A novel discrete artificial bee colony algorithm for the hybrid flowshop scheduling problem with makespan minimisation
Quan-Ke Pan, Ling Wang, Jun-Qing Li, Jun-Hua Duan

PII: S0305-0483(13)00131-X
DOI: http://dx.doi.org/10.1016/j.omega.2013.12.004
Reference: OME1373

To appear in: Omega

Received date: 6 July 2012
Accepted date: 25 December 2013

Cite this article as: Quan-Ke Pan, Ling Wang, Jun-Qing Li, Jun-Hua Duan, A novel discrete artificial bee colony algorithm for the hybrid flowshop scheduling problem with makespan minimisation, Omega, http://dx.doi.org/10.1016/j.omega.2013.12.004

This is a PDF file of an unedited manuscript that has been accepted for publication. As a service to our customers we are providing this early version of the manuscript. The manuscript will undergo copyediting, typesetting, and review of the resulting galley proof before it is published in its final citable form. Please note that during the production process errors may be discovered which could affect the content, and all legal disclaimers that apply to the journal pertain.
A novel discrete artificial bee colony algorithm for the hybrid flowshop scheduling problem with makespan minimisation

Quan-Ke Pan\textsuperscript{a,b}, Ling Wang\textsuperscript{c*}, Jun-Qing Li\textsuperscript{a,b}, Jun-Hua Duan\textsuperscript{a,b}

\textsuperscript{a}State Key Laboratory of Synthetical Automation for Process Industries (Northeastern University), Shenyang, 110819, China
\textsuperscript{b}College of Computer Science, Liaocheng University, Liaocheng, 252059, China
\textsuperscript{c}Tsinghua National Laboratory for Information Science and Technology (TNList), Department of Automation, Tsinghua University, Beijing 100084, China

\textsuperscript{*}Corresponding author. Tel.: +8610 6278 3125; fax: +8610 6278 6911.
wangling@mail.tsinghua.edu.cn

Abstract
The hybrid flowshop scheduling (HFS) problem with the objective of minimising the makespan has important applications in a variety of industrial systems. This paper presents an effective discrete artificial bee colony (DABC) algorithm that has a hybrid representation and a combination of forward decoding and backward decoding methods for solving the problem. Based on the dispatching rules, the well-known NEH heuristic, and the two decoding methods, we first provide a total of 24 heuristics. Next, an initial population is generated with a high level of quality and diversity based on the presented heuristics. A new control parameter is introduced to conduct the search of employed bees and onlooker bees with the intention of balancing the global exploration and local exploitation, and an enhanced strategy is proposed for the scout bee phase to prevent
the algorithm from searching in poor regions of the solution space. A problem-specific local refinement procedure is developed to search for solution space that is unexplored by the honey bees. Afterward, the parameters and operators of the proposed DABC are calibrated by means of a design of experiments approach. Finally, a comparative evaluation is conducted, with the best performing algorithms presented for the HFS problem under consideration, and with adaptations of some state-of-the-art metaheuristics that were originally designed for other HFS problems. The results show that the proposed DABC performs much better than the other algorithms in solving the HFS problem with the makespan criterion.

Highlights

- We present a novel discrete artificial bee colony algorithm to minimize makespn for the hybrid flowshop scheduling problem.
- A total of 24 heuristics are provided to generate a good initial population.
- A new hybrid representation combining the forward decoding and backward decoding method are proposed.
- A new control parameter is introduced to balance the global exploration and local exploitation.
- Computational and statistical analyses show the superiority of the presented approach.

Keywords

Hybrid flowshop; Makespan; Heuristics; Metaheuristics; Artificial bee colony
1. Introduction

The area of flowshop scheduling problems has been a very active field of research during the last 50 years since Johnson’s seminal work in 1954 [1]. The flowshop scheduling problem considers the processing of \( n \) jobs on \( m \) production stages in the same production flow, i.e., first on stage 1, then on stage 2, and so on until stage \( m \). Extensive research has been conducted for the regular flowshop scheduling problem, which assumes that each stage consists of only one machine. Nevertheless, this arrangement is not the case for many production systems today. Real production floors often employ multiple parallel machines at the same stage, to increase the throughput and capacity of the shop floor, to balance the speeds of the stages, or to either eliminate or reduce the impact of the bottleneck stage on the overall shop efficiency [2]. The flowshop scheduling problem with parallel machines, or the hybrid flowshop scheduling (HFS) problem, has important applications in a variety of industrial systems, including the glass, steel, paper and textile industries [1,3,4]. Two state-of-the-art comprehensive reviews on HFS problems can be found in [5] and [6].

The HFS problem that is considered in this paper simulates common manufacturing processes that consist of a set of production stages with at least one stage having two or more identical parallel machines. All of the jobs follow the same processing route through these stages. Unlike the regular flowshop scheduling problem, in which a permutation of jobs for each machine is the only thing needed, the HFS problem, with potentially more than one machine per stage, must consider the assignment of jobs to machines and the sequence of jobs on each machine. Thus, the HFS problem is much more complex. In the literature, the most common and popular measure of the performance of scheduling is the minimisation of the makespan or the maximum completion time (denoted as \( C_{\text{max}} \)). This criterion can lead to a high throughput and a high utilisation of production resources. We also consider minimising the makespan for the HFS problem in this paper. This problem is denoted as \( FHm//C_{\text{max}} \) by using the well-known \( \alpha/\beta/\gamma \) notation [7] and the extension for HFS problems [8]. The complexity of the HFS problem has been proven by Gupta [9] to be NP-hard even when the problem has only two stages (\( m=2 \)) and one of the stages contains a single machine. Therefore, exact algorithms such as branch-and-bound (B&B) and mixed-integer programming (MIP) can optimally solve the HFS problem only when there is a small size or a very simple scenario, for example, the systems with two centres, or systems in which only one of the centres has parallel machines [10-12]. For decades, much more effort has been dedicated to finding high-quality solutions in a reasonable computational time by heuristic and metaheuristic techniques instead of finding the optimal solution.

Heuristic methods are based on the specific characteristics of the problem. For example, Gupta [9] developed a very simple heuristic to minimise the makespan for a two-stage HFS problem, with only one machine at the second stage. Kim et al. [13] recently presented several heuristics for a similar
two-stage HFS with release dates and a product-mix ratio constraint. Riane et al. [14] presented two heuristic procedures for a three-stage HFS problem with a specific structure (one machine in the first and third stages and two dedicated machines in stage two), whereas Soewandi and Elmaghraby [15] proposed several heuristic algorithms for a general three-stage HFS problem with makespan criterion. For the $m$-stage problem, Ruiz and Maroto [1] adapted the well-known NEH heuristic [16] to an HFS problem with sequence-dependent setup times and machine eligibility. Paternina-Arboleda et al. [17] presented a heuristic that was based on the identification and exploitation of the bottleneck stage.

Meta-heuristics presented for HFS problems have grown quickly. For the HFS problem of minimising the makespan, Nowicki and Smutnick [18] proposed a Tabu search algorithm in which a specific neighbourhood was defined by using the concepts of the critical path in a graph and a block of jobs. Engin and Doyen [19] presented an artificial immune approach (AIS), Alaykyran et al. [20] proposed an ant colony optimisation (ACO) method, and Kahraman et al. [21] developed a genetic algorithm (GA$_k$). All of these algorithms were used to solve the benchmark problems provided by Carlier and Neron [11], with a maximum CPU time equal to 1600 s as a termination criterion. The experimental results showed that all of the algorithms solved more instances than the B&B method [11]. Jin et al. [22] presented two simulated annealing (SA) methods to sequence jobs and then to allocate jobs to machines based on a specific partition of the shop. Recently, Niu et al. [23] designed a quantum-inspired immune algorithm (QIA) that combined the concept of the biological immune system and the Quantum computer. More recently, Liao et al. [24] presented a particle swarm optimisation (PSO) algorithm by hybridising the discrete PSO [25] with a bottleneck heuristic [17], to fully exploit the bottleneck stage, and with SA to help escape from local optima. The experiments conducted by the authors showed that the PSO algorithm solved more of Carlier and Neron’s benchmark problems with a smaller deviation than the AIS [19], ACO [20], QIA [23], and B&B [11] in 1600 s. Because the benchmarks were originally designed for the B&B method, and the largest problem size has only 15 jobs and 10 stages, Liao et al. [24] further compared their PSO algorithm with AIS by using larger instances with 30 jobs and 5 stages. The results indicated that the PSO performed significantly better than the AIS [19].

For the HFS problem with more complex settings, Khaloui et al. [26] presented an ACO method with some heuristics that specifically took into account both the earliness and tardiness to compute the heuristic information values. Behnamian and Zandieh [27] proposed a discrete colonial competitive algorithm to determine a schedule that minimises the sum of the linear earliness and quadratic tardiness with simultaneously considering the effects of sequence-dependent setup times and limited waiting time. Ruiz and Maroto [1] provided a genetic algorithm (GA$_k$ for short) to find the minimum makespan for a complex HFS problem that resulted from the addition of unrelated parallel machines at each stage, sequence dependent setup times and machine eligibility. Later, Naderi et al. [2] also considered the same problem and proposed an iterated local search (ILS) algorithm that had much better performance.
Naderi et al. [28] presented an improved simulated annealing (ISA) for an HFS problem that was involved in both sequence-dependent setup and transportation times, to minimise the total completion and total tardiness.

Among the metaheuristics, modelling the collective behaviour of self-organised systems and applying these models to solve real-world problems has been ongoing and has become a class of its own, which is known as swarm intelligence [29]. The artificial bee colony (ABC) algorithm is a fairly new swarm intelligence method that is inspired by the specific aggregating behaviours of honeybee swarms. The ABC was recently proposed by Karaboga [30] to optimise multi-variable and multi-modal continuous functions. Numerical comparisons have demonstrated that the performance of the ABC algorithm is as competitive as other population-based algorithms and that it has the advantage of employing fewer control parameters [31-34]. In addition, several discrete versions of the ABC algorithm have been recently applied to the lot-streaming flowshop problem [35], permutation flowshop problem [29], three-stage HFS problem [3], and order acceptance and scheduling problem [36].

Following the successful applications of the ABC to scheduling problems, this paper presents a novel discrete ABC (DABC) for the HFS problem with the makespan criterion. The DABC includes some novel and advanced techniques, such as a hybrid representation with the forward and backward scheduling methods, a heuristic-based initialisation, a new control parameter to balance exploration and exploitation, a solution generation strategy for a scout bee to lead the algorithm to a promising region, and a problem-specific local procedure to improve the search capability. The effectiveness of the DABC is demonstrated by extensive comparisons against the best existing methods for the considered problem as well as with several adapted algorithms that were originally designed for other HFS problems.

The remainder of the paper is organised as follows. In Section 2, the HFS problem is formulated. In Section 3, a total of 24 heuristics are presented. In Section 4, the basic ABC is introduced. Section 5 describes the basic components of the presented DABC, whereas a calibration of the presented approaches is provided in Section 6. The computational results and comparisons are reported in Section 7. Finally, Section 8 provides the concluding remarks and possible future studies.

2 The hybrid flowshop scheduling problem

The HFS problem can be considered to be an extension of two classical scheduling problems of the regular flowshop and parallel shop. The hybrid flowshop consists of a series of \( m \) production stages (or processing centres) \( M = \{1, 2, \ldots, m\} \). Each stage \( k \in M \) has \( l_k \geq 1 \) identical parallel machines (\( l_k \geq 2 \) for at least one stage). A set of \( n \) jobs \( J = \{1, 2, \ldots, n\} \) must be sequentially processed on the
same production order, i.e., first on stage 1, then on stage 2, and so on until the last stage m. In other words, each job \( j \in J \) has \( m \) different operations: \( O_{1,j}, O_{2,j}, \ldots, O_{m,j} \). Each operation \( O_{k,j} \), \( k \in \{1,2,\ldots,m\} \) corresponds to process job \( j \) on stage \( k \) with a deterministic, non-negative, and uninterrupted processing time \( p_{k,j} \). At each stage \( k \), job \( j \in J \) can be processed on any machine \( i \in \{1,2,\ldots,l_k\} \). At any time, no job can be processed on more than one machine, and no machine can process more than one job simultaneously. All of the jobs are independent and available for processing at time 0. The jobs can wait between stages, and the intermediate storage is unlimited. Job setup times and the travel time between consecutive stages are included in the job processing times at the corresponding stage or can be negligible. The objective is then to find a schedule with which the makespan or maximum completion time is minimised.

To formulate a mathematical model according to the above description, we first list the notation, as follows:

- \( s_{k,j} \): Starting time of job \( j \) at stage \( k \).
- \( U \): A very large positive number.
- \( x_{k,j,i} \): If job \( j \) is assigned to machine \( i \) at stage \( k \), then \( x_{k,j,i} = 1 \); otherwise, \( x_{k,j,i} = 0 \).
- \( y_{k,j,j'} \): If job \( j \) is preceding charge \( j' \) to be processed at stage \( k \), then \( y_{k,j,j'} = 1 \); otherwise, \( y_{k,j,j'} = 0 \).

With the above symbols, the HFS problem is formulated as follows:

\[
\text{Minimise } \mathcal{C}_\text{max} = \max_{p \in P} \{s_{m,j} + p_{m,j}\} \\
\text{s.t.} \\
\sum_{i=1}^{l_k} x_{k,j,i} = 1, \quad \forall j \in J, k \in M \quad (1) \\
s_{k,j} \geq 0, \quad \forall j \in J \quad (2) \\
s_{k+1,j} - s_{k,j} \geq p_{k,j}, \quad \forall j \in J, k, k+1 \in M \quad (3) \\
y_{k,j,j'} + y_{k,j',j} \leq 1, \quad \forall j, j' \in J, k \in M \quad (4) \\
s_{k,j} - (s_{k,j} + p_{k,j}) + U \cdot (3 - y_{k,j,j'} - x_{k,j,i} - x_{k,j',i}) \geq 0, \quad \forall j, j' \in J, k \in M, \quad i \in \{1,2,\ldots,l_k\} \quad (5) \\
x_{k,j,i} \in \{0,1\}, \quad j \in J, \quad k \in M, \quad i \in \{1,2,\ldots,l_k\} \quad (6) \\
y_{k,j,j'} \in \{0,1\}, \quad j, j' \in J, \quad k \in M \quad (7)
The objective function (1) is to minimise the maximum completion time or makespan. Constraints (2) ensure that a job passes through all stages and must be processed by exactly one machine at every stage. Constraints (3) ensure that the starting time of a job on the first stage is larger than or equal to zero. Constraints (4) are the technological requirement, i.e., for two consecutive operations of a job, the next operation can be started only after the preceding operation has been finished. Constraints (5) and (6) describe the machine capacity restriction. For two jobs processed on the same machine, the next job can be started only after the proceeding job has finished. Constraints (7) and (8) define the value ranges for the decision variables.

3 The presented heuristics

Heuristics that explore the specific characteristics of problems can find good solutions with a very limited computational effort. In the recent scheduling literature, it has been a common trend to construct a few initial individuals for metaheuristics by using effective heuristics. For this purpose, we present a total of 24 simple rules and NEH-based heuristics. In the HFS, with potentially more than one machine per stage, we must consider assigning jobs to machines and sequencing jobs on each machine. As is commonly performed in the literature [2, 24], we address sequencing and assignment separately. To be specific, we use the heuristics to generate a simple permutation of the jobs, which indicates the order in which the jobs are launched to the shop at the first stage, and then, we allocate a job to the first machine that becomes available, i.e., the machine that first finishes the job (if any) that was previously assigned to it. The first available machine also results in the earliest completion time of the job. This arrangement is consistent with the objective of minimising the makespan. For the subsequent stage \( k \) \((k = 2,3,\ldots,m)\), we take the completion times of jobs in a previous stage \( k - 1 \) as their release times, and we allocate them to the first available machine according to the increasing order of the release times.

3.1 The simple heuristics

We first adapt simple dispatching rules, including the longest processing time (LPT) and shortest processing time (SPT), to the HFS problem. LPT generates a permutation by sorting the jobs according to their non-increasing total processing times, i.e., \( \sum_{i=1}^{n} p_{i,j} \), whereas SPT produces a permutation according to the non-decreasing total processing times.

Intuitively, the processing time at the first stage has a larger effect on the complete solution than at other stages. Therefore, we generate a permutation of the jobs according to their processing times at the first stage, and we obtain two simple rules: the longest processing time at the first stage (LPTF) and the shortest processing time at the first stage (SPTF). LPTF/SPTF is the same as LPT/SPT but sorts jobs according to their processing times at the first stage, i.e., \( p_{h,j} \), \( j = 1,2,\ldots,n \).
Bottleneck is a phenomenon by which the performance or capacity of an entire system is severely limited by a single component [24]. According to [17], the bottleneck stage $k^b$ is defined as the stage that has the largest flow ratio between the workload and the total available capacity, i.e., $\frac{1}{\sum_{j=1}^{m} p_{k,j}}$, $k = 1, 2, ..., m$. We present two more heuristics: longest processing times till bottleneck stage (LPTB) and shortest processing times till bottleneck stage (SPTB). LPTB/SPTB is the same as LPT/SPT, but it sorts jobs according to their total processing time from stage one to the bottleneck stage $k^b$, i.e., $\sum_{k=1,2,\ldots,m} p_{k,j}$.

### 3.2 The NEH-based heuristics

The NEH from Nawaz et al. [16] was recognised as the highest performing heuristic for the permutation flowshop scheduling problem to minimise the makespan [37]. This approach has been successfully adapted to many other scheduling problems in the literature [38, 39]. Researchers also extended NEH to the HFS problems and obtained good results [1, 2]. NEH first yields a seed sequence $\beta = (\beta_1, \beta_2, \ldots, \beta_n)$ by using LPT. Then, the first job $\beta_1$ is extracted as the first job of the current sequence $\pi$, and the remaining jobs, $\beta_2, \ldots, \beta_n$, are then taken out one by one and are inserted into the best slot of the current sequence $\pi$. The procedure of NEH is described as follows (see Fig. 1):

**Procedure NEH**

1. **Generate a seed sequence** $\beta = (\beta_1, \beta_2, \ldots, \beta_n)$ by using the LPT rule
2. $\pi := (\beta_1)$
3. For $h := 2$ to $n$ do (the NEH enumeration procedure)
   1. Take job $\beta_h$ from $\beta$ and test it in all of the $h$ possible slots of $\pi$
   2. Insert job $\beta_h$ in $\pi$ at the slot that results in the lowest objective value
4. End for
5. Return $\pi$

Fig. 1. The NEH heuristic

There are a total of $\frac{n(n+1)}{2} - 1$ insertions, and each insertion generates a partial sequence. NEH must evaluate all of the generated sequences, and its computational complexity is $O(n^3 m)$.

It can be seen from Fig. 1 that LPT is only used to generate a seed sequence for the NEH enumeration, and the specific characteristics of the problem explored by LPT are not well-utilised in the process of insertion. This method can decrease the performance of NEH. To address this problem, as was performed in [38], we propose an alternative method that evaluates only the insertion of the last
λ jobs instead of all of the n jobs. To be specific, the last λ jobs in the sequence β = (β₁, β₂, ..., βₙ), i.e., βₙ₋ₐ₊₁, βₙ₋ₐ₊₂, ..., and βₙ, undergo the NEH enumeration, whereas the partial sequence (β₁, β₂, ..., βₙ₋ₐ) is used as a starting point. The relative positions of the jobs from (β₁, β₂, ..., βₙ₋ₐ) are not changed as the algorithm progresses. Because these relative positions are generated by using LPT, the new method utilises the nature of the problem in a better way. Furthermore, the number of evaluations of the new method is less than the original NEH because it considers only λ(2n - λ + 1)λ / 2 partial sequences. We denote the new method as NEH_LPT(λ), where λ is a control parameter. The procedure of NEH_LPT(λ) is described as follows (see Fig. 2):

**Procedure** NEH_LPT(λ)

Generate a job sequence β = (β₁, β₂, ..., βₙ) by using the LPT rule

π := (β₁, β₂, ..., βₙ₋ₐ)

for h := n - λ + 1 to n do % (the NEH enumeration procedure)

Take job βₕ from β and test it in all of the λ possible slots of π

Insert job βₕ in π at the slot that results in the lowest objective value

endfor

return π

Fig. 2. The NEH_LPT(λ) heuristic

From Fig. 2, it is clear that NEH_LPT(λ) is equal to NEH if λ = n - 1 and equal to LPT if λ = 0.

The initial seed sequence has an important effect on the performance of the NEH heuristic [39]. Similar to NEH_LPT(λ), by using the five simple rules, SPT, SPTF, LPTF, SPTB, and LPTB, to generate a seed sequence, respectively, we yield five more NEH-based heuristics, i.e., NEH_SPT(λ), NEH_SPTF(λ), NEH_LPTF(λ), NEH_SPTB(λ), and NEH_LPTB(λ). We will calibrate all of these heuristics in Section 6.1.

3.3 The heuristics based on a backward schedule
In Sections 3.1 and 3.2, we first sequence and assign jobs for stage 1, and then for stage 2, and so on, until stage \( m \). We call this method the forward schedule. For a given job sequence, a complete solution can also be yielded by traversing the jobs in the HFS in the opposite direction in reverse order. In other words, the last stage is first considered. Jobs are assigned from the last position of the given permutation to the first position and to the first available machine in a backward manner. For the preceding stages, the jobs are assigned according to the increasing order of their backward release times. We consider an example that has 3 stages and 4 jobs. Both stages 1 and 2 have two identical parallel machines, whereas stage 3 has one machine. Suppose that a given permutation is \( \pi = (1, 2, 3, 4) \), and the processing times of the jobs are given as follows:

\[
p_{s,j} = \begin{bmatrix} 4 & 3 & 6 & 3 \\ 6 & 5 & 1 & 3 \\ 2 & 3 & 2 & 2 \end{bmatrix}
\]

We schedule the jobs in the backward manner as follows.

1. First schedule jobs for stage 3. All of the jobs are allocated to machine 1 by first considering job 4, then job 3, and so on, until job 1.

2. Compute the backward release times for stage 2. The backward release times for jobs 4, 3, 2, and 1 are equal to 2, 4, 7, and 9, respectively. According to the increasing backward release times, we allocate the jobs to the first available machine in the backward manner. Thus, jobs 4, 3, 2, and 1 are arranged on machines 2, 1, 2, and 1, respectively.

3. The jobs in stage 1 are scheduled by using the same method as in stage 2. Finally, we obtain a complete solution with a makespan that is equal to 19 (see Fig. 3).
It is well known that the regular flowshop scheduling problem has the reversibility property: the makespan does not change if the jobs traverse the flowshop in the opposite direction, in reverse order [42]. However, this property is not held by the HFS problem. For a given job sequence, we often obtain different makespan values by using the forward and backward schedule methods. For the above example, the Gantt chart that can be generated by the forward schedule is given in Fig. 4.

We present six simple heuristics by using the backward method to generate a complete solution, i.e., bLPT, bSPT, bLPTL, bSPTL, bLPTB, and bSPTB, where bLPT/bSPT generates a permutation of jobs using LPT/SPT but schedules jobs in the backward manner, and bLPTL/bSPTL generates a job permutation according to the non-increasing/non-decreasing processing times on the last stage, and bLPTB/bSPTB accounts for the non-increasing/non-decreasing processing times from the bottleneck stage to the last stage, i.e., \( \sum_{i=1}^{n} P_{k-i} \). Correspondingly, we generate six NEH-based heuristics by combining them with the NEH heuristics, i.e., \( \text{NEH}_{\text{SPT}}(\lambda) \), \( \text{NEH}_{\text{LPT}}(\lambda) \), \( \text{NEH}_{\text{SPTL}}(\lambda) \), \( \text{NEH}_{\text{LPTL}}(\lambda) \), \( \text{NEH}_{\text{SPTB}}(\lambda) \), and \( \text{NEH}_{\text{LPTB}}(\lambda) \).

4 Introduction to the basic artificial bee colony algorithm

The artificial bee colony (ABC) algorithm was introduced by Karaboga [30] for continuous function optimisations. The ABC is a new swarm intelligence algorithm that is inspired by the intelligent foraging behaviour of honeybees when searching for nectar sources around their hives. In
the ABC, the problem solutions are represented as food sources. The colony of artificial bees consists of three groups, namely, employed bees, onlookers, and scouts. An employed bee is responsible for flying to and making collections from the food source that the bee swarm is exploiting; it shares the information with the onlookers about the found candidate solutions. An onlooker waits in the hive and decides on whether a food source is acceptable or not. This action is performed via watching the dances performed by the employed bees. When an employed bee abandons its food source, it becomes a scout that randomly searches for a new food source by means of some internal motivation or possible external clue. The ABC algorithm performs an iterative evolution process as follows [30-34]:

In the first phase, the ABC sets its parameters and determines the initial population. The parameters include the population size (denoted as $PS$), the number of trials with which a food source is assumed to be abandoned ($\alpha$), and a termination criterion. The initial population of solutions is filled with $PS$ $n$-dimensional real-valued vectors $\{z_1, z_2, ..., z_{PS}\}$ that are randomly generated in the solution space, where $z_u = \{z_{u,1}, z_{u,2}, ..., z_{u,n}\}$ represents the $u$th individual in the population.

After the initialisation, the algorithm is repeated with the search process of the employed bees, onlooker bees, and scout bees until a termination criterion is satisfied. Each employed bee is associated with a food source from the population. Then, it finds a new food source $z_{new}$ in the neighbourhood of its present position $z_u$, as follows.

$$z_{new,v} = z_{u,v} + (z_{u,v} - z_{q,v}) \times r$$

where $q \in \{1, 2, ..., PS\} \land q \neq u$, $v \in \{1, 2, ..., n\}$ is a randomly chosen index, and $r$ is a uniformly distributed real number between $-1$ and $1$.

Then, $z_{new}$ will be evaluated and compared to $z_u$. If $z_{new}$ is better than $z_u$, it will replace $z_u$ and become a new member of the population; otherwise, it will be discarded. In other words, a greedy selection mechanism is applied to choose the new solution or to keep the old solution.

After all of the employed bees finish the search process, each onlooker bee evaluates the nectar information that is taken from all of the employed bees and selects a food source $z_u$ depending on its probability value $\xi_u$, which is calculated as follows:

$$\xi_u = \frac{f_u}{\sum_{u=1}^{PS} f_u}$$

where $f_u$ is the nectar amount (i.e., the fitness value) of $z_u$. Clearly, the higher the $f_u$ is, the larger the probability that $z_u$ is selected.

Once the onlooker has selected a food source $z_u$, it produces a modification to $z_u$ by using the same method as the employed bees. If the modified individual has a nectar amount that is better than $z_u$, it
will replace \( z_u \) and become a new member of the population.

If a food source \( z_u \) is not further improved through a predetermined number of trials limit (\( \alpha \)), then it is assumed to be abandoned, and the corresponding employed bee becomes a scout. The scout produces a food source randomly in the solution space. At each cycle of the basic ABC algorithm, at most one scout goes outside to search for a new food source.

Although the basic ABC algorithm was originally designed for continuous optimisation problems and could not be directly applied to the discrete/combinatorial cases, several modified ABC algorithms have been applied to other types of problems with great success. It is worthwhile to evaluate the performance of the ABC algorithm for the HFS problem that is under consideration.

5 The presented ABC algorithm for the HFS problem

A novel discrete ABC (DABC) algorithm is proposed in this section. We first explain its details, which include the solution representation, population initialisation, employed bee phase, onlooker bee phase, scout bee phase, balance of exploration and exploitation, and problem-specific local search, and then, we give the computational procedure of the presented DABC algorithm.

5.1 A hybrid representation

During the past decades, many encodings of HFS problems have been presented. For example, Nowicki and Smutnick [18] designed a direct encoding method that contains the complete solution to the problem. In the encoding, the assignment of jobs to machines and the processing order of the jobs on each machine are given for each stage. Kurz and Askin [41] presented a random key representation for all of the stages. At each stage, the random key representation assigns a real number to every job. The integer part of this number is used to determine the machine for a job, while the fractional part indicates the processing order of the jobs on the assigned machine. This permutation-based encoding has been widely used in the literature. This approach involves indirect encoding, where a permutation of jobs in an array represents the order in which the jobs are launched to the shop at the first stage. For other stages, the forward schedule method is used to decode a complete solution. Intuitively, a very direct encoding that reflects both the sequencing and assignment is needed for the HFS problem. However, initial versions of the ABC with such a complex encoding produced poor results. Along with the fact observed by Ruiz and Maroto [1], Naderi et al. [2], and many others, we choose the simple permutation encoding in this paper, although the predefined solution space might not be fully addressed.

As mentioned in Section 3, a given sequence of jobs can be decoded into two different complete solutions by using the forward schedule and backward schedule methods, respectively. A solution that is generated by the backward schedule might be never reached by the forward schedule, and vice versa.

To enlarge the search scope and to enhance the diversity of the population, we employ both decoding
methods here. To be specific, an individual is represented as a permutation of jobs with a flag bit that indicates which decoding method is used. If the flag is equal to 0, then we use the forward schedule method; otherwise, the backward method is adopted. An example of an individual is given in Fig. 5.

![Fig. 5. An individual with a flag bit and job sequence](image)

We call this encoding a hybrid representation. As far as we know, this approach is a new representation for the HFS problem. With the presented representation, the existing permutation-based operators can be used in the search process of the ABC algorithm. Thus, the ABC algorithm directly works in discrete solution space, and we call it a discrete ABC (DABC) algorithm.

### 5.2 Initialisation of the population

The DABC starts from a population of initial individuals. We denote the population size as \( PS \). An initial population with a high level of quality and diversity always results in competitive levels of performance in hard scheduling problems. It is very common in the scheduling literature to construct a few good initial individuals by effective heuristics and to produce others randomly. Thus, along with the presented heuristics in Section 3, we present a simple initialisation procedure, as follows:

1. **Step 1:** Generate four different individuals by taking advantage of the presented heuristics. More specifically, one individual is generated by using the best simple heuristic in Section 3.1. One individual is yielded by the best NEH-based heuristic in Section 3.2. The remaining two individuals are generated by using the best simple heuristic and the best NEH-based heuristic with the backward schedule. Let counter \( Cnt = 4 \).

2. **Step 2:** If \( Cnt = PS \), then stop the procedure; otherwise, randomly produce a permutation of all of the jobs and randomly generate a flag bit for an individual.

3. **Step 3:** If the produced individual is different from all of the existing individuals, then place it into the population and set \( Cnt = Cnt + 1 \); otherwise, discard it.

4. **Step 4:** Go to Step 2.

In the initialisation, we generate four different individuals who have high quality instead of only one good individual, as is performed in the literature [1]. This approach is chosen because of the diversity of the population in the evolutionary process. If there is only one individual that is much better than the others in the population, then the algorithm would spend too much time searching around this individual because the individual has a very high opportunity to survive in the competition. This approach easily results in a local optimum.
5.3 Employed bee phase

Employed bees search for food sources in the neighbourhood for their current positions. A neighbourhood must be specified when applying the DABC algorithm to the HFS problem. In Eq. (9), a new individual is yielded by combining the current individual with a randomly selected individual. This method cannot be directly used for the problem under consideration. Therefore, we employ an alternative method. For the permutation-based encoding, the commonly used neighbourhoods are defined by insertion, swap, and pairwise exchange [42]. An insertion consists of removing the job at the $v^{th}$ position and then reinserting it into another position $h$. A swap exchanges two adjacent jobs at positions $v$ and $v+1$. A pairwise exchange exchanges a pair of jobs at the $v^{th}$ and $h^{th}$ positions. For the regular flowshop problem, it was shown that the insertion neighbourhood can be evaluated more efficiently than the pairwise exchange neighbourhood and gives at least the same solution quality [43, 44]. The evaluation to the swap neighbourhood is very fast, but a local search that is based on swap moves leads to a rather low solution quality [45]. However, for a three-stage HFS problem from the steelmaking-continuous casting process, Pan et al. [3] presented a multiple-swap operator that consists of a small number of swaps. The multiple-swap system resulted in much better results than swap, insertion and pairwise exchange for the steelmaking-continuous casting problem. In this paper, we consider the three neighbourhood operators, i.e., insertion, pairwise exchange, and multiple-swap, respectively, and calibrate them in Section 6. Note that the neighbourhood operators are applied only to the job sequence of an individual, and the flag bit is kept unchanged.

A new individual is always accepted if it is better than the current individual, which is the same as in the basic ABC algorithm when it conducts the greedy selection procedure.

5.4 Onlooker bee phase

In the basic ABC algorithm, an onlooker bee selects a food source or individual depending on its winning probability, which is similar to the fitness-proportional selection in the GA. Such an operator requires the fitness of each individual and a mapping calculation. Binary tournament selection is also widely used in the GA applications due to its simplicity and ability to escape from local optima [29]. Additionally, it has been successfully utilised to select an individual for an onlooker bee in the DABC algorithm [3, 29, 35]. We also choose the tournament selection in this paper. An onlooker bee selects an individual in such a way that two individuals are picked up randomly from the population and are compared to each other; then, the better one is chosen. This procedure results in a very fast selection operator because no fitness calculation and mapping are needed.

After the selection, the onlooker utilises the same method as an employed bee to produce a new neighbouring individual. At the onlooker phase of the basic ABC, the current individual is replaced by a neighbour if the neighbour has a better objective value. Here, we employ an alternative method. The new individual does not replace the current individual but instead replaces the worst individual with the
same flag bit in the population, and it is accepted only if it is better than the worst one and at the same time there is no other identical individual in the population. Otherwise, the new individual is rejected. As a result, potential individuals (i.e., individuals who can produce better neighbours) will be given a substantial opportunity to be further explored, whereas non-potential individuals, such as the worst one in the population, are excluded. At the same time, the diversity of the population is also maintained, which helps to avoid premature convergence to a suboptimal solution.

5.5 Scout bee phase

In the scout bee phase, a food source or individual is abandoned if it has not been improved in a predetermined number, $\alpha$, of consecutive iterations. In the basic ABC algorithm, a scout produces an individual randomly in the predefined search scope. This procedure will decrease the search efficacy and be a possible waste of time because a randomly generated individual will be, most likely, a worse solution, and many times, much worse. To direct our algorithm toward the most promising regions in the search space, we present an enhanced strategy to generate an individual for a scout, as follows:

The abandoned individual survives many generations and should carry good information during the evolution process. The search space around it would be the promising region. Therefore, we generate a number $\tau$ of good individuals by performing several random insertions into the job sequence of the abandoned individual. All of the $\tau$ solutions are evaluated, and the best solution is accepted as the new individual for the scout. Note that to avoid the algorithm trap into a local optimum, at least three insertions are performed. This modification is beneficial because it prevents the algorithm from searching in the poor regions of the solution space with no control of the quality of the individuals, and the algorithm has a high chance of moving to a new promising region that has not yet been explored.

5.6 Balance of exploration and exploitation

Employed bee and onlooker bee phases play important roles in the ABC. In the employed bee phase, a population of employed bees search in the neighbourhoods of their own current positions. Many small areas are explored simultaneously. This approach forms a large search scope of solutions. The algorithm is in favor of global exploration. In the onlooker bee phase, only the neighbourhoods of the better individuals who win the tournament competition are searched. The whole search area is limited. Furthermore, the updating mechanism, which replaces the worst individuals in the population by the neighbours of better individuals, decreases the diversity of the population. Thus, the onlooker bee phase helps local exploitation. To well balance the exploration and exploitation, we introduce a new control parameter $\gamma$. Instead of performing the employed bee phase only once, as is performed in the basic ABC algorithm, we perform the employed bee phase $\gamma$ replications before the algorithm enters the scout bee phase. By setting a suitable value for the parameter $\gamma$, the global exploration and local exploitation can be well balanced.
5.7 Problem-specific local search

As mentioned in Section 5.1, there are solutions that are unexplored due to the encoding and decoding methods. Consider an example of the HFS that has two stages, and each stage has two machines. Five jobs are processed with the following processing times.

\[
p_{k,j} =\begin{bmatrix} 4 & 3 & 6 & 2 & 1 \\ 5 & 4 & 1 & 1 & 4 \end{bmatrix}
\]

Given an individual with a flag equal to 0 and the job sequence of \((1, 2, 3, 4, 5)\). We generate a complete solution by using the forward schedule. At the first stage, we find that the sequence of jobs on machine 1 is \((1, 4, 5)\) and on machine 2 is \((2, 3)\). At the second stage, we sort the jobs according to their release times (i.e., the completion time in stage 1) and obtain the sequence \((2, 1, 4, 5, 3)\). Then, by using the forward schedule method, we obtain that the sequence of jobs on machine 1 is \((2, 4, 5)\) and on machine 2 it is \((1, 3)\). The schedule Gantt of the example is shown in Fig. 6, and the makespan is 12.

![Fig. 6. An example of the forward schedule](image)

From Fig. 6, we find that both jobs 4 and 5 are ready for processing when machine 1 of stage 2 finishes job 2. Our decoding method selects job 4 because its release time is smaller. However, we can also choose job 5 without increasing the machine idle time. In this case, we exchange the positions of jobs 4 and 5 in the sequence \((2, 1, 4, 5, 3)\) and obtain a new sequence \((2, 1, 5, 4, 3)\) for stage 2. With this new sequence, we apply the forward schedule to produce a solution that has a smaller makespan, which is equal to 11 in Fig. 7. The DABC cannot find this new solution with the aforementioned encoding and decoding method.
To address this problem and to further improve the performance of the DABC, we present a local search procedure for a given complete solution with the forward schedule method as follows. Let $k = 2$.

Step 1. Generate a sequence $\pi = (\pi_1, \pi_2, ..., \pi_n)$ of jobs according to their completion times at stage $k - 1$ in the given solution. Set $h = 1$.

Step 2. Find the first available machine for $\pi_h$ and denote the machine as $i^*$. Find all of the jobs that are sequenced after $\pi_h$ that are ready to be processed when machine $i^*$ finishes its previous job. Suppose that the set of these jobs is $\Lambda = \{\pi_{h+1}, \pi_{h+2}, ..., \pi_n\} \in \pi$. If $\Lambda$ is empty, then go to Step 4.

Step 3. Exchange the positions in $\pi = (\pi_1, \pi_2, ..., \pi_n)$ of $\pi_h$ and another job from $\Lambda$, and obtain a new sequence $\pi'$. Schedule stage $k$ according to $\pi'$ and the subsequent stages (i.e., $k+1, k+2, ..., m$) according to the release times of the jobs. We denote the yielded complete solution as a neighbour of the given solution. Repeat Step 3 until all of the jobs in $\Lambda$ are considered.

Step 4. Let $h = h + 1$. Go to Step 2 if $h < n$.

Step 5. Let $k = k + 1$. If $k \leq m$, then go to Step 1. Otherwise, evaluate all of the generated neighbours and output the best one.

A similar local search procedure can be generated for the individual by using the backward decoding. In the worst case, the local search procedures must evaluate $(n-1)(n-2)(m-1)/2$ neighbouring solutions. Therefore, we only apply the local search procedures to the best solution that is achieved by the DABC algorithm.

### 5.8 Computational procedure of the DABC algorithm

Having discussed all of the components, we outline the complete computational procedure of the DABC algorithm as follows:

Step 1: Parameter initialisation. Set population size ($PS$), the number of consecutive iterations in
which an individual is not improved ($\alpha$), the replications of the employed bee stage ($\gamma$), and the number of neighbours that are generated for an abandoned individual at scout phase ($\tau$).

Step 2: Population initialisation. Initialise the population $\{\pi_1, \pi_2, \ldots, \pi_{PS}\}$ by using the method presented in Section 5.2.

Step 3: Employed bee phase. For $u = 1, 2, \ldots, PS$, produce a new individual $\pi_u^*$ for the $u^{th}$ employed bee, which is associated with individual $\pi_u$. If $\pi_u^*$ is better than $\pi_u$, then let $\pi_u = \pi_u^*$. Repeat Step 3 until the employed bee phase is performed with $\gamma$ replications.

Step 4: Onlooker phase. For $u = 1, 2, \ldots, PS$, select an individual for the onlooker bee $u$ by using tournament selection. Generate a new individual for the onlooker. Update the population if the generated individual is better than the worst individual with same flag bit and unique in the population.

Step 5: Scout phase. For $u = 1, 2, \ldots, PS$, if individual $\pi_u$ has not been improved during the last $\alpha$ trials, then abandon it. Generate $\tau$ neighbours by performing several insertions to $\pi_u$. Put the best neighbour into the population.

Step 6: Output the best solution achieved so far if the termination criterion is reached; otherwise, go to Step 3.

Step 7: Perform the local search presented in Section 5.7 to find the best solution.

In the above procedure, a novel hybrid representation is presented to enlarge the search scope and diversify the population. A heuristic initialisation is employed to provide a good initial population with a high level of quality and diversity. An enhanced strategy is proposed to prevent the algorithm from searching in the poor regions of the solution space. A new control parameter is introduced to balance the global exploration and local exploitation. A problem-dependent local search is developed to intensively explore the neighbourhood of the solution that is returned by the DABC search. Because a balance of generality and problem specificity is stressed, the approach is expected to achieve good results for the HFS problem with the makespan criterion.

6. Calibration of the proposed algorithms

6.1 Comparison and calibration of the presented heuristics

To compare and calibrate the presented heuristics, we generate a set of instances with $n \in \{20, 40, 60, 80, 100\}$ and $m \in \{5, 8, 10\}$. Ten instances are generated for each combination of $n$ and $m$. This arrangement leads to a total of 150 instances. The processing times are given by a discrete uniform distribution in the interval of $[1, 99]$, and the number of identical parallel machines ($l_i$) in each stage is uniformly generated in the range of $[1, 5]$. The above distribution of processing times, the number of jobs, $n$, and the number of machines, $m$, have been widely used in the scheduling literature. In addition, a set of 150 instances is sufficient to show the difference in the performances between the compared heuristics.
We code all of the 24 heuristics in VC++ 6.0 and run them on a Pentium (R) 4 CPU 3.0 GHz with 1.0 GB Main Memory in a Windows XP Operation System. The relative percentage increase (RPI) is calculated as follows:

\[
RPI(c_i) = \frac{(c_i - c^{\text{min}})}{c^{\text{min}}} \times 100\% 
\]

(11)

where \( c_i \) is the makespan that is generated by a given heuristic, and \( c^{\text{min}} \) is the minimum makespan that is found among all of the heuristics.

The RPI values of the six simple heuristics with the forward schedule (i.e., SPT, LPT, SPTF, LPTF, SPTB, and LPTB), when averaged for each combination of \( n \) and \( m \) (10 data per average), are reported in Table 1.

<table>
<thead>
<tr>
<th>Instances</th>
<th>SPT</th>
<th>LPT</th>
<th>SPTF</th>
<th>LPTF</th>
<th>SPTB</th>
<th>LPTB</th>
</tr>
</thead>
<tbody>
<tr>
<td>20×5</td>
<td>6.84</td>
<td>4.11</td>
<td>3.52</td>
<td>12.26</td>
<td>4.76</td>
<td>15.23</td>
</tr>
<tr>
<td>20×10</td>
<td>12.12</td>
<td>5.88</td>
<td>5.79</td>
<td>13.80</td>
<td>3.96</td>
<td>14.37</td>
</tr>
<tr>
<td>40×5</td>
<td>5.73</td>
<td>4.14</td>
<td>1.83</td>
<td>10.97</td>
<td>2.11</td>
<td>10.09</td>
</tr>
<tr>
<td>40×8</td>
<td>8.08</td>
<td>5.57</td>
<td>1.73</td>
<td>11.46</td>
<td>0.90</td>
<td>12.58</td>
</tr>
<tr>
<td>40×10</td>
<td>7.51</td>
<td>3.65</td>
<td>3.73</td>
<td>10.19</td>
<td>2.35</td>
<td>12.74</td>
</tr>
<tr>
<td>60×5</td>
<td>4.15</td>
<td>4.71</td>
<td>1.87</td>
<td>12.10</td>
<td>1.34</td>
<td>11.29</td>
</tr>
<tr>
<td>60×8</td>
<td>5.83</td>
<td>5.07</td>
<td>3.75</td>
<td>11.95</td>
<td>0.45</td>
<td>14.92</td>
</tr>
<tr>
<td>60×10</td>
<td>6.19</td>
<td>2.64</td>
<td>3.82</td>
<td>10.77</td>
<td>1.14</td>
<td>13.41</td>
</tr>
<tr>
<td>80×5</td>
<td>3.74</td>
<td>2.78</td>
<td>2.01</td>
<td>7.55</td>
<td>1.48</td>
<td>8.02</td>
</tr>
<tr>
<td>80×8</td>
<td>4.94</td>
<td>1.95</td>
<td>3.04</td>
<td>5.65</td>
<td>2.42</td>
<td>6.11</td>
</tr>
<tr>
<td>80×10</td>
<td>3.66</td>
<td>2.19</td>
<td>3.21</td>
<td>6.58</td>
<td>3.43</td>
<td>7.71</td>
</tr>
<tr>
<td>100×5</td>
<td>2.43</td>
<td>3.87</td>
<td>1.70</td>
<td>8.92</td>
<td>0.53</td>
<td>10.29</td>
</tr>
<tr>
<td>100×8</td>
<td>3.68</td>
<td>3.62</td>
<td>2.15</td>
<td>6.98</td>
<td>0.56</td>
<td>9.25</td>
</tr>
<tr>
<td>100×10</td>
<td>4.60</td>
<td>3.37</td>
<td>2.61</td>
<td>8.01</td>
<td>1.06</td>
<td>7.74</td>
</tr>
<tr>
<td>Average</td>
<td>5.80</td>
<td>3.86</td>
<td>3.03</td>
<td>10.01</td>
<td>1.91</td>
<td>11.16</td>
</tr>
</tbody>
</table>

As shown in Table 1, the best performer is SPTB, and it has the minimum overall average RPI (ARPI) of 1.91%. For 11 out of 15 instance sizes, SPTB produces much smaller PRI values than the others. SPTF and LPT rank second and third, with 3.03% and 3.86% ARPI, respectively. The above three heuristics perform much better than the remaining heuristics, i.e., SPT, LPTF, and LPTB.

To compare the NEH-based heuristics with the forward schedule, i.e., \( \text{NEH}_{\text{SPT}}(\lambda) \), \( \text{NEH}_{\text{LPT}}(\lambda) \), \( \text{NEH}_{\text{SPT}}(\lambda) \), \( \text{NEH}_{\text{LPT}}(\lambda) \), \( \text{NEH}_{\text{SPT}}(\lambda) \) and \( \text{NEH}_{\text{LPT}}(\lambda) \), we set \( \lambda \) from 0 to \( n \) with a step equal to \( n \cdot 5\% \). This approach results in 21 different configurations for each heuristic. For each configuration, we run the heuristic to solve all of the 150 instances, and for each instance, we calculate the RPI. We use the ARPI over all of the instances to show the performance in Fig. 8.
Fig. 8. Behaviour of the NEH-based heuristics with the forward schedule as a function of $\lambda$

It can be seen from Fig. 8 that all of the heuristics with $\lambda > 0$ produce a much lower ARPI value than with $\lambda = 0$, which suggests that the NEH-based heuristics with the forward schedule outperform their corresponding simple heuristics at a substantial margin. The smallest ARPI is generated by $NEH_{LPT}(\lambda)$ with $\lambda = n \cdot 50\%$. From this point, the performance of $NEH_{LPT}(\lambda)$ gradually decreases as $\lambda$ increases. A similar behaviour can be observed for $NEH_{SPT}(\lambda)$ and $NEH_{SPTB}(\lambda)$.

We continue our experiment for the simple and NEH-based heuristics with the backward schedule. Table 2 reports the RPI values of the simple heuristics with the backward schedule.

<table>
<thead>
<tr>
<th>Instances</th>
<th>bSPT</th>
<th>bLPT</th>
<th>bSPTL</th>
<th>bLPTL</th>
<th>bSPTB</th>
<th>bLPTB</th>
</tr>
</thead>
<tbody>
<tr>
<td>20×5</td>
<td>7.39</td>
<td>10.10</td>
<td>21.06</td>
<td>2.43</td>
<td>16.20</td>
<td>4.83</td>
</tr>
<tr>
<td>20×8</td>
<td>7.27</td>
<td>3.82</td>
<td>9.83</td>
<td>3.64</td>
<td>16.01</td>
<td>1.72</td>
</tr>
<tr>
<td>40×5</td>
<td>4.62</td>
<td>4.83</td>
<td>11.08</td>
<td>3.45</td>
<td>11.89</td>
<td>2.42</td>
</tr>
<tr>
<td>40×8</td>
<td>6.67</td>
<td>5.46</td>
<td>9.03</td>
<td>4.36</td>
<td>15.75</td>
<td>0.38</td>
</tr>
<tr>
<td>40×10</td>
<td>6.36</td>
<td>4.37</td>
<td>8.48</td>
<td>4.33</td>
<td>13.19</td>
<td>1.14</td>
</tr>
<tr>
<td>60×5</td>
<td>3.56</td>
<td>5.34</td>
<td>13.83</td>
<td>1.59</td>
<td>13.58</td>
<td>2.27</td>
</tr>
<tr>
<td>60×8</td>
<td>3.36</td>
<td>5.00</td>
<td>9.89</td>
<td>1.96</td>
<td>15.42</td>
<td>1.00</td>
</tr>
<tr>
<td>60×10</td>
<td>4.95</td>
<td>3.01</td>
<td>8.72</td>
<td>3.04</td>
<td>13.69</td>
<td>0.76</td>
</tr>
<tr>
<td>80×5</td>
<td>3.79</td>
<td>3.30</td>
<td>11.18</td>
<td>1.76</td>
<td>9.69</td>
<td>1.47</td>
</tr>
<tr>
<td>80×8</td>
<td>4.18</td>
<td>2.45</td>
<td>6.96</td>
<td>2.28</td>
<td>10.72</td>
<td>0.99</td>
</tr>
<tr>
<td>80×10</td>
<td>2.92</td>
<td>2.90</td>
<td>6.90</td>
<td>2.07</td>
<td>9.89</td>
<td>1.24</td>
</tr>
<tr>
<td>100×5</td>
<td>2.06</td>
<td>4.83</td>
<td>10.34</td>
<td>2.70</td>
<td>9.48</td>
<td>1.12</td>
</tr>
<tr>
<td>100×8</td>
<td>3.24</td>
<td>3.79</td>
<td>8.33</td>
<td>1.14</td>
<td>11.31</td>
<td>1.16</td>
</tr>
<tr>
<td>100×10</td>
<td>3.72</td>
<td>3.46</td>
<td>6.95</td>
<td>2.90</td>
<td>9.67</td>
<td>1.11</td>
</tr>
<tr>
<td>Average</td>
<td>4.81</td>
<td>4.63</td>
<td>10.42</td>
<td>2.81</td>
<td>12.97</td>
<td>1.47</td>
</tr>
</tbody>
</table>

It is clear from Table 2 that bLPTB is the best performer, with 1.47% APRI, which is much smaller than those generated by bLPTL (2.81%), bLPT (4.63%), bSPT (4.81%), bSPTL (10.42%), and
bSPTB (12.97%).

The performances of the NEH-based heuristics with the backward schedule are shown in Fig. 9. As can be observed, the lowest APRI value is reached by $b\text{NEH}_{\text{SPT}}(\lambda)$, with $\lambda = n/45\%$.

![Fig. 9. Behaviour of the NEH-based heuristics with the backward schedule as a function of $\lambda$](image)

From the above discussion, we select SPTB, $NEH_{LPT}(\lambda)$, bLPTB and $b\text{NEH}_{\text{SPT}}(\lambda)$ to generate four different solutions for the initial population of the DABC algorithm, where we set $\lambda = n/2$ for simplicity.

### 6.2 Calibration of the proposed DABC algorithm

This section calibrates the presented DABC algorithm. The parameters are: the population size ($PS$), the number of consecutive iterations in which an individual is not improved ($\alpha$), the replications in the employed bee stage ($\gamma$), and the number of neighbours that are generated for an abandoned individual at the scout phase ($\tau$). After primary experiments, we set $PS$ to four levels: 20, 30, 40, and 50; $\alpha$ to four levels: 30, 50, 100, and $\infty$; $\gamma$ at four levels: 1, 10, 20, and 30; and $\tau$ at four levels: 1, 10, 20, and 30. For the solution representation, we consider three methods: the presented hybrid encoding, the job permutation encoding with only forward decoding, and the job permutation encoding with only backward decoding. As mentioned in Section 5.3, we also test three solution generation operators: insertion, pairwise exchanges, and multiple-swap. All of the possible combinations of the above factors result in a total of $4 \times 4 \times 4 \times 3 \times 3 = 2304$ different configurations for the proposed DABC.
We consider a full factorial design, and we code the DABC in VC++ 6.0. All of the above 2304 configurations are run on the aforementioned PC with the termination criterion set to the maximum elapsed CPU time of $t = 10nm$ milliseconds. We randomly generate an instance for each combination of $n \in \{20, 40, 60, 80\}$ and $m \in \{5, 8, 10\}$, and five independent replications are conducted for each instance. The RPI is calculated as a response variable. Note that in the experiment, we have a total of 2304 algorithm configurations, which are executed 5 times and give a total of $2304 \times 5 = 11520$ results for each instance. The minimum makespan $c_{\text{min}}$ that is used when computing RPI is the best result among the 11520 results.

The experimental results are analysed by using the $F$-ratio for those factors whose $p$-value is close to zero. The greatest $F$-ratio corresponds to a type of solution generation operators. This finding suggests that the solution generation operator has the most important effect over the response variable among the considered factors. The means of insertion, pairwise exchanges, and multiple-swap with the Tukey HSD intervals (at the 95% confidence level) are plotted in Fig 10. Clearly, multi-swap leads to the worst performing DABC, while no significant difference is observed between the results that are generated by the insertion and by pairwise exchange. We select insertion in our DABC because it produces the smallest ARPI value.

![Fig. 10. Means plot for the type of solution generation operators](image)

The type of representation corresponds to the second largest $F$-ratio. There are three levels that are considered for this factor. The means plot for this factor can be observed in Fig. 11. It can be observed that the presented hybrid representation results in significantly better ARPI than the others, which suggests the effectiveness of the hybrid representation for the HFS problem with the makespan criterion.
We next examine the number of consecutive iterations in which an individual is not improved ($\alpha$). From the means plot in Fig. 12, it is clear that not using the scout phase or $\alpha = \infty$ leads to a very bad DABC. For $\alpha$ equal to 30, 50, and 100, the DABC with $\alpha = 30$ produces the best results.

We proceed with the analysis, and we report the means plot of the replications of the employed bee stage ($\gamma$) in Fig. 13. Clearly, $\gamma = 10$, 20, and 30 lead to a better performing DABC than $\gamma = 1$, which suggests the effectiveness of the presented strategy to balance exploration and exploitation in Section 5.6. However, no significant difference is observed between $\gamma = 10$, 20, and 30.
The means plot of the number of neighbours generated for an abandoned individual at the scout phase ($\tau$) is reported in Fig. 14. As can be observed, the DABC with $\tau = 10, 20, \text{ or } 30$ yields much better results than with $\tau = 1$, which demonstrates the effectiveness of the enhanced strategy for the scout bee phase.

The means plot of the population size (PS) is reported in Fig. 15. Clearly, no significant difference is observed among the four tested levels: 20, 30, 40, and 50.
Finally, we set the parameters as follows according to the above analysis: $PS=30; \alpha=30; \gamma=20; \tau=30$. For the representation and solution generation method, we use the hybrid representation and insertion operators.

7 Comparative evaluations

7.1 Comparison based on benchmarks

Carlier and Neron [11] designed an HFS benchmark set that consisted of 77 instances and represented each instance by the number of jobs, the number of stages and the machine layout at the stages, where the machine layout is defined by the letters $a$, $b$, $c$, and $d$ with the following meanings:

- $a$: There is one machine at the middle stage and three machines at the other stages.
- $b$: There is one machine at the first stage and three machines at the other stages.
- $c$: There are two machines at the middle stage and three machines at the other stages.
- $d$: There are three machines at each stage.

For example, the notation $j15c10c1$ suggests a 10-stage HFS problem with 15 jobs. The letters $j$ and the first $c$ are abbreviations for the job and stage, respectively, and the second letter $c$ defines the structure of the machine layout at the stages; the last number 1 is the problem index for a specific type.

The 77 instances are divided into 53 easy problems and 24 hard problems, according to their machine configurations. The problems with $a$ and $b$ layouts are easier to solve, while the problems with $c$ and $d$ layouts are relatively hard; as a result, they are mostly grouped as hard problems.

The above benchmarks were originally established for testing the B&B method [11]. Thus, the problem size is limited, and the largest size has 15 jobs and 10 stages. To compare the different algorithms, Liao et al. [24] presented 10 much harder instances by extending the number of jobs from 15 to 30 and the number of parallel identical machines at each stage from 2 or 3 to a random number in the range of [3, 5], and the processing time distribution was changed from [3, 20] to [1, 99]. The representation of the instances in [24] is similar to [11].

7.1.1 Comparison based on the benchmark from [24]

This section compares the presented DABC with the particle swarm optimisation (PSO) [24] and the artificial immune approach (AIS) [19], which are the two best-performing algorithms for the HFS with the makespan criterion in the literature. We run our DABC on the aforementioned PC with a termination criterion set at the maximum elapsed CPU time of $t = 50 \text{ms}$ milliseconds. As was performed in [24], 20 independent replications were conducted for each of the 10 benchmark instances.
We report in Table 3 the minimum makespan, maximum makespan, average makespan, and standard deviation generated by the DABC, and the average computational time in which DABC finds the final solutions. For the results of the AIS and PSO, we take them directly from the literature [24]. The authors ran these two algorithms for each instance with 20 replications on a PC with Intel Core 2 Duo CPU P7350 2.0 GHz AND 2 GB RAM within a CPU time limit of 200 seconds.

It can be seen from Table 3 that the DABC yields much smaller average makespan values than both PSO and AIS for all of the instances. Additionally, the DABC further improves the current best solution that is found by the PSO for 9 out of 10 instances. A Z-test with a 0.05 significant level shows that the DABC performs statistically significantly better than the PSO and AIS for 8 and 10 instances, respectively. For the average computational time in which an algorithm finds the final solutions, although our computer is quite different from that used in [24], it is clear that our DABC is much faster because it spends only 3.86 seconds on average, while that of the PSO and AIS are 79.69 and 99.09 seconds, respectively. Thus, it can be concluded that the presented DABC outperforms the existing PSO and AIS at a considerable margin.
Table 3. Computational results of PSO, AIS and DABC (The best known solutions are in bold)

<table>
<thead>
<tr>
<th>Instances</th>
<th>PSO ave</th>
<th>PSO min</th>
<th>PSO max</th>
<th>PSO std</th>
<th>AIS ave</th>
<th>AIS min</th>
<th>AIS max</th>
<th>AIS std</th>
<th>DABC ave</th>
<th>DABC min</th>
<th>DABC max</th>
<th>DABC std</th>
<th>Significant?</th>
<th>CPU(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>j30c5e1</td>
<td>474.70</td>
<td>471</td>
<td>478</td>
<td>1.42</td>
<td>485.35</td>
<td>479</td>
<td>488</td>
<td>2.58</td>
<td>466.35</td>
<td>464</td>
<td>469</td>
<td>1.42</td>
<td>Y</td>
<td>96.16</td>
</tr>
<tr>
<td>j30c5e2</td>
<td>616.25</td>
<td>616</td>
<td>617</td>
<td>0.44</td>
<td>620.70</td>
<td>619</td>
<td>625</td>
<td>1.63</td>
<td>616.00</td>
<td>616</td>
<td>616</td>
<td>0.00</td>
<td>N</td>
<td>55.28</td>
</tr>
<tr>
<td>j30c5e3</td>
<td>610.25</td>
<td>602</td>
<td>619</td>
<td>4.70</td>
<td>625.70</td>
<td>614</td>
<td>635</td>
<td>4.81</td>
<td>598.45</td>
<td>596</td>
<td>600</td>
<td>1.10</td>
<td>Y</td>
<td>64.56</td>
</tr>
<tr>
<td>j30c5e4</td>
<td>577.10</td>
<td>575</td>
<td>580</td>
<td>1.52</td>
<td>588.55</td>
<td>582</td>
<td>594</td>
<td>3.38</td>
<td>567.45</td>
<td>566</td>
<td>571</td>
<td>1.47</td>
<td>Y</td>
<td>108.63</td>
</tr>
<tr>
<td>j30c5e5</td>
<td>606.80</td>
<td>605</td>
<td>609</td>
<td>1.11</td>
<td>618.75</td>
<td>610</td>
<td>624</td>
<td>3.42</td>
<td>604.65</td>
<td>603</td>
<td>606</td>
<td>0.67</td>
<td>N</td>
<td>79.84</td>
</tr>
<tr>
<td>j30c5e6</td>
<td>612.50</td>
<td>605</td>
<td>619</td>
<td>3.49</td>
<td>625.75</td>
<td>620</td>
<td>631</td>
<td>3.01</td>
<td>616.75</td>
<td>613</td>
<td>608</td>
<td>1.74</td>
<td>Y</td>
<td>67.99</td>
</tr>
<tr>
<td>j30c5e7</td>
<td>630.60</td>
<td>629</td>
<td>632</td>
<td>0.75</td>
<td>641.30</td>
<td>635</td>
<td>659</td>
<td>4.67</td>
<td>626.35</td>
<td>626</td>
<td>628</td>
<td>0.75</td>
<td>Y</td>
<td>87.18</td>
</tr>
<tr>
<td>j30c5e8</td>
<td>684.20</td>
<td>678</td>
<td>688</td>
<td>2.50</td>
<td>697.50</td>
<td>686</td>
<td>704</td>
<td>5.14</td>
<td>675.25</td>
<td>674</td>
<td>680</td>
<td>1.59</td>
<td>Y</td>
<td>97.67</td>
</tr>
<tr>
<td>j30c5e9</td>
<td>654.65</td>
<td>651</td>
<td>658</td>
<td>1.87</td>
<td>670.20</td>
<td>662</td>
<td>676</td>
<td>3.85</td>
<td>645.50</td>
<td>643</td>
<td>647</td>
<td>1.05</td>
<td>Y</td>
<td>83.80</td>
</tr>
<tr>
<td>j30c5e10</td>
<td>599.75</td>
<td>594</td>
<td>617</td>
<td>5.28</td>
<td>613.45</td>
<td>604</td>
<td>619</td>
<td>5.33</td>
<td>578.95</td>
<td>575</td>
<td>582</td>
<td>1.99</td>
<td>Y</td>
<td>77.46</td>
</tr>
<tr>
<td>Mean</td>
<td>606.68</td>
<td>602.6</td>
<td>611.7</td>
<td>2.31</td>
<td>618.73</td>
<td>611.1</td>
<td>625.5</td>
<td>3.78</td>
<td>599.78</td>
<td>596.60</td>
<td>600.70</td>
<td>1.18</td>
<td>Y</td>
<td>79.69</td>
</tr>
</tbody>
</table>

**CPU(s)**

<table>
<thead>
<tr>
<th>PSO</th>
<th>AIS</th>
<th>DABC</th>
</tr>
</thead>
<tbody>
<tr>
<td>96.16</td>
<td>99.44</td>
<td>4.61</td>
</tr>
</tbody>
</table>
7.1.2 Comparison based on the benchmark from [11]

We use the presented DABC to solve the 77 instances of Carlier and Neron [11]. Because the problem size is very limited, we stop our algorithm at the very short maximum running time of
\[ t = 2 \times 10^3 \text{ milliseconds.} \]

The DABC is run 20 times to obtain the best makespan from the replications. For the other compared algorithms, i.e., the particle swarm optimisation [24], the artificial immune approach (AIS) [19], the ant colony optimisation (ACO) [20], the quantum-inspired immune algorithm (QIA) [23], and the branch and bound (B&B) [12], we obtain the computational results from their original paper. Note that the PSO, AIS, ACO, and B&B limited their run time to 1600 s and also accepted the best makespan from 20 replications as the solution. The computational results are summarised in Table 4. Similar to in [24], we also report the percentage deviation between the solution \( C_{\text{max}} \) and the lower bound (LB), as follows:

\[
\%\text{deviation} = \left( \frac{C_{\text{max}} - LB}{LB} \right) \times 100
\]

<table>
<thead>
<tr>
<th>Instances</th>
<th>DABC</th>
<th>PSO</th>
<th>QIA</th>
<th>ACO</th>
<th>AIS</th>
<th>B&amp;B</th>
<th>LB</th>
<th>DABC</th>
<th>PSO</th>
<th>QIA</th>
<th>ACO</th>
<th>AIS</th>
<th>B&amp;B</th>
</tr>
</thead>
<tbody>
<tr>
<td>j10c5a2</td>
<td>88</td>
<td>88</td>
<td>88</td>
<td>88</td>
<td>88</td>
<td>88</td>
<td>88</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>j10c5a3</td>
<td>117</td>
<td>117</td>
<td>117</td>
<td>117</td>
<td>117</td>
<td>117</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>j10c5a4</td>
<td>121</td>
<td>121</td>
<td>121</td>
<td>121</td>
<td>121</td>
<td>121</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>j10c5a5</td>
<td>122</td>
<td>122</td>
<td>122</td>
<td>124</td>
<td>122</td>
<td>122</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1.64</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>j10c5a6</td>
<td>110</td>
<td>110</td>
<td>110</td>
<td>110</td>
<td>110</td>
<td>110</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>j10c5b1</td>
<td>130</td>
<td>130</td>
<td>130</td>
<td>131</td>
<td>130</td>
<td>130</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.77</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>j10c5b2</td>
<td>107</td>
<td>107</td>
<td>107</td>
<td>107</td>
<td>107</td>
<td>107</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>j10c5b3</td>
<td>109</td>
<td>109</td>
<td>109</td>
<td>109</td>
<td>109</td>
<td>109</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>j10c5b4</td>
<td>122</td>
<td>122</td>
<td>122</td>
<td>122</td>
<td>122</td>
<td>122</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1.64</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>j10c5b5</td>
<td>153</td>
<td>153</td>
<td>153</td>
<td>153</td>
<td>153</td>
<td>153</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>j10c5b6</td>
<td>115</td>
<td>115</td>
<td>115</td>
<td>115</td>
<td>115</td>
<td>115</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>j10c5c1</td>
<td>68</td>
<td>68</td>
<td>68</td>
<td>68</td>
<td>68</td>
<td>68</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1.47</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>j10c5c2</td>
<td>72</td>
<td>72</td>
<td>72</td>
<td>72</td>
<td>72</td>
<td>72</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2.7</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>j10c5c3</td>
<td>71</td>
<td>71</td>
<td>71</td>
<td>71</td>
<td>71</td>
<td>71</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4.23</td>
<td>1.41</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>j10c5c4</td>
<td>66</td>
<td>66</td>
<td>66</td>
<td>66</td>
<td>66</td>
<td>66</td>
<td>0</td>
<td>0</td>
<td>13.6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>j10c5c5</td>
<td>78</td>
<td>78</td>
<td>78</td>
<td>78</td>
<td>78</td>
<td>78</td>
<td>0</td>
<td>0</td>
<td>12.8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>j10c5c6</td>
<td>69</td>
<td>69</td>
<td>69</td>
<td>69</td>
<td>69</td>
<td>69</td>
<td>0</td>
<td>0</td>
<td>4.35</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>j10c5d1</td>
<td>66</td>
<td>66</td>
<td>66</td>
<td>66</td>
<td>66</td>
<td>66</td>
<td>0</td>
<td>0</td>
<td>13.6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>j10c5d2</td>
<td>78</td>
<td>78</td>
<td>78</td>
<td>78</td>
<td>78</td>
<td>78</td>
<td>0</td>
<td>0</td>
<td>12.8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>j10c5d3</td>
<td>69</td>
<td>69</td>
<td>69</td>
<td>69</td>
<td>69</td>
<td>69</td>
<td>0</td>
<td>0</td>
<td>4.35</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>j10c5d4</td>
<td>72</td>
<td>72</td>
<td>72</td>
<td>72</td>
<td>72</td>
<td>72</td>
<td>0</td>
<td>0</td>
<td>4.23</td>
<td>1.41</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>j10c5d5</td>
<td>71</td>
<td>71</td>
<td>71</td>
<td>71</td>
<td>71</td>
<td>71</td>
<td>0</td>
<td>0</td>
<td>2.7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>j10c5d6</td>
<td>66</td>
<td>66</td>
<td>66</td>
<td>66</td>
<td>66</td>
<td>66</td>
<td>0</td>
<td>0</td>
<td>13.6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>j10c10a1</td>
<td>139</td>
<td>139</td>
<td>139</td>
<td>139</td>
<td>139</td>
<td>139</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>j10c10a2</td>
<td>158</td>
<td>158</td>
<td>158</td>
<td>158</td>
<td>158</td>
<td>158</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>j10c10a3</td>
<td>148</td>
<td>148</td>
<td>148</td>
<td>148</td>
<td>148</td>
<td>148</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>j10c10a4</td>
<td>149</td>
<td>149</td>
<td>149</td>
<td>149</td>
<td>149</td>
<td>149</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>j10c10a5</td>
<td>148</td>
<td>148</td>
<td>148</td>
<td>148</td>
<td>148</td>
<td>148</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>j10c10a6</td>
<td>146</td>
<td>146</td>
<td>146</td>
<td>146</td>
<td>146</td>
<td>146</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>j10c10b1</td>
<td>163</td>
<td>163</td>
<td>163</td>
<td>163</td>
<td>163</td>
<td>163</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>j10c10b2</td>
<td>157</td>
<td>157</td>
<td>157</td>
<td>157</td>
<td>157</td>
<td>157</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>j10c10b3</td>
<td>169</td>
<td>169</td>
<td>169</td>
<td>169</td>
<td>169</td>
<td>169</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>j10c10b4</td>
<td>159</td>
<td>159</td>
<td>159</td>
<td>159</td>
<td>159</td>
<td>159</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>j10c10b5</td>
<td>165</td>
<td>165</td>
<td>165</td>
<td>165</td>
<td>165</td>
<td>165</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>j10c10b6</td>
<td>165</td>
<td>165</td>
<td>165</td>
<td>165</td>
<td>165</td>
<td>165</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>j15c5a1</td>
<td>178</td>
<td>178</td>
<td>178</td>
<td>178</td>
<td>178</td>
<td>178</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
The performances of all of the compared algorithms are summarised in Table 5. For the easy problems, it can be seen from Table 5 that the DABC solves to optimality 48 of the 53 problems (90.6%), with an 0.89% overall average deviation. However, the PSO, AIS and B&B can solve 47 of the 53 problems (88.7%) with a larger deviation that is equal to 0.95%, 0.99%, and 2.12%, respectively. The ACO can solve 29 of the 45 problems (64.4%) with a 0.92% overall average deviation. The QIA can solve 29 problems to optimality. However, no results of the QIA for the remaining 24 problems were reported. For the hard problems, the DABC solves 18 of the 24 problems (75%) and has the smallest average deviation of 2.85%. The same results are generated by the PSO. For the other algorithms, both the solved problems and overall average deviations are much worse than those by the DABC. In a total, the DABC finds the optima for more instances with smaller overall deviations than the PSO, AIS and B&B. Thus, it can be concluded that the presented DABC outperforms all of the compared algorithms in solving the existing HFS benchmark of Carlier and Neron [11].

### Table 5 Performance summary of different algorithms

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>% solved</th>
<th>Easy problems</th>
<th>Hard problems</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>% solved</td>
<td>%dev</td>
<td># of prob.</td>
</tr>
<tr>
<td>DABC</td>
<td>90.6</td>
<td>0.89</td>
<td>53</td>
</tr>
<tr>
<td>PSO</td>
<td>88.7</td>
<td>0.95</td>
<td>53</td>
</tr>
<tr>
<td>QIA</td>
<td>100.0</td>
<td>0.00</td>
<td>29</td>
</tr>
<tr>
<td>ACO</td>
<td>64.4</td>
<td>0.92</td>
<td>45</td>
</tr>
<tr>
<td>AIS</td>
<td>88.7</td>
<td>0.99</td>
<td>53</td>
</tr>
<tr>
<td>B&amp;B</td>
<td>88.7</td>
<td>2.17</td>
<td>53</td>
</tr>
</tbody>
</table>

7.2 Comparison based on randomly generated instances
This section re-implements seven effective metaheuristics that were published recently for the HFS problems and conducts a series of experiments that are based on 150 instances that are randomly generated by using a method similar to the method in Section 6.1. The metaheuristics are the artificial immune system (AIS) [19], the ant colony algorithm (ACO) [20], the genetic algorithm (GA K) [21], the genetic algorithm (GA R) [1], the improved simulated annealing (ISA) [28], the iterated local search (ILS) [2], and the artificial bee colony algorithm (ABC P) [3]. The AIS, ACO, and GA K were presented with the same problems that were considered in this paper, whereas the GA R, ILS, ISA, and ABC P were presented with more complex HFS problems. All of the algorithms were demonstrated to be very effective on their original problems. We apply the GA R, ILS, ISA, and ABC P to the problem that is under consideration by modifying the objective evaluation. For the parameters of the seven metaheuristics, we directly take them from the literature. All of the algorithms adopt the same maximum elapsed CPU time limit of $t = \rho \cdot n$ seconds as a termination criterion, where $\rho$ has been tested at three values: 10, 20 and 30. All of the algorithms have been coded in Visual C++ 6.0, and the experiments are performed on the same PC as mentioned above. We conducted 20 independent runs for each algorithm for each of the 150 instances. We adopt the minimal makespan found at $\rho = 30$ as $c_{\text{min}}$ to compute the RPI. The computed results, averaged across the 20 replications for each instance and grouped for each subset, are reported in Tables 6-8. (The minimum RPI values are in bold.)

### Table 6. Computational results of the algorithms ($\rho = 10$)

<table>
<thead>
<tr>
<th>Instances</th>
<th>DABC</th>
<th>ILS</th>
<th>ISA</th>
<th>ABC P</th>
<th>AIS</th>
<th>ACO</th>
<th>GA R</th>
<th>GA K</th>
</tr>
</thead>
<tbody>
<tr>
<td>20×5</td>
<td>0.03</td>
<td>0.45</td>
<td>0.62</td>
<td>0.43</td>
<td>0.71</td>
<td>5.85</td>
<td>0.99</td>
<td>2.01</td>
</tr>
<tr>
<td>20×8</td>
<td>0.12</td>
<td>0.54</td>
<td>0.87</td>
<td>0.81</td>
<td>1.32</td>
<td>8.68</td>
<td>1.34</td>
<td>3.97</td>
</tr>
<tr>
<td>20×10</td>
<td>0.34</td>
<td>0.56</td>
<td>1.08</td>
<td>0.58</td>
<td>1.29</td>
<td>6.54</td>
<td>1.24</td>
<td>3.61</td>
</tr>
<tr>
<td>40×5</td>
<td>0.03</td>
<td>0.16</td>
<td>0.24</td>
<td>0.42</td>
<td>0.38</td>
<td>4.39</td>
<td>0.54</td>
<td>1.40</td>
</tr>
<tr>
<td>40×8</td>
<td>0.03</td>
<td>0.14</td>
<td>0.26</td>
<td>0.49</td>
<td>0.52</td>
<td>5.33</td>
<td>0.73</td>
<td>2.72</td>
</tr>
<tr>
<td>40×10</td>
<td>0.16</td>
<td>0.65</td>
<td>0.94</td>
<td>1.29</td>
<td>1.34</td>
<td>6.37</td>
<td>1.44</td>
<td>3.43</td>
</tr>
<tr>
<td>60×5</td>
<td>0.09</td>
<td>0.19</td>
<td>0.29</td>
<td>0.36</td>
<td>0.47</td>
<td>5.26</td>
<td>0.59</td>
<td>1.88</td>
</tr>
<tr>
<td>60×8</td>
<td>0.15</td>
<td>0.27</td>
<td>0.38</td>
<td>0.62</td>
<td>0.59</td>
<td>3.32</td>
<td>0.71</td>
<td>1.96</td>
</tr>
<tr>
<td>60×10</td>
<td>0.20</td>
<td>0.40</td>
<td>0.60</td>
<td>0.81</td>
<td>0.90</td>
<td>3.54</td>
<td>1.01</td>
<td>2.58</td>
</tr>
<tr>
<td>80×5</td>
<td>0.01</td>
<td>0.18</td>
<td>0.14</td>
<td>0.29</td>
<td>0.26</td>
<td>2.31</td>
<td>0.38</td>
<td>0.96</td>
</tr>
<tr>
<td>80×8</td>
<td>0.15</td>
<td>0.26</td>
<td>0.44</td>
<td>0.58</td>
<td>0.59</td>
<td>3.59</td>
<td>0.78</td>
<td>2.23</td>
</tr>
<tr>
<td>80×10</td>
<td>0.09</td>
<td>0.30</td>
<td>0.29</td>
<td>0.46</td>
<td>0.57</td>
<td>3.29</td>
<td>0.64</td>
<td>2.31</td>
</tr>
<tr>
<td>100×5</td>
<td>0.00</td>
<td>0.04</td>
<td>0.02</td>
<td>0.12</td>
<td>0.04</td>
<td>1.22</td>
<td>0.17</td>
<td>0.38</td>
</tr>
<tr>
<td>100×8</td>
<td>0.08</td>
<td>0.24</td>
<td>0.18</td>
<td>0.58</td>
<td>0.35</td>
<td>2.37</td>
<td>0.46</td>
<td>1.40</td>
</tr>
<tr>
<td>100×10</td>
<td>0.11</td>
<td>0.24</td>
<td>0.24</td>
<td>0.43</td>
<td>0.47</td>
<td>2.71</td>
<td>0.57</td>
<td>1.95</td>
</tr>
<tr>
<td>Average</td>
<td>0.11</td>
<td>0.31</td>
<td>0.44</td>
<td>0.55</td>
<td>0.65</td>
<td>4.32</td>
<td>0.77</td>
<td>2.19</td>
</tr>
</tbody>
</table>

### Table 7. Computational results of the algorithms ($\rho = 20$)

<table>
<thead>
<tr>
<th>Instances</th>
<th>DABC</th>
<th>ILS</th>
<th>ISA</th>
<th>ABC P</th>
<th>AIS</th>
<th>ACO</th>
<th>GA R</th>
<th>GA K</th>
</tr>
</thead>
<tbody>
<tr>
<td>20×5</td>
<td>0.02</td>
<td>0.35</td>
<td>0.52</td>
<td>0.32</td>
<td>0.63</td>
<td>5.81</td>
<td>0.86</td>
<td>1.92</td>
</tr>
<tr>
<td>20×8</td>
<td>0.06</td>
<td>0.38</td>
<td>0.75</td>
<td>0.69</td>
<td>1.21</td>
<td>8.62</td>
<td>1.16</td>
<td>3.72</td>
</tr>
<tr>
<td>20×10</td>
<td>0.26</td>
<td>0.43</td>
<td>0.97</td>
<td>0.46</td>
<td>1.19</td>
<td>6.51</td>
<td>1.01</td>
<td>3.31</td>
</tr>
<tr>
<td>40×5</td>
<td>0.02</td>
<td>0.13</td>
<td>0.20</td>
<td>0.27</td>
<td>0.26</td>
<td>4.30</td>
<td>0.45</td>
<td>1.33</td>
</tr>
<tr>
<td>40×8</td>
<td>0.01</td>
<td>0.05</td>
<td>0.20</td>
<td>0.32</td>
<td>0.41</td>
<td>5.26</td>
<td>0.49</td>
<td>2.53</td>
</tr>
<tr>
<td>40×10</td>
<td>0.12</td>
<td>0.40</td>
<td>0.83</td>
<td>1.00</td>
<td>1.17</td>
<td>6.33</td>
<td>1.20</td>
<td>3.25</td>
</tr>
<tr>
<td>60×5</td>
<td>0.04</td>
<td>0.14</td>
<td>0.23</td>
<td>0.33</td>
<td>0.40</td>
<td>5.22</td>
<td>0.49</td>
<td>1.79</td>
</tr>
<tr>
<td>60×8</td>
<td>0.08</td>
<td>0.19</td>
<td>0.31</td>
<td>0.54</td>
<td>0.63</td>
<td>3.31</td>
<td>0.59</td>
<td>1.85</td>
</tr>
<tr>
<td>60×10</td>
<td>0.14</td>
<td>0.25</td>
<td>0.53</td>
<td>0.68</td>
<td>0.81</td>
<td>3.50</td>
<td>0.85</td>
<td>2.43</td>
</tr>
<tr>
<td>80×5</td>
<td>0.01</td>
<td>0.15</td>
<td>0.10</td>
<td>0.28</td>
<td>0.22</td>
<td>2.30</td>
<td>0.33</td>
<td>0.93</td>
</tr>
<tr>
<td>80×8</td>
<td>0.09</td>
<td>0.18</td>
<td>0.37</td>
<td>0.42</td>
<td>0.30</td>
<td>3.54</td>
<td>0.65</td>
<td>2.04</td>
</tr>
<tr>
<td>80×10</td>
<td>0.06</td>
<td>0.25</td>
<td>0.24</td>
<td>0.41</td>
<td>0.47</td>
<td>3.25</td>
<td>0.51</td>
<td>2.17</td>
</tr>
<tr>
<td>100×5</td>
<td>0.00</td>
<td>0.03</td>
<td>0.01</td>
<td>0.11</td>
<td>0.03</td>
<td>1.20</td>
<td>0.12</td>
<td>0.33</td>
</tr>
<tr>
<td>100×8</td>
<td>0.03</td>
<td>0.20</td>
<td>0.14</td>
<td>0.50</td>
<td>0.27</td>
<td>2.34</td>
<td>0.39</td>
<td>1.27</td>
</tr>
<tr>
<td>100×10</td>
<td>0.08</td>
<td>0.20</td>
<td>0.21</td>
<td>0.39</td>
<td>0.36</td>
<td>2.67</td>
<td>0.46</td>
<td>1.81</td>
</tr>
</tbody>
</table>
Table 8. Computational results of the algorithms ( $\rho =30$)

<table>
<thead>
<tr>
<th>Instances</th>
<th>DABC</th>
<th>ILS</th>
<th>ISA</th>
<th>ABC $\tau$</th>
<th>AIS</th>
<th>ACO</th>
<th>GAR</th>
<th>GA $K$</th>
</tr>
</thead>
<tbody>
<tr>
<td>20×5</td>
<td>0.01</td>
<td>0.28</td>
<td>0.49</td>
<td>0.32</td>
<td>0.60</td>
<td>5.74</td>
<td>0.80</td>
<td>1.86</td>
</tr>
<tr>
<td>20×8</td>
<td>0.04</td>
<td>0.34</td>
<td>0.69</td>
<td>0.61</td>
<td>1.18</td>
<td>8.52</td>
<td>1.05</td>
<td>3.57</td>
</tr>
<tr>
<td>20×10</td>
<td>0.19</td>
<td>0.36</td>
<td>0.90</td>
<td>0.43</td>
<td>1.10</td>
<td>6.45</td>
<td>0.91</td>
<td>3.17</td>
</tr>
<tr>
<td>40×5</td>
<td>0.01</td>
<td>0.11</td>
<td>0.17</td>
<td>0.19</td>
<td>0.25</td>
<td>4.26</td>
<td>0.39</td>
<td>1.30</td>
</tr>
<tr>
<td>40×8</td>
<td>0.01</td>
<td>0.03</td>
<td>0.16</td>
<td>0.27</td>
<td>0.36</td>
<td>5.23</td>
<td>0.35</td>
<td>2.41</td>
</tr>
<tr>
<td>40×10</td>
<td>0.09</td>
<td>0.33</td>
<td>0.74</td>
<td>0.79</td>
<td>1.09</td>
<td>6.27</td>
<td>1.04</td>
<td>3.15</td>
</tr>
<tr>
<td>60×5</td>
<td>0.03</td>
<td>0.11</td>
<td>0.20</td>
<td>0.29</td>
<td>0.48</td>
<td>5.20</td>
<td>0.46</td>
<td>1.76</td>
</tr>
<tr>
<td>60×8</td>
<td>0.05</td>
<td>0.16</td>
<td>0.28</td>
<td>0.47</td>
<td>0.58</td>
<td>3.31</td>
<td>0.53</td>
<td>1.78</td>
</tr>
<tr>
<td>60×10</td>
<td>0.12</td>
<td>0.21</td>
<td>0.50</td>
<td>0.61</td>
<td>0.75</td>
<td>3.49</td>
<td>0.79</td>
<td>2.35</td>
</tr>
<tr>
<td>80×5</td>
<td>0.01</td>
<td>0.14</td>
<td>0.10</td>
<td>0.27</td>
<td>0.20</td>
<td>2.29</td>
<td>0.29</td>
<td>0.89</td>
</tr>
<tr>
<td>80×8</td>
<td>0.06</td>
<td>0.14</td>
<td>0.33</td>
<td>0.36</td>
<td>0.47</td>
<td>3.53</td>
<td>0.56</td>
<td>1.97</td>
</tr>
<tr>
<td>80×10</td>
<td>0.05</td>
<td>0.24</td>
<td>0.21</td>
<td>0.39</td>
<td>0.44</td>
<td>3.25</td>
<td>0.46</td>
<td>2.10</td>
</tr>
<tr>
<td>100×5</td>
<td>0.00</td>
<td>0.02</td>
<td>0.01</td>
<td>0.10</td>
<td>0.02</td>
<td>1.20</td>
<td>0.11</td>
<td>0.32</td>
</tr>
<tr>
<td>100×8</td>
<td>0.02</td>
<td>0.19</td>
<td>0.12</td>
<td>0.44</td>
<td>0.26</td>
<td>2.32</td>
<td>0.37</td>
<td>1.20</td>
</tr>
<tr>
<td>100×10</td>
<td>0.07</td>
<td>0.19</td>
<td>0.19</td>
<td>0.36</td>
<td>0.34</td>
<td>2.65</td>
<td>0.41</td>
<td>1.73</td>
</tr>
<tr>
<td>Average</td>
<td>0.05</td>
<td>0.19</td>
<td>0.34</td>
<td>0.39</td>
<td>0.54</td>
<td>4.25</td>
<td>0.57</td>
<td>1.97</td>
</tr>
</tbody>
</table>

It is clear from Tables 6-8 that the presented DABC algorithm is the winner in terms of the overall solution quality. For the shortest CPU time, $\rho =10$, the DABC generates the lowest overall average RPI (ARPI), which is equal to 0.11% and is much smaller than those generated by the ILS (0.31%), ISA (0.44%), ABC $\tau$ (0.55%), AIS (0.65%), ACO (4.32%), GAR (0.77%) and GA $K$ (2.19%). For all of the 15 instance sizes, the DABC achieves the minimum RPI values and surpasses the others at a considerable margin. The ILS produces the second lowest ARPI value of 0.31%, which is much better than the third value of 0.44% provided by the ISA. The recently presented ABC $\tau$ ranks fourth, with 0.55% as the ARPI value. The worst two performing algorithms are the ACO and GA $K$, which had much larger ARPI values equal to 4.32% and 2.19%, respectively. For the CPU time $\rho = 20$ and $\rho = 30$, we find from Tables 7 and 8 that all of the algorithms improve their results with additional elapsed CPU time. Additionally, the DABC again beats all of the other algorithms at a considerable margin. Hence, it is concluded that the DABC is very effective, and on average, it outperforms the ILS, ISA, ABC $\tau$, AIS, ACO, GAR and GA $K$ for the problem under consideration.

We further perform a multifactor ANOVA to compare the six best performing algorithms, i.e., DABC, ILS, ISA, ABC $\tau$, AIS, and GA $K$, and we consider the type of algorithm and $\rho$ as factors. We take the ACO and GA $K$ out of the experiments because their performance is substantially worse. Fig. 16 reports the means and 95% Tukey HSD confidence intervals of the interaction between the type of algorithms and the allowed CPU time $\rho$. From the figure, it is clear that the DABC performs significantly better than the other algorithms at different elapsed CPU times.
From the above comparisons, it can be concluded that the presented DABC algorithm is a new state-of-the-art algorithm for the hybrid flowshop scheduling problem with the objective of minimising the makespan.

8 Conclusions

This paper addressed the HFS problem with the makespan criterion, which has important applications in glass, steel, paper, textile, and many other industries. To propose an effective artificial bee colony algorithm, we first presented a total of 24 heuristics, which were utilised to generate an initial population that had a high level of quality and diversity. Then, we presented a hybrid representation by combining the forward decoding and backward decoding methods. As far as we know, this approach is a novel representation for the HFS problem. Afterward, we introduced a new control parameter to balance the exploration and exploitation and, in addition, an enhanced strategy for the scout bee phase (to prevent the algorithm from searching in poor regions) and a problem-specific local refinement procedure to search for solution spaces that were unexplored by the honey bees. We obtained an effective DABC by calibrating the parameters and operators by means of a design of experiments approach. Finally, a comparative evaluation was conducted with the best performing algorithms presented for the HFS problem under consideration and with the adaptations of some state-of-the-art metaheuristics that were originally designed for other HFS problems. The results showed that the DABC outperforms the other algorithms when solving the HFS problem with the makespan criterion, at a substantial margin.

Future work could focus on further exploration of problem-specific characteristics and developing more effective heuristic methods, neighbouring solution generation operators, and local search procedures for the HFS problem. Moreover, due to the effectiveness of the DABC, it could be interesting to analyse its performance by using other objective criteria such as the total flowtime and to generalise its application to other combinatorial optimisation problems, including the blocking.
flowshop, lot-streaming flowshop problem, and examination timetabling problem [46-51]. Certain realistic scheduling problems, such as the steelmaking-continuous casting problem from the modern steelmaking industry, appear to be a promising venue of research for the application of the techniques studied in this paper. Of course, each problem would need special tailoring and experimentation, which would furnish the basis for future research.

Acknowledgments

The authors would like to thank Jasques Carlier and Emmanuel Neron, Ching-Jong Liao, Evi Tjandradjaja, and Tsui-Ping Chung for their benchmark problems and valuable help. This research is partially supported by National Science Foundation of China under Grants 61174187, 61174189, 61104179, Program for New Century Excellent Talents in University (NCET-13-0106), Science Foundation of Liaoning Province in China (2013020016), Basic Scientific Research Foundation of Northeast University under Grant N110208001, Starting Foundation of Northeast University under Grant 29321006, Doctoral Program Foundation of Institutions of Higher Education of China (20130002110057, 20130042110035), IAPI Fundamental Research Funds (2013ZCX04-04), and Shandong Province Key Laboratory of Intelligent Information Processing and Network Security (Liaocheng University).
References


