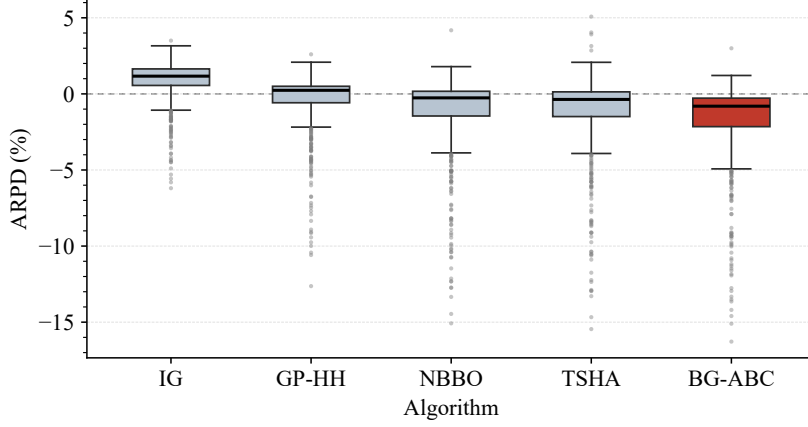


Fig. 4 Boxplots of the per-instance ARPD distributions of the compared algorithms over the 540 benchmark instances.

lowest, indicating that the improvement of BG-ABC is not driven by a few favorable instances but holds consistently across the benchmark set.

Table 6 ARPD values of the compared algorithms grouped by problem parameters.

		ARPD				
		IG	GP-HH	TSHA	NBBO	BG-ABC
n	100	1.26	-1.35	-2.23	-2.39	-3.18
	200	0.37	0.38	-0.32	-0.24	-0.73
m	5	0.93	-0.78	-1.8	-1.74	-2.38
	10	0.82	-0.42	-1.32	-1.29	-1.93
	20	0.69	-0.26	-0.71	-0.92	-1.56
f	4	0.72	0.08	-0.55	-0.71	-1.35
	6	0.83	-0.46	-1.33	-1.33	-1.97
	8	0.89	-1.08	-1.95	-1.91	-2.55
d	30	0.41	0.32	0.38	0.23	-0.34
	40	0.79	-0.01	-0.78	-0.79	-1.44
	50	1.24	-1.77	-3.42	-3.39	-4.09
SDST	[1,50]	0.49	-0.33	-0.68	-0.71	-1.16
	[1,125]	1.14	-0.64	-1.87	-1.92	-2.74
Average		0.81	-0.49	-1.28	-1.32	-1.95

To determine whether the performance differences among the compared algorithms are statistically significant, we employ the Friedman test as a non-parametric multiple-comparison procedure, followed by the Bonferroni-Dunn post-hoc test to identify specific pairwise differences [53]. Since IG performs substantially worse than all the other algorithms and lies far outside their rank range, it is excluded from the tests to avoid distorting the rank-based results. The proposed BG-ABC algorithm is designated as the control algorithm.

Table 7 presents the average ranks and RPD statistics of the compared algorithms. We can see that BG-ABC achieves the best average rank, substantially lower than the others, while IG, listed for completeness only, ranks last. The Friedman test on the remaining four algorithms yields $\chi^2 = 1180.3650$ (df = 3) with a p -value approaching zero, decisively rejecting the null hypothesis of equivalent performance.

Table 7 Average ranks and RPD statistics of the compared algorithms.

Algorithm	Avg. Rank	ARPD (%)	Median RPD (%)
BG-ABC	1.0361	-1.9541	-0.8072
TSHA	2.5194	-1.2761	-0.3633
NBBO	2.7444	-1.3171	-0.2615
GP-HH	3.7000	-0.4856	0.2361
IG	5.0000	0.8100	1.1659

Since the null hypothesis is rejected, the Bonferroni-Dunn post-hoc test is further performed to identify specific pairwise differences between the control algorithm and each competitor. The critical difference (CD) is computed as:

$$CD = q_\alpha \sqrt{\frac{a(a+1)}{6N}} \quad (17)$$

where $\alpha = 4$, $N = 540$, $q_{0.05} = 2.498$ and $q_{0.01} = 3.150$, yielding $CD_{0.05} = 0.1963$ and $CD_{0.01} = 0.2475$. Table 8 presents the Bonferroni-Dunn post-hoc test results with BG-ABC as the control algorithm.

Table 8 Bonferroni-Dunn post-hoc test results (control algorithm: BG-ABC).

Comparison	Rank difference	$\alpha = 0.05$	$\alpha = 0.01$
BG-ABC vs. TSHA	1.4833	Significant	Significant
BG-ABC vs. NBBO	1.7083	Significant	Significant
BG-ABC vs. GP-HH	2.6639	Significant	Significant

Fig. 5 compares the average ranks with the critical difference thresholds. Since every pairwise rank difference exceeds $CD_{0.05}$, no pair of algorithms is statistically indistinguishable at the 95% confidence level. The same conclusion holds at the 99% level. Therefore, BG-ABC significantly outperforms TSHA, NBBO, and GP-HH at both confidence levels.

The reasons why BG-ABC outperforms the compared algorithms can be explained as follows. On the one hand, IG and GP-HH are generic metaheuristics directly

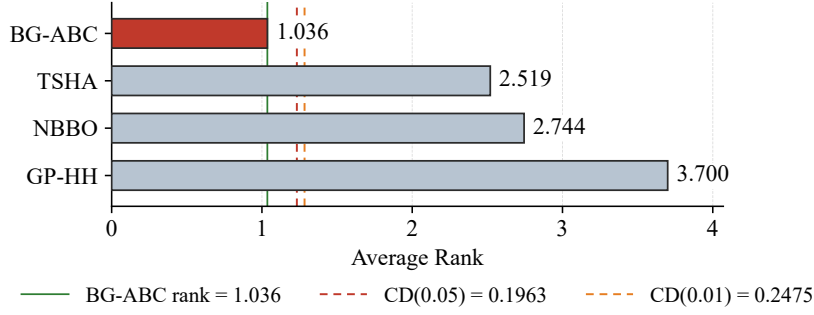


Fig. 5 Average ranks of the compared algorithms obtained by the Friedman test, together with the critical difference thresholds ($CD_{0.05} = 0.1963$ and $CD_{0.01} = 0.2475$).

adapted to this problem, and NBBO does not fully exploit the knowledge of the problem, so that none of them makes sufficient use of the domain-specific insight into the solution space. On the other hand, TSHA explicitly exploits these structural properties and therefore achieves competitive performance. However, as a single-descent heuristic, TSHA is inherently limited in its global exploration capability and easily gets trapped in local optima. In contrast, BG-ABC builds upon these structural insights and effectively integrates them into a population-based ABC framework. The block compression reduces the search space to accelerate global search, while the embedded SW-UCB1, CJS-focused local search, and refinement-and-reconstruction mechanisms work together to balance exploration and exploitation within this compressed space. This explains the consistent superiority of BG-ABC.

7 Conclusion

This paper proposes a BG-ABC algorithm for the DAPFSP-SDST. The algorithm integrates four key components: an idle-aware block compression strategy to reduce the search space, an SW-UCB1 mechanism for adaptive operator selection in the employed bee phase, a CJS local search with a two-level evaluation strategy in the onlooker bee phase, and a refinement-and-reconstruction mechanism in the scout bee phase to escape local optima. Experimental results on a comprehensive set of benchmark instances show that BG-ABC significantly outperforms TSHA, NBBO, IG and GP-HH. The ablation study confirms that each component contributes positively to the overall performance. These results verify the value of BG-ABC in handling complex scheduling decisions within multi-stage supply chain environments.

Future work may extend the block-guided framework to other distributed scheduling assembly environments, such as flexible job shops, hybrid flowshops, and DAPFSP variants with additional practical constraints like batch delivery and transportation costs. On the algorithmic side, it would be interesting to explore the integration of deep reinforcement learning (DRL) algorithms to further improve the adaptive search capability. Finally, applying BG-ABC to real-world supply chain networks with dynamic customer demands would be a valuable direction.

Supplementary information. Not applicable.

Acknowledgements. Not applicable.

Declarations

- Funding
This work is supported by the Zhejiang Provincial Natural Science Foundation of China under Grant No. LY22F030020, and by the University-Industry Collaborative Project between Zhejiang University of Finance & Economics and the industry partner under Grant No. 2026hx116.
- Conflict of interest/Competing interests
The authors declare no competing interests.
- Ethics approval and consent to participate
Not applicable
- Consent for publication
Not applicable
- Data availability
The benchmark instances used in this study are publicly available at <http://soa.iti.es>. The datasets generated and/or analysed during the current study are available from the corresponding author on reasonable request.
- Code availability
The code used during the current study is available from the corresponding author on reasonable request.
- Author contribution
J. X. curated data, developed software, and drafted the manuscript. H. S. led conceptualization and methodology, acquired funding, and edited the manuscript. All authors reviewed the manuscript.

References

- [1] Fu, Y., Li, Z., Gao, K., Fathollahi-Fard, A.M., Zhang, Z.: Integrated scheduling of distributed manufacturing with assembly and distribution: state of the art, challenges, and future directions. *Int. J. Prod. Res.* **64**(11), 4669–4710 (2025) <https://doi.org/10.1080/00207543.2025.2602898>
- [2] Liu, T., Liu, J., Qiu, J., Lai, C., Zhang, Z.: Recent advances in assembly flow shop scheduling and its extensions to distributed manufacturing and integrated supply chain. *Comput. Ind. Eng.* **212**, 111711 (2026) <https://doi.org/10.1016/j.cie.2025.111711>
- [3] Behnamian, J., Fatemi Ghomi, S.M.T.: A survey of multi-factory scheduling. *J. Intell. Manuf.* **27**(1), 231–249 (2016) <https://doi.org/10.1007/s10845-014-0890-y>
- [4] Koulamas, C., Kyparisis, G.J.: The three-stage assembly flowshop scheduling problem. *Comput. Oper. Res.* **28**(7), 689–704 (2001) [https://doi.org/10.1016/S0305-0548\(00\)00004-6](https://doi.org/10.1016/S0305-0548(00)00004-6)

- [5] Hatami, S., Ruiz, R., Andrés-Romano, C.: The distributed assembly permutation flowshop scheduling problem. *Int. J. Prod. Res.* **51**(17), 5292–5308 (2013) <https://doi.org/10.1080/00207543.2013.807955>
- [6] Pan, Q.-K., Gao, L., Li, X.-Y., Framinan, J.M.: Effective constructive heuristics and meta-heuristics for the distributed assembly permutation flowshop scheduling problem. *Appl. Soft Comput.* **81**, 105492 (2019) <https://doi.org/10.1016/j.asoc.2019.105492>
- [7] Hatami, S., Ruiz, R., Andrés-Romano, C.: Heuristics and metaheuristics for the distributed assembly permutation flowshop scheduling problem with sequence dependent setup times. *Int. J. Prod. Econ.* **169**, 76–88 (2015) <https://doi.org/10.1016/j.ijpe.2015.07.027>
- [8] Song, H., Lin, J., Chen, Y.-R.: An effective two-stage heuristic for scheduling the distributed assembly flowshops with sequence dependent setup times. *Comput. Oper. Res.* **173**, 106850 (2025) <https://doi.org/10.1016/j.cor.2024.106850>
- [9] Wang, S.-Y., Wang, L.: An estimation of distribution algorithm-based memetic algorithm for the distributed assembly permutation flow-shop scheduling problem. *IEEE Trans. Syst. Man Cybern. Syst.* **46**(1), 139–149 (2016) <https://doi.org/10.1109/TSMC.2015.2416127>
- [10] Zhang, Z.-Q., Qian, B., Hu, R., Jin, H.-P., Wang, L.: A matrix-cube-based estimation of distribution algorithm for the distributed assembly permutation flow-shop scheduling problem. *Swarm Evol. Comput.* **60**, 100785 (2021) <https://doi.org/10.1016/j.swevo.2020.100785>
- [11] Song, H.-B., Yang, Y.-H., Lin, J., Ye, J.-X.: An effective hyper heuristic-based memetic algorithm for the distributed assembly permutation flow-shop scheduling problem. *Appl. Soft Comput.* **135**, 110022 (2023) <https://doi.org/10.1016/j.asoc.2023.110022>
- [12] Ferone, D., Hatami, S., González-Neira, E.M., Juan, A.A., Festa, P.: A biased-randomized iterated local search for the distributed assembly permutation flow-shop problem. *Int. Trans. Oper. Res.* **27**(3), 1368–1391 (2020) <https://doi.org/10.1111/itor.12719>
- [13] Lin, J., Zhang, S.: An effective hybrid biogeography-based optimization algorithm for the distributed assembly permutation flow-shop scheduling problem. *Comput. Ind. Eng.* **97**, 128–136 (2016) <https://doi.org/10.1016/j.cie.2016.05.005>
- [14] Lin, J., Wang, Z.-J., Li, X.: A backtracking search hyper-heuristic for the distributed assembly flow-shop scheduling problem. *Swarm Evol. Comput.* **36**, 124–135 (2017) <https://doi.org/10.1016/j.swevo.2017.04.007>
- [15] Zhang, W., Hao, J., Liu, F.: Effective social spider optimization algorithms for

- distributed assembly permutation flowshop scheduling problem in automobile manufacturing supply chain. *Sci. Rep.* **14**, 6370 (2024) <https://doi.org/10.1038/s41598-024-57044-8>
- [16] Hamzadayı, A., Arvas, M.A., Elmi, A.: Distributed assembly permutation flow shop problem: Single seekers society algorithm. *J. Manuf. Syst.* **61**, 613–631 (2021) <https://doi.org/10.1016/j.jmsy.2021.10.012>
 - [17] Sang, H.-Y., Pan, Q.-K., Li, J.-Q., Wang, P., Han, Y.-Y., Gao, K.-Z., Duan, P.: Effective invasive weed optimization algorithms for distributed assembly permutation flowshop problem with total flowtime criterion. *Swarm Evol. Comput.* **44**, 64–73 (2019) <https://doi.org/10.1016/j.swevo.2018.12.001>
 - [18] Huang, Y.-Y., Pan, Q.-K., Huang, J.-P., Suganthan, P.N., Gao, L.: An improved iterated greedy algorithm for the distributed assembly permutation flowshop scheduling problem. *Comput. Ind. Eng.* **152**, 107021 (2021) <https://doi.org/10.1016/j.cie.2020.107021>
 - [19] Qiu, B., Gao, K., Yu, H., Sadollah, A.: Modelling and scheduling distributed assembly permutation flow-shops using reinforcement learning-based evolutionary algorithms. *Eng. Appl. Artif. Intell.* **142**, 109851 (2025) <https://doi.org/10.1016/j.engappai.2024.109851>
 - [20] Zhang, Z.-Q., Hu, R., Qian, B., Jin, H.-P., Wang, L., Yang, J.-B.: A matrix cube-based estimation of distribution algorithm for the energy-efficient distributed assembly permutation flow-shop scheduling problem. *Expert Syst. Appl.* **194**, 116484 (2022) <https://doi.org/10.1016/j.eswa.2021.116484>
 - [21] Yu, H., Gao, K., Li, Z., Suganthan, P.N.: Energy-efficient multi-objective distributed assembly permutation flowshop scheduling by q-learning based meta-heuristics. *Appl. Soft Comput.* **166**, 112247 (2024) <https://doi.org/10.1016/j.asoc.2024.112247>
 - [22] Cheng, L., Wang, L., Cai, J., Hu, K., Xiong, Y., Xia, Q.: An effective biogeography-based optimization algorithm for multi-objective green scheduling of distributed assembly permutation flowshop scheduling problem. *Comput. Oper. Res.* **183**, 107158 (2025) <https://doi.org/10.1016/j.cor.2025.107158>
 - [23] Sun, R., Song, H., Chen, Y., Zhang, X., Liu, L., Lin, J., Cui, Y.: A multiple-objective memetic algorithm for the energy-efficient scheduling of distributed assembly flow shops. *Symmetry* **18**(2), 315 (2026) <https://doi.org/10.3390/sym18020315>
 - [24] Yang, Y.-H., Li, X.: A knowledge-driven constructive heuristic algorithm for the distributed assembly blocking flow shop scheduling problem. *Expert Syst. Appl.* **202**, 117269 (2022) <https://doi.org/10.1016/j.eswa.2022.117269>

- [25] Zhao, F., Zhao, J., Wang, L., Tang, J.: An optimal block knowledge driven backtracking search algorithm for distributed assembly no-wait flow shop scheduling problem. *Appl. Soft Comput.* **112**, 107750 (2021) <https://doi.org/10.1016/j.asoc.2021.107750>
- [26] Shao, W.-S., Pi, D.-C., Shao, Z.-S.: Local search methods for a distributed assembly no-idle flow shop scheduling problem. *IEEE Syst. J.* **13**(2), 1945–1956 (2019) <https://doi.org/10.1109/jsyst.2018.2825337>
- [27] Yang, S., Xu, Z.: The distributed assembly permutation flowshop scheduling problem with flexible assembly and batch delivery. *Int. J. Prod. Res.* **59**(13), 4053–4071 (2021) <https://doi.org/10.1080/00207543.2020.1757174>
- [28] Luo, Q., Deng, Q., Guo, X., Gong, G., Zhao, X., Chen, L.: Modelling and optimization of distributed assembly hybrid flowshop scheduling problem with transportation resource scheduling. *Comput. Ind. Eng.* **186**, 109717 (2023) <https://doi.org/10.1016/j.cie.2023.109717>
- [29] Deng, J., Wang, L., Wang, S.-Y., Zheng, X.-L.: A competitive memetic algorithm for the distributed two-stage assembly flow-shop scheduling problem. *Int. J. Prod. Res.* **54**(12), 3561–3577 (2016) <https://doi.org/10.1080/00207543.2015.1084063>
- [30] Zhang, Z., Yu, S., Tang, Q., Zhang, L., Li, Z., Cheng, L., Li, Y.: A matheuristic-based self-learning approach for distributed heterogeneous assembly flowshop scheduling with multiple assembly factories and make-to-order delivery. *Swarm Evol. Comput.* **97**, 101996 (2025) <https://doi.org/10.1016/j.swevo.2025.101996>
- [31] Zhao, C., Wu, L., Tan, W., Zuo, C.: A knowledge-based two-population optimization algorithm for distributed heterogeneous assembly permutation flowshop scheduling with batch delivery and setup times. *Swarm Evol. Comput.* **97**, 102035 (2025) <https://doi.org/10.1016/j.swevo.2025.102035>
- [32] Zhao, F., Yang, T., Xu, T., Zhu, N., Liang, H.: A bayesian-based hyper-heuristic algorithm for the integrated scheduling of distributed production assembly and delivery problem. *Appl. Soft Comput.* **187**, 114313 (2026) <https://doi.org/10.1016/j.asoc.2025.114313>
- [33] Song, H.-B., Lin, J.: A genetic programming hyper-heuristic for the distributed assembly permutation flow-shop scheduling problem with sequence dependent setup times. *Swarm Evol. Comput.* **60**, 100807 (2021) <https://doi.org/10.1016/j.swevo.2020.100807>
- [34] Huang, J., Gu, X.: Distributed assembly permutation flow-shop scheduling problem with sequence-dependent set-up times using a novel biogeography-based optimization algorithm. *Eng. Optim.* **54**(4), 593–613 (2022) <https://doi.org/10.1080/0305215X.2021.1886289>

- [35] Yu, H., Gao, K., Li, Z., Duan, P.: Double-learning-strategy-based evolutionary algorithm for scheduling multiobjective distributed assembly permutation flowshops with setup time. *IEEE Trans. Syst. Man Cybern. Syst.* **55**(2), 925–935 (2025) <https://doi.org/10.1109/TSMC.2024.3490544>
- [36] Allahverdi, A., Gupta, J.N.D., Aldowaisan, T.: A review of scheduling research involving setup considerations. *Omega* **27**(2), 219–239 (1999) [https://doi.org/10.1016/S0305-0483\(98\)00042-5](https://doi.org/10.1016/S0305-0483(98)00042-5)
- [37] Allahverdi, A., Allahverdi, M.: The fourth comprehensive review of scheduling problems with setup times. *Eur. J. Oper. Res.* **335**(1), 1–30 (2026) <https://doi.org/10.1016/j.ejor.2026.03.041>
- [38] Ruiz, R., Stützle, T.: An iterated greedy heuristic for the sequence dependent setup times flowshop problem with makespan and weighted tardiness objectives. *Eur. J. Oper. Res.* **187**(3), 1143–1159 (2008) <https://doi.org/10.1016/j.ejor.2006.07.029>
- [39] Navaei, J., Fatemi Ghomi, S.M.T., Jolai, F., Mozdgir, A.: Heuristics for an assembly flow-shop with non-identical assembly machines and sequence dependent setup times to minimize sum of holding and delay costs. *Comput. Oper. Res.* **44**, 52–65 (2014) <https://doi.org/10.1016/j.cor.2013.10.008>
- [40] Karabulut, K., Öztö, H., Kizilay, D., Tasgetiren, M.F., Kandiller, L.: An evolution strategy approach for the distributed permutation flowshop scheduling problem with sequence-dependent setup times. *Comput. Oper. Res.* **142**, 105733 (2022) <https://doi.org/10.1016/j.cor.2022.105733>
- [41] He, X., Pan, Q.-K., Gao, L., Neufeld, J.S., Gupta, J.N.D.: Historical information based iterated greedy algorithm for distributed flowshop group scheduling problem with sequence-dependent setup times. *Omega* **123**, 102997 (2024) <https://doi.org/10.1016/j.omega.2023.102997>
- [42] Karaboga, D.: An idea based on honey bee swarm for numerical optimization. Technical Report TR06, Erciyes University, Engineering Faculty, Computer Engineering Department, Kayseri, Turkey (2005)
- [43] Gorkemli, B., Kaya, E., Karaboga, D., Akay, B.: A review on the versions of artificial bee colony algorithm for scheduling problems. *J. Comb. Optim.* **49**(4), 62 (2025) <https://doi.org/10.1007/s10878-025-01296-1>
- [44] Zuo, Y., Wang, P., Fan, Z., Li, M., Guo, X., Gao, S.: Minimizing fuzzy makespan in a distributed assembly flow shop by using an efficient artificial bee colony algorithm. *J. Intell. Fuzzy Syst.* **45**(4), 7025–7046 (2023) <https://doi.org/10.3233/JIFS-230592>
- [45] Jiang, Z., Meng, L., Ren, Y., Zhang, B., Zhang, C.: Novel mathematical models

- and hybrid algorithms for distributed assembly hybrid flow shop scheduling problem. *Digital Twin* **2**(4), 2576265 (2025) <https://doi.org/10.1080/27525783.2025.2576265>
- [46] Wang, J., Lei, D., Cai, J.: An adaptive artificial bee colony with reinforcement learning for distributed three-stage assembly scheduling with maintenance. *Appl. Soft Comput.* **117**, 108371 (2022) <https://doi.org/10.1016/j.asoc.2021.108371>
 - [47] Wang, J., Tang, H., Lei, D.: A q-learning artificial bee colony for distributed assembly flow shop scheduling with factory eligibility, transportation capacity and setup time. *Eng. Appl. Artif. Intell.* **123**, 106230 (2023) <https://doi.org/10.1016/j.engappai.2023.106230>
 - [48] Li, D., Gao, K., Duan, P., Suganthan, P.N., Wu, N.: Reinforcement learning assisting artificial bee colony algorithm for scheduling distributed assembly flowshops with batch delivery. *IEEE Trans. Syst. Man Cybern. Syst.* **55**(12), 9295–9308 (2025) <https://doi.org/10.1109/TSMC.2025.3613727>
 - [49] Li, D., Gao, K., Yin, L., Suganthan, P.N., Wu, N., Zhao, L.: Tri-objective distributed assembly flow shop scheduling with batch delivery: Indicator-driven and reinforcement learning-enhanced artificial bee colony algorithms. *Eng. Appl. Artif. Intell.* **177**, 114943 (2026) <https://doi.org/10.1016/j.engappai.2026.114943>
 - [50] Yu, H., Gao, K.-Z., Ma, Z.-F., Pan, Y.-X.: Improved meta-heuristics with q-learning for solving distributed assembly permutation flowshop scheduling problems. *Swarm Evol. Comput.* **80**, 101335 (2023) <https://doi.org/10.1016/j.swevo.2023.101335>
 - [51] Naderi, B., Ruiz, R.: The distributed permutation flowshop scheduling problem. *Comput. Oper. Res.* **37**(4), 754–768 (2010) <https://doi.org/10.1016/j.cor.2009.06.019>
 - [52] Cheng, B.-W., Chang, C.-L.: A study on flowshop scheduling problem combining Taguchi experimental design and genetic algorithm. *Expert Syst. Appl.* **32**(2), 415–421 (2007) <https://doi.org/10.1016/j.eswa.2005.12.002>
 - [53] García, S., Molina, D., Lozano, M., Herrera, F.: A study on the use of non-parametric tests for analyzing the evolutionary algorithms' behaviour: a case study on the cec'2005 special session on real parameter optimization. *J. Heur.* **15**(6), 617–644 (2009) <https://doi.org/10.1007/s10732-008-9080-4>