Journal of Administrative and Economic Sciences • Vol 18, Issue 1 (2025) • p 737-756

Optimizing Distributed Flowshop Scheduling with an Artificial Bee Colony Algorithm: Implications for Operations Management

Mansour Eddaly

Departement of Operations Management, College of Business and Economics, Qassim University • Buraydah 52571• Saudi Arabia.

m.eddaly@qu.edu.sa

Abstract:

The Distributed Assembly Permutation Flowshop Scheduling Problem (DAPFSP) extends classical flowshop scheduling by integrating distributed production stages with a final centralized assembly stage. This problem is NP-hard and arises in many real-world applications such as automotive, electronics, and aerospace manufacturing, where products are composed of multiple jobs processed across several factories before final assembly. In this paper, we propose an Artificial Bee Colony (ABC) algorithm specifically adapted to the DAPFSP. The algorithm incorporates six neighborhood structures and six local search procedures based on swap and insert moves, applied at the job, product, and assembly levels, to balance exploration and exploitation effectively. Extensive computational experiments on 810 benchmark instances from the literature demonstrate the competitiveness of the proposed approach. The results show that the ABC algorithm is able to improve 481 best-known solutions, achieving average negative deviations with respect to reference methods while maintaining reasonable computational times. Compared with existing algorithms, Furthermore, the ABC algorithm achieves average improvement margins of 9%–48% across various problem scales, decreasing the average deviation by 0.17 on average and by up to 0.73 for large-size instances.

Keywords: Distributed Assembly Permutation Flowshop Scheduling Problem, Swarm intelligence, Artificial Bee Colony (ABC) algorithm, Local search.

JEL Classification codes: C61, C65, L23, O32.

1. Introduction

The Distributed Assembly Permutation Flowshop Scheduling Problem (DAPFSP) is an advanced scheduling model that integrates distributed production with centralized assembly operations. In this problem, jobs are first processed in multiple identical factories, each operating under the permutation flowshop constraint, where all machines follow the same job sequence. Once processing is completed, the resulting parts or components are transported to a final assembly stage, where they are merged into finished products. This dual-level decision framework requires determining not only the assignment and sequencing of jobs within distributed factories but also the synchronization of operations at the assembly stage. The primary objective is typically to minimize the makespan, which depends on both the latest completion time across distributed factories and the efficiency of assembly scheduling (Hatami et al., 2013; Liu et al., 2016; Lin & Zhang, 2016; Zhang et al., 2024).

Hatami et al. (2013) were the first to introduce the concept of distributed manufacturing into assembly scheduling, formally defining the DAPFSP. Since then, the model has been recognized as highly relevant to real-world applications in manufacturing networks where components are produced across different plants and centrally assembled, such as in the automotive, aerospace, and electronics industries (Li et al., 2016). More recently, Zhang et al. (2024) extended the DAPFSP by developing a more realistic variant tailored to the automobile manufacturing supply chain. In their formulation, each component of the final product is composed of multiple subparts, which are first produced in component-manufacturing factories and then assembled into larger components before final assembly. This structure reflects the actual complexity of automotive production, where major components such as engines and axles are themselves assemblies of smaller parts.

From a computational perspective, the problem is particularly challenging: since the classical permutation flowshop scheduling problem is already NP-hard for three or more machines (Garey et al., 1976), the distributed and assembly extensions further increase their difficulty. As a result, the DAPFSP is also NP-hard and generally requires the use of heuristic and metaheuristic approaches to obtain high quality solutions for large scale instances (Li et al., 2016).

In this paper, we address DAPFSP by proposing an Artificial Bee Colony (ABC) algorithm, and we evaluate its effectiveness through extensive computational experiments and statistical analyses. The main contributions of this paper can be summarized as follows:

- Algorithmic design: We propose a hybrid Artificial Bee Colony (ABC) algorithm specifically adapted to the hierarchical structure of the DAPFSP, integrating exploration and exploitation mechanisms across multiple decision levels.
- Neighborhood innovation : We introduce six neighborhood structures based on swap and insert moves applied at the job, product, and assembly levels, enabling adaptive search diversification and intensification.
- Local Search framework: We develop six complementary local search procedures, also based on swap and insert operators, to refine solutions at different hierarchy layers using a first-improvement strategy.

- Comprehensive evaluation : We conduct extensive computational experiments on 810 benchmark instances from the literature, covering diverse problem scales and configurations.

The remainder of this paper is organized as follows. Section 2 reviews the related literature on distributed and assembly flowshop scheduling problems. Section 3 presents the problem formulation and its main features. Section 4 describes the proposed hybrid Artificial Bee Colony (ABC) algorithm in detail. Section 5 reports and analyzes the computational results obtained on benchmark instances. Finally, Section 6 concludes the paper and highlights directions for future research.

2. Literature review

This section provides an overview of the main studies related to distributed and assembly flowshop scheduling problems. The objective is to identify the key methodological developments and highlight the research gaps that motivate the design of the proposed Artificial Bee Colony (ABC) algorithm.

Hatami et al. (2013) made a seminal contribution by formally introducing the DAPFSP, which extends the classical and distributed flowshop models by incorporating a final assembly stage. They proposed a mixed-integer linear programming formulation to capture both the distributed production environment, where jobs are processed in multiple identical factories under the permutation constraint, and the subsequent synchronization required at the assembly stage. Recognizing the NP-hard nature of the problem, the authors developed constructive heuristics and a Variable Neighborhood Descent (VND) metaheuristic specifically designed for the structure of the DAPFSP. Through extensive computational experiments and statistical analyses, they demonstrated that the VND significantly outperforms simpler heuristics in terms of solution quality. Beyond methodological advances, their work highlights the practical relevance of the DAPFSP to modern supply chains, where components are manufactured across distributed plants and integrated into centralized assembly facilities, providing a more realistic framework for production and operations management.

Hatami et al. (2014) focused on the DAPFSP and proposed two simple yet effective constructive algorithms to generate feasible schedules. Unlike metaheuristics, which can be computationally demanding, their constructive approaches provide lightweight, fast methods to obtain initial solutions for the DAPFSP. The algorithms are based on intuitive sequencing and job-assignment rules adapted to handle both the distributed production stage and the final assembly stage. The study complements their earlier formal introduction of the DAPFSP by offering practical heuristics that can serve as stand-alone solution methods for small to medium-sized instances or as initial solution generators for more sophisticated metaheuristics. Their results demonstrate that these constructive heuristics, despite their simplicity, can produce competitive schedules and are valuable for balancing solution quality with computational efficiency.

Hatami et al. (2015) extended the study of the DAPFSP by incorporating sequence-dependent setup times, a realistic feature often encountered in manufacturing environments where changeovers between jobs are not negligible. To address the

increased complexity introduced by setup times, the authors developed both constructive heuristics and advanced metaheuristics, including a Variable Neighborhood Search (VNS) and a Genetic Algorithm (GA) adapted to the problem. These methods were designed to handle both the distributed production scheduling decisions and the synchronization requirements of the final assembly stage while accounting for the added setup constraints. Extensive computational experiments on a wide range of benchmark instances showed that the proposed metaheuristics, particularly the VNS, achieved high-quality solutions and clearly outperformed simple heuristics. This work represents a significant step in bridging the gap between theoretical scheduling models and the practical realities of production and assembly systems, where setup times play a crucial role in efficiency.

Li et al. (2015) tackled the DAPFSP by designing a specialized GA. Their approach focused on encoding both the assignment of jobs to distributed factories and the sequencing decisions within each factory, as well as the coordination with the final assembly stage. To improve the efficiency of the GA, they introduced problem-specific genetic operators, including crossover and mutation strategies, aimed at preserving feasible schedules and enhancing exploration of the solution space. The algorithm was tested on benchmark instances of the DAPFSP, and the experimental results demonstrated that the GA achieved competitive solution quality within reasonable computational times. The study showed the potential of evolutionary computation for solving complex distributed and assembly scheduling problems, and it contributed an alternative metaheuristic approach to complement the heuristics and neighborhood search strategies previously proposed in the literature.

Wang and Wang (2015) proposed an innovative Estimation of Distribution Algorithm-based Memetic Algorithm (EDA-MA) for solving the DAPFSP. Their approach integrates the global search ability of estimation of distribution algorithms, which build probabilistic models to guide solution generation, with the local refinement capability of memetic algorithms. To make the method effective for the DAPFSP, they introduced specialized solution representations, problem-oriented local search operators, and adaptive learning mechanisms that capture both job assignment to distributed factories and synchronization with the final assembly stage. The algorithm was tested on benchmark instances and compared with state-of-the-art metaheuristics, showing superior performance in terms of makespan minimization and robustness across problem scales. This study significantly contributed to the literature by demonstrating how probabilistic model–based evolutionary computation, when combined with local search, can provide highly effective solutions for complex distributed and assembly scheduling problems.

Liu et al. (2016) proposed a Variable Neighborhood based Memetic Algorithm (VNMMA) to solve the DAPFSP. Their approach combined the exploration strength of VNS with the exploitation capability of memetic algorithms, which integrate evolutionary computation with local search. A specialized encoding scheme was designed to represent job assignments, sequencing in distributed factories, and synchronization at the assembly stage. The algorithm also employed problem-specific neighborhood structures and local improvement operators to intensify the search around

promising solutions. Computational experiments on benchmark instances demonstrated that the proposed VNMMA was able to produce high-quality solutions and outperformed traditional evolutionary algorithms in both accuracy and robustness. The paper contributed by showing that hybrid metaheuristics, which balance diversification and intensification, are particularly effective for tackling the complexity of the DAPFSP.

Lin and Zhang (2016) developed an effective hybrid biogeography-based optimization (HBBO) algorithm to address the DAPFSP. Their approach combined the principles of biogeography-based optimization (BBO) with problem-specific hybridization strategies, such as customized initialization, adaptive migration, and local search mechanisms, to enhance both exploration and exploitation of the search space. The HBBO was designed to simultaneously optimize job assignment across distributed factories, sequencing under the permutation flowshop constraint, and synchronization at the assembly stage. Extensive computational experiments on benchmark instances demonstrated that the HBBO significantly outperformed standard BBO and other baseline heuristics in terms of solution quality and robustness. This work contributed to the DAPFSP literature by showing the effectiveness of bio-inspired optimization algorithms and highlighting the potential of hybridization strategies for solving large-scale and complex scheduling problems.

Lin et al. (2017) proposed a backtracking search hyper-heuristic (BSHH) to solve the DAPFSP. Unlike traditional metaheuristics that rely on a single search strategy, their hyper-heuristic framework adaptively selects and combines different low-level heuristics to balance exploration and exploitation during the search process. The backtracking search mechanism was employed to dynamically guide the heuristic selection and parameter adaptation, enhancing the algorithm's robustness across different problem instances. The proposed BSHH simultaneously handled job assignment to distributed factories, sequencing under the permutation flowshop constraint, and synchronization at the assembly stage. Extensive computational experiments demonstrated that the method achieved competitive or superior performance compared to state-of-the-art metaheuristics, while offering stronger generalization and stability. This study contributed to the DAPFSP literature by introducing hyper-heuristic search as an effective paradigm, highlighting its potential for addressing highly complex scheduling problems with multiple decision layers.

Zhang et al. (2024) advanced the study of the DAPFSP by proposing a more realistic formulation focused on the automobile manufacturing supply chain. In their model, each component of the final product is composed of multiple subparts, which are first produced in distributed component-manufacturing factories, then assembled into larger components before final assembly, thereby reflecting the hierarchical complexity of real automotive production. To solve this challenging NP-hard problem, the authors designed three enhanced variants of the Social Spider Optimization (SSO) algorithm, incorporating hybrid local search strategies, restart mechanisms, and adaptive probability selection to balance exploration and exploitation. They also introduced a three-level solution representation and novel initialization methods to better capture the hierarchical structure of the problem. Through extensive computational experiments on

a large set of extended benchmark instances, the proposed algorithms—particularly the improved SSO with restart and adaptive selection—were shown to achieve superior performance compared to baseline methods. Their work not only provides effective heuristics for large-scale DAPFSP instances but also offers valuable insights into managing distributed production and assembly coordination in complex supply chains.

Table 1: Comparative summary of studies on the Distributed Assembly Permutation Flowshop Scheduling Problem (DAPFSP)

Study	Solution methods	Contributions
Hatami et al. (2013)	MILP, VND	First formal definition of DAPFSP; mixed-integer formulation; VND metaheuristic integrating distributed and assembly coordination
Hatami et al. (2014)	Constructive heuristics	Fast constructive algorithms for feasible schedules; lightweight and efficient
Hatami et al. (2015)	VNS, GA	Considers sequence-dependent setup times; hybrid metaheuristics improve quality
Li et al. (2015)	GA	Specialized encoding and genetic operators for DAPFSP
Wang & Wang (2015)	EDA-MA	Combines estimation of distribution and memetic search; robust performance
Liu et al. (2016)	VNMMA	Combines VNS and memetic algorithm; balances exploration and exploitation
Lin & Zhang (2016)	HBBO	Bio-inspired hybrid with adaptive migration and local search
Lin et al. (2017)	BSHH	Hyper-heuristic combining multiple search strategies; robust and generalizable
Zhang et al. (2024)	Hybrid SSO variants	Multi-level hierarchical model for automotive supply chain; adaptive restart and local search

As summarized in Table 1, previous research on the DAPFSP has primarily focused on adapting classical metaheuristics such as VND, VNS, GA, and BBO to the distributed and assembly scheduling framework. While these studies have advanced the field, most existing methods face challenges related to limited neighborhood diversity, weak integration between exploration and exploitation, and reduced scalability on large instances. Moreover, few algorithms explicitly exploit the hierarchical structure of the

problem, comprising job-level, product-level, and assembly-level decision, within their search mechanisms.

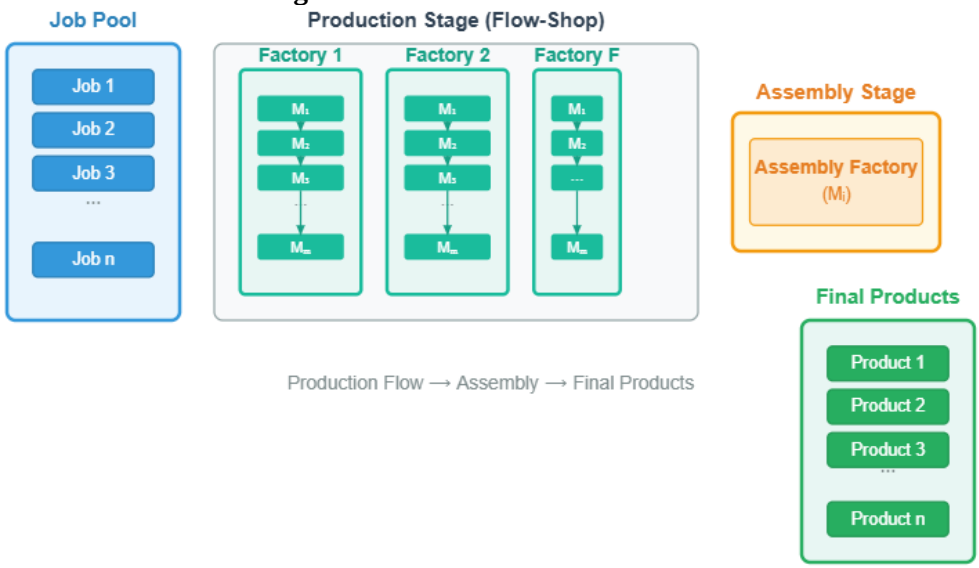
3. Problem description

The DAPFSP is a two-stage scheduling model consisting of a production stage followed by a final assembly stage (Hatami et al., 2013).

- *Production stage*
  - Production is carried out in a set F of f identical factories or production centers.
  - A set J of n jobs must be scheduled across these factories, each of which is modeled as a classical permutation flowshop with a set M of machines where :
    - The processing times denoted by p_{ij} where $i \in M$ refers to the machine index and $j \in J$ refers to the job index.
    - Jobs cannot be transferred from one factory to another once processing has begun.
    - All jobs are independent and available at time zero.
    - Each machine processes at most one job at a time,
    - Each job is processed by exactly one machine at a time ;
    - Preemption is not allowed, meaning operations must be completed without interruption.
    - Setup times as well as transportation times are included in the processing times.
- *Assembly stage*
  - A single assembly machine M_A, on which a set P of k final products is assembled.
  - Each product $h \in P$ has a predefined assembly program consisting of a subset $J_h \subseteq J$ of jobs. If $|J_h|$ denotes the number of jobs required for product h, then $\sum_{h=1}^{k} |J_h| = n$.
  - Assembly of product h can only begin once all jobs in J_h have been completed in the production stage.

The objective of the DAPFSP is to assign jobs to factories and determine job sequences on all machines to minimize the makespan at the assembly stage.

Figure 1: Illustration of the DAPFSP



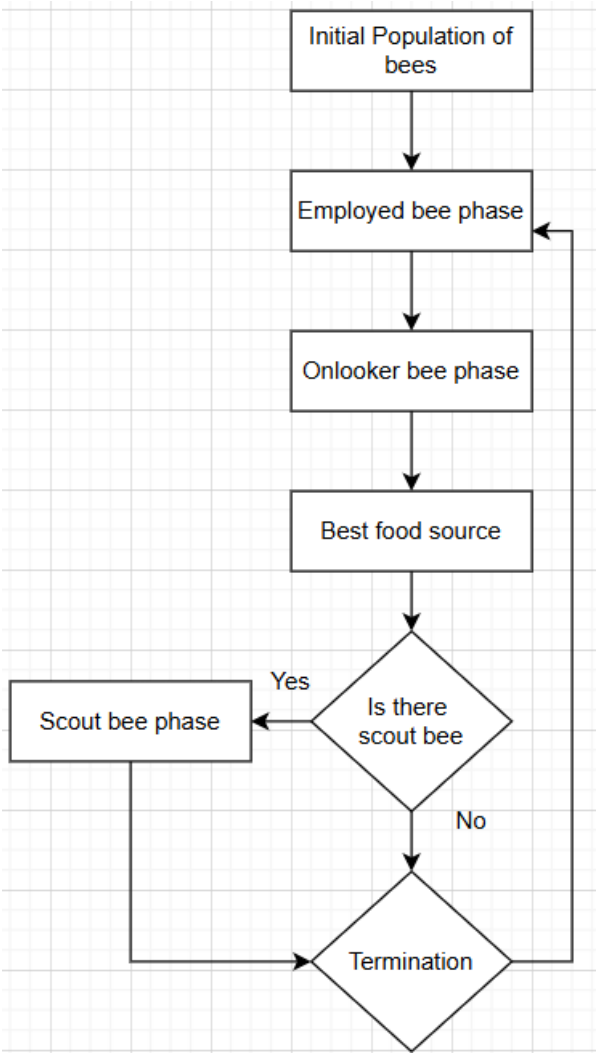
4. The proposed hybrid Artificial Bee Colony (ABC) algorithm

To solve the Distributed Assembly Permutation Flowshop Scheduling Problem (DAPFSP), we propose an Artificial Bee Colony (ABC)–based algorithm specifically adapted to the characteristics of the problem. The ABC framework divides the search process into distinct phases, each simulating a particular role of bees in a colony: initialization of food sources (solutions), exploration by employed bees, exploitation by onlooker bees, and diversification through scout bees. Within each phase, solution structures are modified and evaluated to balance exploration of new regions of the search space with exploitation of promising areas already identified. To further enhance solution quality, problem-specific local search procedures are integrated into the algorithm. The overall structure of the proposed ABC algorithm is outlined in Appendix A, where the complete pseudocode is provided for clarity and reproducibility. The following subsections present a detailed explanation of each algorithmic component, including the initialization of solutions, the behavior of employed, onlooker, and scout bees, and the set of local search operators integrated into the framework.**The basic algorithm**

The Artificial Bee Colony (ABC) algorithm, introduced by Karaboga (2005), is a swarm intelligence-based metaheuristic inspired by the foraging behavior of honeybee colonies. In this algorithm, artificial bees are divided into three groups: employed bees, onlooker bees, and scout bees. Employed bees explore the neighborhood of their current food sources, which represent candidate solutions, and share the quality of these sources with onlookers. Onlooker bees then probabilistically select food sources based on their fitness and further exploit promising regions of the search space. When a food source cannot be improved for a certain number of trials, it is abandoned, and the corresponding employed bee becomes a scout that randomly generates a new source, thus ensuring exploration of unexplored regions. The search process iterates through these phases, employed bee, onlooker bee, and scout bee, until a termination condition such as a maximum number of iterations or a satisfactory solution quality is reached.

With its simple structure, few control parameters, and effective balance between exploration and exploitation, the ABC algorithm has been widely applied to both continuous and combinatorial optimization problems. Figure 2 illustrates the flow diagram of the basic ABC algorithm.

Figure 2: the basic ABC algorithm



4.1. Encoding solutions

The assembly sequence is represented by a permutation π_A of size $| P |$, where the l^{th} element in the permutation denotes the product located in position l. Each factory $f \in F$ is represented by the sequence of products assigned to that factory. Since the jobs belonging to a given product are never split across different factories, each factory f is encoded by a permutation of the jobs that compose its assigned products. Consequently, a feasible solution to the DAPFSP is defined by one assembly sequence together with $| F |$ job sequences, each corresponding to a factory.

Example:

Consider an instance with three products, two factories, and ten jobs. Product P_1 consists of jobs $\{J_1, J_3, J_6\}$, product P_2 includes jobs $\{J_4, J_8, J_9\}$, and product P_3 is composed of jobs $\{J_2, J_5, J_7, J_{10}\}$. An example of an assembly sequence is $\pi_A = \{P_3, P_1, P_2\}$, meaning that product P_3 is assembled first, followed by product P_1, and finally product P_2. Suppose that factory F_1 is assigned products P_2 and P_3 in this order, while factory F_2 is assigned product P_1. A possible representation of the job sequences for these factories is $\{J_4, J_8, J_9, J_2, J_5, J_7, J_{910}\}$ for F_1 and $\{J_1, J_3, J_6\}$. It should be noted that the order of jobs in each factory sequence corresponds to their processing order across the machines.

4.2. Initial population of bees

The initial population is composed of B feasible solutions. In the first solution, the assembly sequence is generated using the Shortest Assembly Time rule, where products are ordered in non-decreasing assembly times, so that the product with the shortest assembly time appears first, and so on. In the remaining solutions, the assembly sequence is generated randomly. The job sequences for each factory are then constructed as follows: according to the assembly order, each product is assigned to one of the factories. When the number of products exceeds the number of factories, the last $k - |F|$ products in the assembly sequence are randomly assigned to factories, one by one. For instance, if $k = 6$ and $F = 3$, then each factory should be assigned two products.

4.3. The employed bee phase

In this phase, a new food source (solution) is generated in the neighborhood of each employed bee. The current food source is replaced by the new one if the latter yields a better fitness value. In our algorithm, at each iteration, one of the following six neighborhood structures is applied to create a neighbor solution:

- Structure 1: randomly select two products in the assembly sequence and swap their positions.
- Structure 2: randomly select one product in the assembly sequence and insert it into a different random position.
- Structure 3: for each factory, randomly select two products and swap their positions.
- Structure 4: for each factory, randomly select one product and insert it into a different random position.
- Structure 5: for each factory, randomly select two jobs belonging to products and swap their positions.
- Structure 6: for each factory, for each product, randomly select one job and insert it into a different random position.

The choice of these six neighborhood structures was guided by the hierarchical characteristics of the DAPFSP, which involves three interdependent decision levels: job sequencing within factories, product assignment across factories, and assembly sequencing at the final stage. Structures 1–2 explore the assembly level, promoting

global diversification by altering the order of assembled products. Structures 3–4 operate at the factory–product level, refining product distribution and internal sequencing among distributed production units. Finally, Structures 5–6 act at the job level, performing fine-grained adjustments within each product's internal sequence to intensify the search around promising regions. This multi-level design allows the algorithm to balance exploration and exploitation dynamically, ensuring that modifications at different hierarchy layers jointly contribute to improved makespan performance while preserving solution feasibility. The complete pseudocode describing these neighborhood structures is provided in Appendix B.

To further improve solution quality, a local search procedure is applied with probability $prob$; the details of this procedure are presented later. If the newly generated solution is better than the current one, it replaces it. Moreover, if the new solution is superior to the best solution obtained so far, the global best solution is updated accordingly.

4.4. The onlooker bee phase

An onlooker bee evaluates the solutions generated by the employed bees and selects one according to a probability calculated from the fitness values using the roulette wheel selection method. Once a solution is selected, the onlooker generates a neighbor solution in the same manner as in the employed bee phase; however, in this case the local search procedure is always applied, without probability. The current solution is then replaced by the candidate solution only if the latter achieves better fitness value.

4.5. The scout bee phase

After the search processes of the employed and onlooker bees are completed, the ABC algorithm checks whether any food sources have been exhausted. In such cases, scout bees are introduced to explore new food sources through random search. In this way, the scouts ensure diversification within the ABC algorithm by directing the search toward unexplored regions of the solution space. In our proposed algorithm, diversification is further reinforced by discarding the worst solution from the current population and replacing it with the best solution found so far.

4.6. The local search procedures

In our algorithm, six local search procedures are employed, relying on swap and insert moves applied at different levels: the sequence of jobs within each product in each factory, the sequence of products in each factory, and the assembly sequence. All procedures follow the first-improvement rule, meaning that whenever a move produces a better solution than the current one, the latter is immediately replaced. Each of the six procedures is applied iteratively until no further improvement can be achieved.

Let $LSinsert\_job$ be the local search procedure designed to improve the sequence of jobs within each product in each factory using insert moves. For each factory f, and for each product assigned to that factory, a job is selected and inserted into all possible alternative positions. Similarly, $LSswap\_job$ denotes the local search based on swap moves applied to jobs in each factory, where all possible pairwise swaps of job positions are performed within the assigned products. The procedures $LSinsert\_product$ and

$LSswap\_product$ are defined as insert and swap moves applied to the product sequence in each factory, respectively. Finally, $LSinsert\_assembly$ and $LSswap\_assembly$ correspond to insert and swap moves applied to the assembly sequence.

The adoption of these six local search procedures is motivated by the multi-layered decision structure of the DAPFSP. Each level—job, product, and assembly—has a distinct influence on the overall makespan, requiring targeted improvement mechanisms. The procedures LSinsert_job and LSswap_job refine local sequencing decisions within individual products in each factory, enabling the correction of small mis-orderings that directly affect processing efficiency. The LSinsert_product and LSswap_product procedures focus on the factory-level sequencing of products, which is critical for balancing workloads among distributed production units and improving synchronization before assembly. Finally, LSinsert_assembly and LSswap_assembly operate at the global assembly stage, where small re-orderings can significantly reduce idle times between component arrivals.

The combination of insert and swap operators at all three levels ensures both exploration and intensification: insert moves enable broader re-arrangements with moderate disruption (diversification), while swap moves fine-tune promising sequences (intensification). Using the first-improvement rule allows rapid convergence toward high-quality local optima without excessive computational overhead. Altogether, these procedures provide a coherent local search framework aligned with the hierarchical and distributed nature of the DAPFSP, ensuring that improvements at each level contribute effectively to overall makespan reduction. A detailed pseudocode of all local search procedures is presented in Appendix C for clarity and reproducibility.

5. Computational results

The proposed ABC algorithm was implemented in C++, and all experiments were conducted under Windows 10 on a laptop computer equipped with a processor 13th Gen Intel(R) Core(TM) i7-1360P (2.20 GHz) and 16 GB of memory. To evaluate the performance of the proposed algorithm against comparative methods, we used the set of 810 large benchmark instances generated by Hatami et al. (2013), with configurations defined by $|F| \in \{4,6,8\}$ factories, $|M| \in \{5,10,20\}$ machines per factory, $|J| \in \{100,200,500\}$ jobs, and $|P| \in \{30,40,50\}$ products. The processing times of jobs at the production stage were drawn from a uniform distribution over [1,99], while the assembly processing time of each product h was defined within the range $[N_h \mid ,99 \times \mid N_h \mid]$, where $\mid N_h \mid$ is the number of jobs required for product hhh. The algorithm parameters were set experimentally: the bee population size was fixed at $P = 60$, the probability of applying local search to each employed bee was set to $prob = 0.02$, the number of onlookers was 60, and the number of scouts was 3. As a stopping criterion, a time limit of 24 seconds was used for instances with fewer than 500 jobs, and 60 seconds for instances with $|J| = 500$.

For each instance, the proposed algorithm was executed five times, and the minimum deviation (Δ_{min}) and average deviation (Δ_{avg}) were reported according to the relative percentage deviation (RPD), defined as:

$$RPD = 100 \times \frac{a\,lg\,o - best}{best}$$

where *algo* is the solution obtained by the algorithm and *best* is the best-known solution. The performance of the proposed algorithm is evaluated against the five Variable Neighborhood Descent (VND) methods of Hatami et al. (2013). The computational results are summarized in Table 2 and Table 3.

Table 2: Computational results in terms of RPD

		The algorithms proposed by Hatami et al.(2013)						**ABC algorithm**	
Instances		VND_{H11}	VND_{H12}	VND_{H21}	VND_{H22}	VND_{H31}	VND_{H32}	Δ_{min}	Δ_{avg}
Factories	4	0.06	0.03	0.05	0.01	0.05	0.01	-0,43	-0,09
	6	0.03	0.01	0.02	0.00	0.02	0.00	-0,54	-0,25
	8	0.02	0.00	0.01	0.00	0.01	0.00	-0,51	-0,22
Products	30	0.03	0.01	0.04	0.01	0.04	0.01	-0,70	-0,29
	40	0.04	0.02	0.02	0.01	0.02	0.01	-0,69	-0,36
	50	0.04	0.01	0.02	0.00	0.02	0.00	-0,12	0,07
Jobs	100	0.05	0.02	0.03	0.01	0.03	0.01	-0,67	-0,44
	200	0.03	0.01	0.02	0.00	0.02	0.00	-0,73	-0,31
	500	0.03	0.01	0.03	0.01	0.03	0.01	0,10	0,36
mean		**0.04**	**0.01**	**0.01**	**0.01**	**0.03**	**0.01**	**-0.48**	**-0.17**

Table 2 reports the results obtained in terms of RPD. Our algorithm improves 481 best-known solutions out of the 810 benchmark instances. On average, across all tested instances, both the minimum and average deviations are non-positive, with respective values of −0.48% and −0.17%, whereas the best deviations achieved by the VND algorithms are 0.01%. In terms of minimal deviations, negative values are obtained for all instance classes except those with $|J| = 500$. A similar observation holds for the average deviations, where only two classes, instances with 50 products and instances with 500 jobs, exhibit positive deviations. These results indicate that, in terms of solution quality, the proposed ABC algorithm clearly outperforms the VND algorithms of Hatami et al. (2013). This improvement can be explained by the stronger exploitation ability of our approach. Specifically, the proposed ABC algorithm integrates six local search procedures that optimize all components of the problem, including job sequences within each product, product sequences in each factory, and the assembly sequence. Furthermore, while the compared VND algorithms explore only insert moves, our method employs both insert and swap moves, thereby enriching the search process and enhancing performance.

Figure 3 compares the performance of the proposed ABC algorithm with the six VNDH variants for different numbers of factories. The ABC method consistently achieves negative deviation values, indicating clear improvements over all benchmark algorithms. The largest gain occurs for six factories, where both the average and

minimum ABC results show the greatest reduction. Although the improvement margin slightly decreases for eight factories, the ABC algorithm remains superior overall, confirming its robustness and scalability in distributed production settings.

Figure 3: Performance vs. number of factories

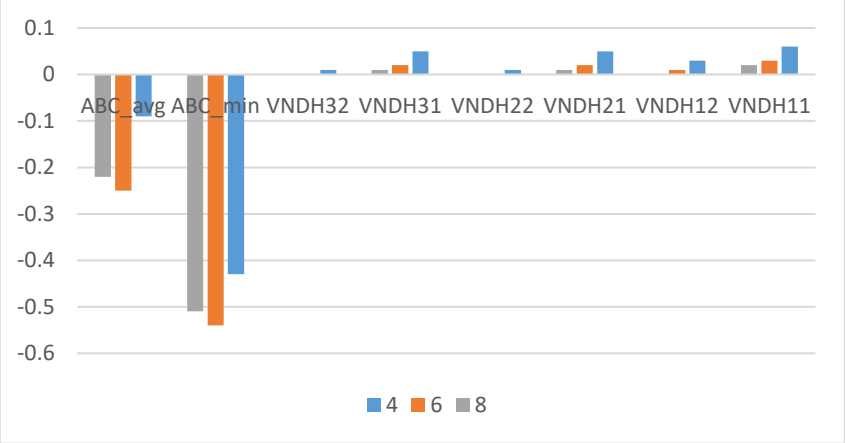


Figure 4 compares the performance of the proposed ABC algorithm for different numbers of products (30, 40, and 50). The ABC method consistently yields negative deviation values, demonstrating clear improvements over all benchmark algorithms. The largest gain is observed for 40 products, where both the average and minimum ABC results show the greatest reduction. As the number of products increases to 50, the improvement margin slightly narrows but remains substantial, confirming that the ABC algorithm maintains superior performance and scalability across different product scales.

Figure 4: Performance vs. number of products

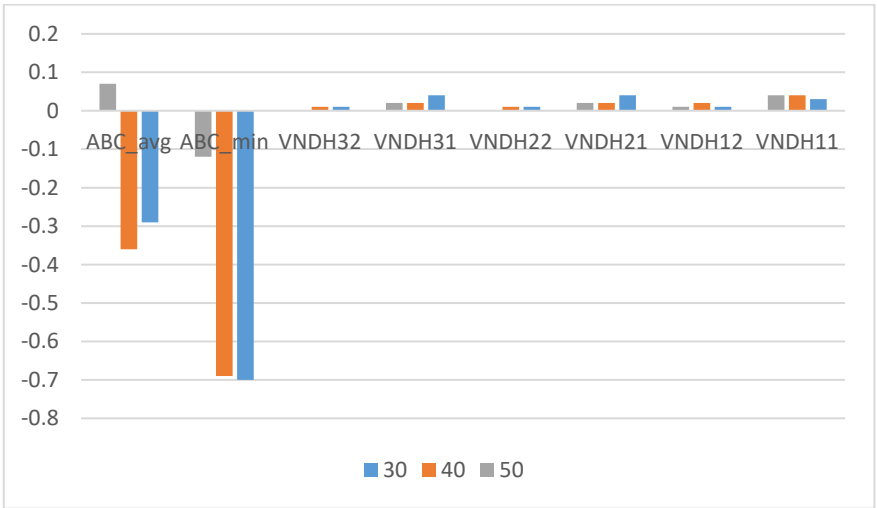


Figure 5 compares the performance of the proposed ABC algorithm across different job sizes (100, 200, and 500 jobs). The ABC algorithm consistently yields negative deviation values for 100 and 200 jobs, indicating substantial improvement over all benchmark algorithms. The most pronounced enhancement occurs at 200 jobs, where both the average and minimum ABC results reach their lowest deviation levels (around −0.6 to −0.7). For the largest instance size (500 jobs), the deviation becomes slightly positive, suggesting that problem complexity begins to challenge the algorithm's

exploration capacity. Nevertheless, the ABC method still performs competitively, demonstrating strong scalability and robustness across varying job sizes.

Figure 5: Performance vs. number of jobs

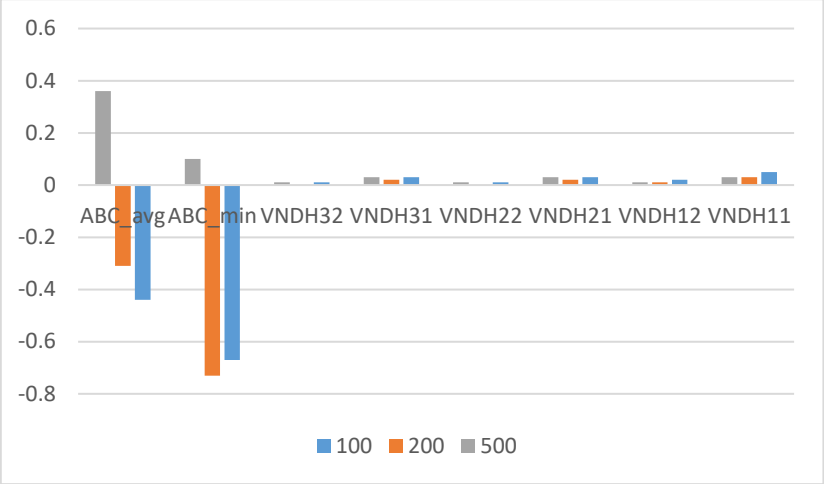


Table 3 presents the CPU times required to obtain the best solutions. It should be noted that these values correspond to the time needed to reach the best solution in each run. For the proposed algorithm, the average CPU time across all instances is 15.16 seconds, which is lower than that of VNDH32. As expected, instances with 100 jobs are the least time-consuming, while those with with $|J| = 500$ require the largest computational effort.

Table 3: CPU times

		The algorithms proposed by Hatami et al.(2013)						ABC algorithm
Instances		VND_{H11}	VND_{H12}	VND_{H21}	VND_{H22}	VND_{H31}	VND_{H32}	
Factories	4	4.39	6.79	2.90	7.67	2.55	42.87	17.42
	6	3.49	7.73	2.85	8.94	1.95	6.11	10.12
	8	3.26	9.56	1.86	10.21	1.83	20.64	11.14
Products	30	3.64	8.05	3.14	11.00	2.70	45.20	16.73
	40	3.59	7.12	2.45	8.05	1.96	5.54	14.10
	50	3.91	8.91	2.02	7.77	1.66	18.88	8.60
Jobs	100	1.09	2.84	0.27	0.72	0.24	0.43	0.60
	200	2.02	3.85	0.58	2.22	0.66	1.37	3.27
	500	8.03	17.39	6.76	23.88	5.41	67.81	54.48
mean		**3.71**	**8.03**	**2.54**	**8.94**	**2.11**	**23.20**	**15.16**

6. Conclusions

In this work, we proposed an Artificial Bee Colony (ABC) algorithm to address the Distributed Assembly Permutation Flowshop Scheduling Problem (DAPFSP), an NP-hard problem that integrates distributed production scheduling and centralized assembly. The proposed approach incorporates six neighborhood structures and six local search procedures, combining swap and insert moves at the job, product, and assembly levels. Computational experiments on the benchmark instances of Hatami et

al. (2013) demonstrated that the algorithm is highly competitive. In particular, it was able to improve 481 out of 810 best-known solutions, yielding negative deviations for most instance classes, and requiring less CPU time than some of the compared VND algorithms. These results confirm that the hybridization of ABC with multiple local search strategies strengthens exploitation and enhances solution quality.

Several directions for future research can be considered. First, adaptive strategies could be introduced to dynamically adjust the choice of neighborhood structures and local search operators based on instance characteristics or search progress. Second, hybridization with other metaheuristics, such as genetic algorithms or tabu search, may further improve diversification. Third, large-scale instances involving thousands of jobs and products, as well as heterogeneous factories and multi-assembly stages, should be explored to evaluate the scalability of the approach. Finally, practical extensions of the DAPFSP, such as sequence-dependent setup times, transportation delays between factories and assembly, or energy-efficient scheduling objectives, represent promising avenues to enhance the realism and applicability of the proposed algorithm in complex manufacturing systems.

References

Garey, M. R., Johnson, D. S., & Sethi, R. (1976). The complexity of flowshop and jobshop scheduling. *Mathematics of Operations Research, 1*(2), 117–129.

Hatami, S., Ruiz, R., & Andres-Romano, C. (2013). The distributed assembly permutation flowshop scheduling problem. *International Journal of Production Research, 51*(16), 4891–4903.

Hatami, S., Ruiz, R., & Andrés-Romano, C. (2014). Two simple constructive algorithms for the distributed assembly permutation flowshop scheduling problem. In *Proceedings of the 9th International Conference on Hybrid Artificial Intelligence Systems (HAIS 2014)*, pp. 139–145. Springer.

Hatami, S., Ruiz, R., & Andrés-Romano, C. (2015). Heuristics and metaheuristics for the distributed assembly permutation flowshop scheduling problem with sequence dependent setup times. *International Journal of Production Economics, 169*, 76–88.

Karaboga, D. (2005). *An idea based on honey bee swarm for numerical optimization*. Technical Report TR06, Erciyes University, Engineering Faculty, Computer Engineering Department, Kayseri, Turkey.

Li, J. Q., Pan, Q. K., & Duan, P. Y. (2016). An iterated greedy heuristic for solving the distributed assembly permutation flowshop scheduling problem. *Knowledge-Based Systems, 97*, 156–168.

Li, X., Zhang, X., Yin, M., & Wang, J. (2015). A genetic algorithm for the distributed assembly permutation flowshop scheduling problem. In *2015 IEEE Congress on Evolutionary Computation (CEC)*, pp. 3096–3101. IEEE.

Lin, J., & Zhang, S. (2016). An effective hybrid biogeography-based optimization algorithm for the distributed assembly permutation flow-shop scheduling problem. *Computers & Industrial Engineering, 97*, 128–136.

Lin, J., Wang, Z. J., & Li, X. (2017). A backtracking search hyper-heuristic for the distributed assembly flowshop scheduling problem. *Swarm and Evolutionary Computation, 36*, 124–135.

Liu, B., Wang, K., & Zhang, R. (2016). Variable neighborhood based memetic algorithm for distributed assembly permutation flowshop. In *2016 IEEE Congress on Evolutionary Computation (CEC)*, pp. 1682–1686. IEEE.

Shengluo Yang & Zhigang Xu, 2021. "The distributed assembly permutation flowshop scheduling problem with flexible assembly and batch delivery," International Journal of Production Research, Taylor & Francis Journals, vol. 59(13), pages 4053-4071, July.

Wang, S. Y., & Wang, L. (2015). An estimation of distribution algorithm-based memetic algorithm for the distributed assembly permutation flow-shop scheduling problem. *IEEE Transactions on Systems, Man, and Cybernetics: Systems, 46*(1), 139–149.

Zhang, W., Hao, J., & Liu, F. (2024). Effective social spider optimization algorithms for distributed assembly permutation flowshop scheduling problem in automobile manufacturing supply chain. Scientific Reports, 14, 6370. https://doi.org/10.1038/s41598-024-57044-8

Appendix A. Pseudocode of the Proposed ABC Algorithm

```
Input:
  B              // colony size = number of solutions (food sources)
  p              // number of products |P|
  |F|            // number of factories
  prob           // probability to apply local search in employed phase
  limit          // trial limit for abandonment (scout condition)
  MaxIter        // maximum iterations
  Fitness(·)     // objective (assembly makespan)
  LS_insert_job, LS_swap_job, LS_insert_product, LS_swap_product,
  LS_insert_assembly, LS_swap_assembly  // local-search operators (first-improvement)
Output:
  BestSol        // best solution found
-------------------------------------------------------------
Procedure ABC_DAPFSP()
  // ---------- Initialization ----------
  Create array Population[1..B]
  For b = 1..B:
    if b == 1 then
      πA ← AssemblySequence_By_ShortestAssemblyTime()  // non-decreasing assembly times
    else
      πA ← RandomAssemblySequence(p)
    end if

    // Assign products to factories following assembly order:
    Assign[1..p] ← AssignProductsByAssemblyOrder(πA, |F|)
      // If p > |F|, randomly assign the last (p - |F|) products one-by-one to factories
    // Build factory job sequences respecting product grouping:
    For each factory f ∈ F:
      πf ← BuildFactoryJobSequence(f, Assign, πA) // concatenate jobs of assigned products in some start
order
    Population[b] ← (πA, {πf for f∈F})
    FitnessVal[b] ← Fitness(Population[b])
    Trials[b]     ← 0
  end For
  BestSol ← argmin_b FitnessVal[b]
  BestFit ← min_b  FitnessVal[b]

  iter ← 0
  // ---------- Main Loop ----------
  while iter < MaxIter:
    // ===== Employed Bee Phase =====
    For b = 1..B:
      x ← Population[b]
      // Choose one of six neighborhood structures uniformly or by a policy
      s ← RandomChoice({S1..S6})
      v ← ApplyNeighborhood(x, s)
        // S1: swap two products in assembly sequence
        // S2: insert one product to another position in assembly sequence
        // S3: for each factory: swap two products in that factory
        // S4: for each factory: insert one product to another position in that factory
        // S5: for each factory: swap two jobs (within products assigned to that factory)
        // S6: for each factory & product: insert one job to another position

      // Optional local search with probability prob
      if rand() < prob:
        v ← ApplyLocalSearch(v)
          // Apply one or more of:
          //   LS_insert_job, LS_swap_job,
```

```
        //   LS_insert_product, LS_swap_product,
        //   LS_insert_assembly, LS_swap_assembly
        // All use first-improvement and repeat until no improvement.
      if Fitness(v) < FitnessVal[b]:
        Population[b] ← v
        FitnessVal[b] ← Fitness(v)
        Trials[b] ← 0
        if FitnessVal[b] < BestFit:
          BestFit ← FitnessVal[b]
          BestSol ← Population[b]
        end if
      else
        Trials[b] ← Trials[b] + 1
      end if
    end For
    // ===== Onlooker Bee Phase =====
    // Roulette-wheel selection on fitness (lower is better → convert to probabilities)
    probs[1..B] ← RouletteWheelFromFitness(FitnessVal)
    For o = 1..B:
      b ← SampleIndex(probs)
      x ← Population[b]
      s ← RandomChoice({S1..S6})
      v ← ApplyNeighborhood(x, s)
      // Local search ALWAYS applied in onlooker phase
      v ← ApplyLocalSearch(v)
      if Fitness(v) < FitnessVal[b]:
        Population[b] ← v
        FitnessVal[b] ← Fitness(v)
        Trials[b] ← 0
        if FitnessVal[b] < BestFit:
          BestFit ← FitnessVal[b]
          BestSol ← Population[b]
        end if
      else
        Trials[b] ← Trials[b] + 1
      end if
    end For
    // ===== Scout Bee Phase =====
    For b = 1..B:
      if Trials[b] ≥ limit:
        // Abandon and reinitialize (random exploration)
        x ← ReinitializeSolution(p, |F|)
          // random assembly sequence + product assignment + factory job sequences
        Population[b] ← x
        FitnessVal[b] ← Fitness(x)
        Trials[b] ← 0
      end if
    end For
    // ===== Elitist Replacement (your diversification rule) =====
    worstIdx ← argmax_b FitnessVal[b]
    Population[worstIdx] ← BestSol
    FitnessVal[worstIdx] ← BestFit
    Trials[worstIdx] ← 0

    iter ← iter + 1
  end while
  return BestSol
End Procedure
-------------------------------------------------------------
```

Appendix B: Pseudo code of the neighborhood structures

```
Subroutine ApplyNeighborhood(x, s):
  switch s:
    case S1: return SwapTwoProductsInAssembly(x)
    case S2: return InsertOneProductInAssembly(x)
    case S3: return SwapTwoProductsPerFactory(x)
    case S4: return InsertOneProductPerFactory(x)
    case S5: return SwapTwoJobsPerFactory(x)
    case S6: return InsertOneJobPerProductPerFactory(x)
  end switch
```

Appendix C: Pseudo code of the local search procedures

```
Subroutine ApplyLocalSearch(x):
  improved ← true
  while improved:
    improved ← false
    for LS in [LS_insert_job, LS_swap_job,
               LS_insert_product, LS_swap_product,
               LS_insert_assembly, LS_swap_assembly]:
      x_new ← FirstImprovement(LS, x)
      if Fitness(x_new) < Fitness(x):
        x ← x_new
        improved ← true
      end if
    end for
  end while
  return x
```