Contents lists available at GrowingScience

International Journal of Industrial Engineering Computations

homepage: www.GrowingScience.com/ijiec

Modeling and optimization of the hybrid flow shop scheduling problem with sequence-dependent setup times

Huiting Xue[a], Leilei Meng[a*], Peng Duan[a], Biao Zhang[a], Wenqiang Zou[a] and Hongyan Sang[a]

[a]*School of Computer Science, Liaocheng University, Liaocheng 252000, China*

C H R O N I C L E

Article history:
Received October 20 2023
Received in Revised Format December 18 2023
Accepted January 4 2024
Available online January 5 2024

Keywords:
Hybrid flow shop scheduling problem
Sequence-dependent setup times
Artificial bee colony algorithm
Mixed-integer linear programming

A B S T R A C T

The hybrid flow shop scheduling problem (HFSP) is an extension of the classic flow shop scheduling problem and widely exists in real industrial production systems. In real production, sequence-dependent setup times (SDST) are very important and cannot be neglected. Therefore, this study focuses HFSP with SDST (HFSP-SDST) to minimize the makespan. To solve this problem, a mixed-integer linear programming (MILP) model to obtain the optimal solutions for small-scale instances is proposed. Given the NP-hard characteristics of HFSP-SDST, an improved artificial bee colony (IABC) algorithm is developed to efficiently solve large-sized instances. In IABC, permutation encoding is used and a hybrid representation that combines forward decoding and backward decoding methods is designed. To search for the solution space that is not included in the encoding and decoding, a problem-specific local search strategy is developed to enlarge the solution space. Experiments are conducted to evaluate the effectiveness of the MILP model and IABC. The results indicate that the proposed MILP model can find the optimal solutions for small-scale instances. The proposed IABC performs much better than the existing algorithms and improves 61 current best solutions of benchmark instances.

1. Introduction

In the current era of rapid informatization, intelligent manufacturing is advancing rapidly, becoming a key initiative driving high-quality development in the manufacturing industry. Manufacturing firms can enhance product quality and equipment utilization through scientifically effective scheduling strategies, thereby bolstering market competitiveness and realizing greater economic benefits. Flow shop scheduling has become one of the most crucial topics in intelligent factory technologies, directly influencing production efficiency and product quality. Hybrid flow shop scheduling problem (HFSP) is a prevalent type of production scheduling problem in industries such as textiles, steel, and metallurgy (Pan, Wang, Mao, Zhao, & Zhang, 2012). In HFSP, multiple production stages typically exist, each stage possibly having multiple identical or different machines, and jobs need to be processed in a certain sequence across these stages. In fact, HFSP with even two stages has proved to be NP-Hard (Garey & Johnson, 1978). However, in actual production workshops, due to process constraints, certain necessary non-productive operations such as equipment cleaning and material loading need to be carried out in advance when the job is being processed. The execution of these operations results in additional time consumption, known as setup times. Based on a review of literature on similar problems published in recent years, there are three different types of setup times: first, the setup time depends on the job to be processed and is independent of the sequence; second, the setup time depends on the processing sequence of the jobs and is sequence-dependent; third, the setup time is determined by the order of the jobs and the machine on which they are processed. Setup times typically occur before job processing and are independent of processing times. The setup time studied in this paper is related to the job processing sequence (Jungwattanakit, Reodecha, Chaovalitwongse, & Werner, 2005). The setup time studied in this paper is related to the processing sequence, and its significance has been demonstrated (Allahverdi, 2015). References (He, Pan, Gao, Neufeld, & Gupta, 2024; Meng et al., 2022) indicated that setup

* Corresponding author
E-mail mengleilei@lcu-cs.com (L. Meng)

ISSN 1923-2934 (Online) - ISSN 1923-2926 (Print)
2024 Growing Science Ltd.
doi: 10.5267/j.ijiec.2024.1.001

time is a crucial element in production scheduling. Properly arranging sequence-dependent setup times (SDST) can effectively reduce idle time on the production line and machine waiting times, enhance resource utilization, reduce production costs, and boost production efficiency. Therefore, HFSP with SDST (HFSP-SDST) is of significant research value. The research of this paper focuses on the HFSP-SDST (Kurz & Askin, 2003). The primary objective is to minimize the maximum makespan. The scheduling task of HFSP-SDST includes job sequencing and machine assignment, representing a typical combinatorial optimization problem.

The artificial bee colony (ABC) algorithm is a swarm intelligence optimization algorithm that simulates the process of honeybees searching for food (Dai, Pan, Miao, Suganthan, & Gao, 2023). Due to its robustness and strong global search capabilities, ABC has been widely applied to solve HFSP-SDST. However, ABC's limitations, such as poor local search capabilities and slow convergence speed, restrict the solution space and consequently affect its efficiency. Its encoding and decoding methods are very important and determine the solution space. This paper proposes an improved artificial bee colony algorithm (IABC) based on traditional ABC. In IABC, a problem-specific local search (LS) strategy and a hybrid decoding method are developed to better explore the solution space and enhance algorithm efficiency. Although the ABC algorithm has proven to be effective, it is only an approximate solution algorithm, and it cannot guarantee to obtain an optimal solution even for very small-scale instances. Therefore, this paper also develops a mixed-integer linear programming model (MILP) to obtain optimal solutions for small-scale instances. Specifically, the proposed IABC in this paper improved 61 out of 120 benchmark instances from the reference (Pan, Gao, Li, & Gao, 2017). The contributions of this research are as follows:

• A MILP model is proposed to solve the small-sized instances of HFSP-SDST to optimality.
• A hybrid decoding representation method is introduced by combining the strengths of both forward and backward decoding approaches.
• A LS process based on hybrid decoding is proposed, aiming to explore the neighborhood of the solution space more thoroughly.
• The proposed IABC improves 61 current best solutions of benchmark instances.

The structure of the paper is as follows: Section 2 provides an extensive literature review. In Section 3, the problem description along with the MILP is presented. Section 4 introduces our proposed method. Section 5 includes parameter testing for the proposed method, along with extensive experiments and data analysis, evaluating the effectiveness of the proposed approach in solving the HFSP-SDST problem. Finally, Section 6 summarizes and concludes the paper.

2 Literature Review

A review of some relevant literature pertaining to the problem considered in this paper is conducted. In the early stages, research primarily focused on relatively small-scale HFSP, and more precise algorithms were employed, such as Branch and Bound (B&B) (Fattahi, Hosseini, Jolai, & Tavakkoli-Moghaddam, 2014) and mixed-integer programming (MIP) (Meng, Zhang, Ren, Zhang, & Lv, 2020; Meng et al., 2023). As the scale of HFSP increased, these exact algorithms proved inadequate for the best resolution, leading to the emergence of various improved algorithms. Later on, HFSP gave rise to many more realistic variants, with SDST being one of them, garnering widespread attention from researchers.

2.1 The literature review on HFSP

The earliest research on HFSP appeared in a paper from the 1970s. The algorithm proposed in the paper has been successfully applied to the nylon polymerization problem (Salvador, 1973). It was demonstrated that the two-stage HFSP with multiple identical parallel machines in each stage is an NP-Hard problem, even when the problem comprises only two stages, with one stage having only a single machine (Gupta, 1986). They proposed and validated an efficient heuristic algorithm to search for approximate solutions. A specialized encoding scheme for the HFSP was proposed, which combines local search and evolutionary search based on the differential evolution algorithm (Xu & Wang, 2011). They conducted experiments to validate that the proposed differential Evolution-based algorithm outperformed the contemporary genetic algorithms. An effective discrete artificial bee colony algorithm that combines forward and backward decoding methods was introduced (Pan, Wang, Li, & Duan, 2014). They developed a local optimization procedure for exploring new solution spaces. An enhanced artificial immune system (AIS) algorithm was proposed (Engin & Döyen, 2004). One notable feature of this algorithm, in comparison to genetic algorithms, is the use of mutation rates as adaptive parameters. Furthermore, they improved AIS algorithm parameters using a multi-step experimental design approach. A novel approach to solving the HFSP with lagrangian relaxation (LR) algorithms aimed at minimizing the total weighted delay was introduced (Nishi, Hiranaka, & Inuiguchi, 2010). A new method was devised that combines particle swarm optimization (PSO) with the bottleneck heuristic algorithm (C.-J. Liao, Tjandradjaja, & Chung, 2012). They utilized simulated annealing with PSO to escape local optima. An improved migration of birds optimization (MBO) algorithm was proposed, which incorporates various techniques such as diversified initialization methods, hybrid neighborhood structures, jump mechanisms, problem-specific heuristics, and local search processes (Pan & Dong, 2014). This enhanced MBO algorithm is applied to solve HFSP with the total flow time criterion. A hybrid evolutionary algorithm (HEA) was presented (Fan et al., 2021). The algorithm utilizes permutation-based encoding and employs two heuristic decoding methods to search the solution space. As the algorithm converges, a tabu search (TS) process is activated to expand the search space.

2.2 The literature review on HFSP-SDST

To the best of our knowledge, the earliest research on HFSP-SDST was published (Johnson, 1954). In this paper, the author examined a two-stage flow shop with setup times and introduced a heuristic method to address the minimization of makespan, which is considered an early significant contribution to the HFSP-SDST problem. A novel heuristic rule based on genetic algorithms was proposed (Ruiz & Maroto, 2006). Their algorithm tackled HFSP-SDST, each stage having unrelated parallel machines, and machine constraints. This approach was successfully applied to the tile manufacturing industry. An immune algorithm was proposed to solve HFSP-SDST, and it was experimentally validated that the proposed algorithm outperforms other algorithms (Zandieh, Ghomi, & Husseini, 2006). The HFSP-SDST with unrelated parallel machines was studied, and an improved genetic algorithm was proposed to minimize a convex combination of makespan and the number of tardy jobs (Jungwattanakit, Reodecha, Chaovalitwongse, & Werner, 2008). A hybrid simulated annealing algorithm was proposed to solve HFSP-SDST and transportation times (Naderi, Zandieh, Balagh, & Roshanaei, 2009). They demonstrated the effectiveness of the proposed algorithm through comparisons with other high-performance algorithms. A hybrid metaheuristic algorithm, the multiple objective genetic algorithm combined with variable neighborhood search (MOHM), was introduced for solving HFSP-SDST problems (Behnamian & Ghomi, 2011).An efficient algorithm consisting of independent parallel genetic algorithms was presented, enabling the population to search for the best solutions in different directions (Rashidi, Jahandar, & Zandieh, 2010).The two-stage HFSP-SDST problem was addressed, and an efficient heuristic algorithm that embeds the NEH process was proposed (Lee, Hong, & Choi, 2015). This approach rapidly generates solutions, which is beneficial for automated best-check machine operations in practical applications. The bi-objective HFSP-SDST problem, involving the minimization of total weighted tardiness and total setup time, was investigated (Tian, Li, & Liu, 2016). They introduced a pareto-based adaptive bi-objective variable neighborhood search that effectively resolved the problem under consideration. A population-based squirrel search algorithm (SSA) was introduced and applied to scheduling problems for the first time (Khare & Agrawal, 2019). Additionally, they introduced the whale optimization algorithm (WOA) and grey wolf optimization algorithm (GWO), which were also employed for the first time to address HFSP-SDST problems. A learning iterated greedy search metaheuristic was presented to minimize the maximum completion time in a hybrid flexible flow shop problem with sequence dependent setup times encountered at a manufacturing plant (Ozsoydan & Sağir, 2021).Problems with two types of SDST were explored: those dependent on job sequences and those dependent on job sequences and machine assignments (Y. Liao, Zhantao, Li, & Chenfeng, 2022). They proposed three heuristic algorithms and conducted comparative analyses. A metaheuristic algorithm based on genetic algorithms, along with three heuristic algorithms, was introduced (Jemmali & Hidri, 2023). They also provided three lower bounds based on relaxation methods to evaluate the efficiency of the algorithms.

3. Problem description

3.1 HFSP-SDST definition

The HFSP-SDST problem addressed in this paper can be described as follows: A set of n jobs *(j ∈J={1,...,j,...,n})* must traverse through s production stages (represented as *k ∈S={1,...,k,...,s}*) in a predefined production sequence. In other words, each job initially enters stage 1 for processing, subsequently moves to stage 2, and so forth, until it reaches the final stage *s*. At each stage, only one machine is available for job processing, making the selection of the right machine a critical decision. Moreover, the setting time $st_{k,j,j'}$ between two consecutive jobs processed on the same machine is defined as the time required for processing job *j* in stage *k* before job *j' (j, j' ∈J, j <j')*. In particular, if job *j* is the initial item processed on the machine at stage *k*, the setup time will be denoted as $st_{k,j,j}$. There are a total of *m* identical machines *(i ∈M={1,...,i,...,m})*, and each stage *k* is equipped with m_k ($\sum_{k=1}^{s} m_k = m$)identical parallel machines, and at least one stage must have a machine count of $m_k \geq 2$. Every job *j* can potentially be processed on any machine within stage *k*, and the processing time is represented as $p_{j,k}$, where *j ∈J*, *k ∈S*. It is essential to highlight that SDST is associated with the predecessor job on the same stage, indicating that different predecessor jobs will result in different setup times.

In practical applications, SDST can be employed to precisely define machine setup activities, such as changing machine tools or loading materials. The primary objective of this study is to enhance production efficiency by minimizing the makespan (denoted as C_{max}). It is essential to emphasize that at any given point in time, each machine can process only one job simultaneously, and the same job cannot be processed on multiple machines concurrently. In addition, the following assumptions are made:

1) All machines are simultaneously available and reliable, with no breakdowns.
2) The buffer or storage capacity between any two machines or any two stages is considered to be unlimited.
3) Transportation times between machines are disregarded.
4) Jobs are non-divisible, and processing in the current stage must be completed before moving to the next stage.
5) Setup times are assumed to be asymmetric, meaning that $st_{k,j,j'}$ and $st_{k,j',j}$ are different.

These assumptions lay the foundation for tackling the complexities of the HFSP-SDST problem, ensuring a comprehensive approach for optimization. This study aims to provide valuable insights for addressing similar complex manufacturing scheduling problems, opening new avenues for future research in this domain.

3.2 The MILP model

The parameters and decision variables used are shown as follows:

Parameters	
j	Indexes of jobs.
n	Total number of jobs.
i	Indexes of machines.
m	Total number of machines.
k	Indexes of stages.
s	Total number of stages.
S	Set of stages and $S = \{1, ..., k, ..., s\}$.
J	Set of jobs and $J = \{1, ..., j, ..., n\}$.
M	Set of machines and $M = \{1, ..., i, ..., m\}$.
L	A large positive number.
Decision variables	
$p_{j,k}$	The processing time of job *j* in stage *k*.
m_k	The number of machines in stage *k*.
$st_{k,j,j'}$	The setup time for processing job *j* before job *j'* on stage *k*.
$X_{j,k,i}$	A binary decision variable, which equals 1 if job *j* is scheduled to be processed on machine *i* in stage *k*; 0 otherwise
$Y_{j,j',k}$	A binary decision variable, which equals 1 if job *j* is processed directly or indirectly before job *j'* in stage *k*; 0 otherwise.
$B_{j,k}$	The completion time of job *j* at stage *k*.
$E_{j,k}$	The start time of job *j* at stage *k*.
C_{max}	The makespan.

$$min\ C_{max} \tag{1}$$

$$\sum_{i \in m_k} X_{j,k,i} = 1, \forall j \in J, k \in S \tag{2}$$
$$E_{j,k} = B_{j,k} + \sum_{i \in m_k} (p_{j,k} X_{j,k,i}), \forall j \in J, k \in S \tag{3}$$
$$E_{j,k} \leq B_{j,k+1}, \forall j \in J, k \in \{1, \ldots, S-1\} \tag{4}$$
$$E_{j,k} + st_{k,j,j'} \leq B_{j',k} + L(3 - Y_{j,j',k} - X_{j,k,i} - X_{j',k,i}), \forall j \in J, j' \in J, j < j', i \in m_k \tag{5}$$
$$E_{j',k} + st_{k,j',j} \leq B_{j,k} + L(2 + Y_{j',j,k} - X_{j,k,i} - X_{j',k,i}), \forall j \in J, j' \in J, j < j', i \in m_k \tag{6}$$
$$C_{max} \geq E_{j,s}, \forall j \in J \tag{7}$$
$$B_{j,k} \geq 0, \forall j \in J, k \in S \tag{8}$$
$$X_{j,k,i} \in \{0,1\}, \forall j \in J, \forall k \in S, \forall i \in m_k \tag{9}$$
$$Y_{j,j',k} \in \{0,1\}, \forall j \in J, \forall j' \in J, j < j', k \in S \tag{10}$$

where, the objective function (1) is to minimize the makespan. Constraint (2) states that each job can only be processed on exactly one machine in any stage. Constraint (3) represents the relationship between the start and end times of a job. Specifically, the end time is equal to the start time plus the processing time. Constraint (4) ensures that the end time of any predecessor operation for a job is not greater than the start time of its successor operation. Pairwise constraints (5)-(6) are used to guarantee the precedence relationship between two different jobs on the same machine, ensuring that the start time of the later job is not less than the completion time of the earlier job plus the setup time between the two jobs. Constraint (7) is the maximum makespan constraint. Constraints (8)-(10) define the feasible range for decision variables.

4. The IABC algorithm

4.1 Basic ABC

The ABC is a population-based swarm intelligence optimization algorithm (Karaboga, 2005; Karaboga & Basturk, 2008), simulating the process of honeybees searching for food. In the basic ABC, it consists of three main components: food sources, employed bees, unemployed bees, and two fundamental behaviors: recruiting for a food source and abandoning a food source. Each food source represents a potential solution to an optimization problem, and the amount of nectar of a food source corresponds to the fitness of a solution. Employed bees represent candidate solutions in the search space, and they collect information about food sources. Their main task is to discover food sources, store relevant information, and share this information with unemployed bees in the beehive. Unemployed bees include onlooker bees and scout bees. After employed bees return information about food sources, follower bees choose better-quality food sources to explore further based on certain probabilities. If certain food sources remain unchanged for a certain number of consecutive iterations, they are abandoned, and the employed bees corresponding to those food sources become scout bees, which randomly search for new food sources. The ABC has been successfully applied in various fields, including optimization problems, data mining, machine learning, power systems, image processing, and more. It is widely regarded as a powerful optimization tool capable of effectively finding the best solutions to problems. The parameters in the ABC algorithm include population size (*PSize*), the number of iterations (*L*) after which solutions that have not been updated are discarded, and the termination criteria (*LimitTime*).

4.2 IABC algorithm

In this section, the IABC algorithm is described form seven parts, namely encoding and decoding, initialization of the population, employed bee phase, onlooker bee phase, scout bee phase, problem-specific local search and the framework.

4.2.1 Encoding and decoding

HFSP typically has two main encoding methods: matrix encoding and permutation encoding. Matrix encoding enumerates the machines used by each job in each stage and the order in which jobs are processed, providing detailed information. However, due to the excessive detail, this representation is cumbersome and challenging to operate and analyze effectively. In comparison, permutation encoding is more concise and easier to understand. Therefore, in the study of HFSP-SDST, permutation encoding was chosen as the encoding method used in this paper. Permutation encoding is an operation-based encoding method that involves randomly permuting the serial numbers of all jobs to form an individual. The length of everyone is equal to the number of jobs to be processed, and the position of each job in the individual represents its processing sequence in the first stage. The simplicity and ease of operation of this encoding method make it easier to handle and optimize the solution process of HFSP and better meet the needs of HFSP to directly reflect job sequencing and machine selection.

Conventional flow shop scheduling problems satisfy the reversibility of processing. In other words, without changing the assignment of jobs to machines, applying the backward process to a forward schedule, starting from the last stage and moving forward to the first stage, does not alter the maximum completion time. However, HFS lacks reversibility, and using forward decoding and backward decoding can often yield different solutions that cannot be obtained using just one of the methods. Therefore, to increase the diversity of the population and obtain better solutions, this paper adopts a hybrid decoding method that considers SDST and randomly selects between forward and backward decoding. In other words, everyone generates a flag to record the decoding method it uses. A flag value of 0 indicates forward decoding, while a flag value of 1 indicates backward decoding. The hybrid decoding approach proposed in this section is evaluated in Section 5.3.2. The specific decoding process includes the following two rules:

(1) Rules for job scheduling: The decoding method is chosen with equal probability based on the flag. The job sequence represents the order in which jobs enter the shop in the first stage. In the subsequent stages, based on the values of the jobs from the previous stage as given by formula (12), their start times for the next stage are determined. Forward decoding typically starts by scheduling the first stage, then schedules the next stage based on the completion times of jobs in the first stage, and so on until the final stage. In contrast to forward decoding, backward decoding begins by processing the last stage first, sequentially processing jobs according to the initial job sequence. Jobs from previous stages are allocated to machines that finish first in the next stage, based on the order of completion times.

(2) Rules for machine selection: When proceeding to the stage *k (k=1, 2, ..., m)*, taking forward decoding as an example, in the first stage, the initial sequence generated for the first stage is used to select the machine that will process the job to be completed earliest, rather than choosing the machine that becomes available first. From the second stage onward, based on the completion time of job *j* in stage *k-1*, the processing time in stage *k*, and the sequence of jobs processed on the machine, the earliest completion time for each job on machine *i* is determined. Specifically, it is calculated as the maximum between the sum of the machine's idle time R_m and setup time $st_{k,j',j}$ and the completion time $E_{i,k-1}$ from the previous stage, represented by formula (11). Then, the job *j* is assigned to the machine *i* with the minimum value as per formula (12) for job processing. Repeat the above steps until all jobs have completed processing in *s* stages.

$$\max\left(R_m + t_{m,j'j}, E_{j,k-1}\right) \tag{11}$$
$$\max\left(R_m + t_{m,j'j}, E_{j,k-1}\right) + P_{j,m} \tag{12}$$

For example, let's consider a problem with 4 jobs and 2 stages. The first stage involves 3 machines, while the second stage has 2 machines. The initial processing sequence Π_l = (1, 2, 3, 4). The processing times and setup times for the jobs are as follows:

$$[p_{k,j}] = \begin{Bmatrix} 3\ 2\ 3\ 4 \\ 3\ 1\ 2\ 4 \end{Bmatrix} \quad [st_{1,j,j'}]_{4*4} = \begin{Bmatrix} 1\ 2\ 2\ 1 \\ 1\ 1\ 2\ 3 \\ 3\ 1\ 2\ 2 \\ 2\ 2\ 3\ 2 \end{Bmatrix} \quad [st_{2,j,j'}]_{4*4} = \begin{Bmatrix} 2\ 1\ 2\ 3 \\ 3\ 2\ 1\ 2 \\ 1\ 2\ 1\ 2 \\ 2\ 1\ 3\ 2 \end{Bmatrix}$$

The specific backward scheduling process for the above example is as follows:

(1) First is the second stage. The processing is carried out in the order of the jobs, which are 4, 3, 2, and 1. Job 4 is assigned

to machine 4.
(2) Based on the completion times of the second stage jobs, the scheduling for the first stage is arranged. The release times for jobs 4, 3, 2, and 1 are 4, 2, 4, and 8, respectively. Based on the values of the formula (12), schedule job 3 to machine 1 first. Job 4 is assigned to machine 2, job 2 is assigned to machine 3, and job 1 is assigned to machine 1. Since this instance assumes only two stages, the scheduling process concludes here, resulting in the backward scheduling Gantt chart as shown in Fig. 2. Fig. 1 is the Gantt chart for the forward scheduling of this instance. It can be observed that the makespan obtained from the forward scheduling is 13, while for the backward scheduling, it is 12. This further demonstrates that the proposed backward decoding in this paper indeed leads to solutions that cannot be achieved by a single forward decoding, validating its effectiveness.

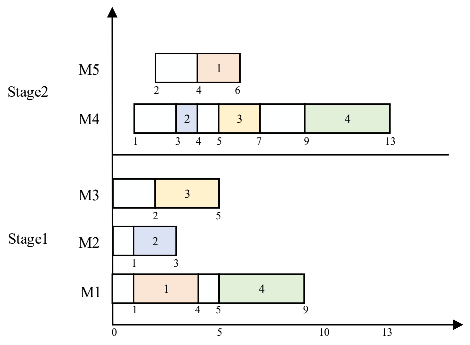
Fig. 1. The forward scheduling Gantt chart for sequence Π_l

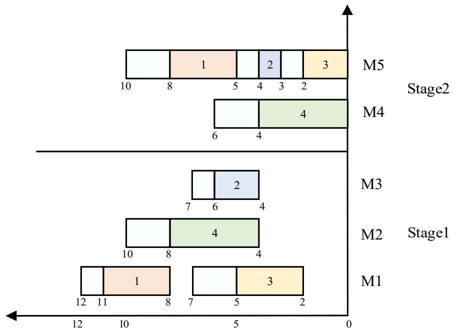
Fig. 2. The backward scheduling Gantt chart for sequence Π_l

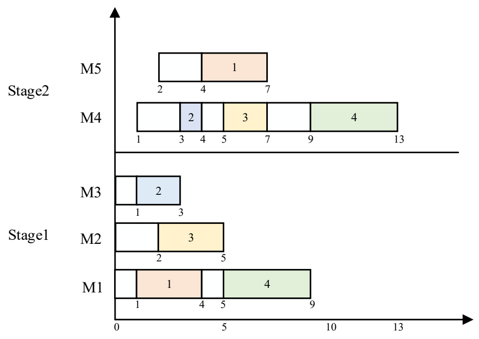
Fig. 3. The forward scheduling Gantt chart for sequence Π_l'

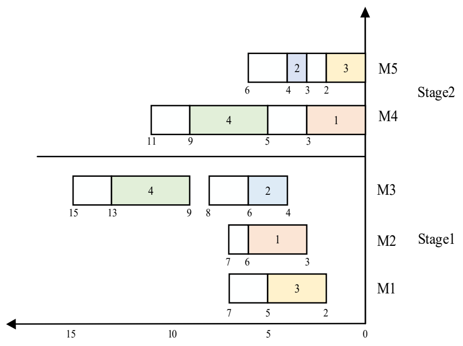
Fig. 4. The backward scheduling Gantt chart for sequence Π_l'

When the initial sequence of the above example is Π_l= (1, 3, 2, 4), a different makespan is obtained through the same processing path compared to the previous one. The specific process Gantt charts are shown in Fig. 3 and Fig. 4. It can be observed that the makespan is different for forward decoding and backward decoding, with a makespan of 13 for forward decoding and 15 for backward decoding. Forward decoding is superior to backward decoding. In summary, this further demonstrates that the HFSP-SDST is not reversible. Using only forward decoding often cannot achieve the solution obtained by using only backward decoding, and vice versa. Considering the strengths and limitations of the two decoding methods, a hybrid decoding approach is proposed. As the name implies, hybrid decoding selects one of the two decoding methods with equal probability as the decoding method for the current individual. All individuals use a flag bit (0 or 1) to determine which decoding method to employ.

4.2.2 Initialization of the population

The IABC algorithm typically employs a random approach to generate *PSize* individuals in the initial population, denoted by $\Pi = \{\Pi_1, \ldots, \Pi_l, \ldots, \Pi_{PSize}\}$, where $\Pi_l = (\pi_{l,1}, \ldots, \pi_{l,j}, \ldots, \pi_{l,n})$ and Π_l represents the l individual in the population, and $\pi_{l,j}$ denotes the j job within the l individual. Flag bits are also randomly generated as 0 or 1. In this paper, random initialization is adopted to create the population, ensuring the diversity of the initial population.

4.2.3 Employed bee phase

In this stage, scout bees need to venture out in the vicinity of their current location to search for food sources. Two neighborhood structures, namely insert and pairwise exchange are used, which have proved to be very effective (Pan et al., 2014). An insertion involves removing the job at position p within an individual and then reinserting it at another position p'. A pairwise exchange operation consists of r swapping the jobs at positions v and w within the sequence. The specific neighborhood structures are illustrated in Fig. 5 and Fig. 6. During the scout bee stage, the following process is implemented: A control parameter, α (representing the number of iterations in the employed bee stage), is introduced. Everyone will undergo α insert neighborhood operations, updating the solution if it is improved compared to the original solution. Then, they undergo α pairwise exchange operations. Finally, if the neighboring solution is better than the current best solution, then update the current best solution. It's important to note that the individual's decoding method remains unchanged throughout this process (Pan et al., 2014). The detailed process of the employed bee phase is as shown in Algorithm 1.

```
Algorithm1 Employed Bee Phase
Input: a set of solution $\Pi_l = (\pi_{l,1}, \pi_{l,2}, \ldots, \pi_{l,n})$,
       a parameter α;
Output: a set of improved solution $\Pi_l = (\pi_{l,1}, \pi_{l,2}, \ldots, \pi_{l,n})$;
1:  for j = 0 to 1
2:    Cnt = j;
3:    for rep = 1 to α
4:      for i = 1 to PSize;
5:        if Cnt = 0 then
6:          $\Pi_l^* \leftarrow insert(\Pi_l)$;
7:        else
8:          $\Pi_l^* \leftarrow$ pairwise exchange $(\Pi_l)$;
9:        endif
10:        if $C_{max}(\Pi_i^*) < C_{max}(\Pi_l)$ then
11:          $\Pi_l = \Pi_l^*$;
12:        endif
13:      endfor
14:      j ++;
15:   endfor
16: Find the best solution $\Pi_{best}$;
17: UpdateBestSolution $(\Pi_{best})$;
```

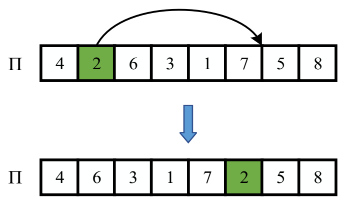

Fig. 5. Insert

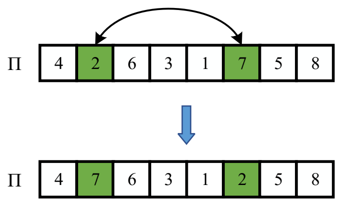

Fig. 6. Pairwise exchange

4.2.4 Onlooker bee phase

In the basic ABC, the onlooker bees select higher-quality food sources in the neighborhood for further exploration with a certain probability. If the neighborhood solution is better than the current solution, it replaces the current solution. In this paper, a tournament selection strategy is used for choice, which differs from the strategy used in the basic ABC (Pan et al., 2014). The tournament selection strategy is known for its simplicity and its ability to escape local optima, and it is widely used in various algorithms. In this strategy, two individuals are randomly chosen to compete, and the winner is selected from the two. After the selection, the onlooker bees phase uses the same method as the employed bee phase to generate a new neighborhood solution. If the solution obtained after the search is better than the worst individual in the population with the same decoding method and no other identical individuals, the neighborhood solution replaces the worst solution. If the neighboring solution is better than the current best solution, then update the current best solution. The detailed process of the onlooker bee phase is as shown in **Algorithm 2**.

```
Algorithm2 Onlooker Bee Phase
Input: a population $\Pi$;
Output: an improved solution $\Pi$;
1:  for i = 1 to PSize
2:    Randomly select two different solutions $\Pi_{pt1}$ and $\Pi_{pt2}$;
3:    $\Pi_{pt} \leftarrow BinaryTournament(\Pi_{pt1}, \Pi_{pt2})$;
4:    $\Pi_i^* \leftarrow insert(\Pi_i)$;
5:    Find the worst individual $\Pi_{worst}$ with the same flag bit as $\Pi_i^*$;
6:    if $C_{max}(\Pi_i^*) < C_{max}(\Pi_{worst})$ then
7:      $\Pi_{worst} = \Pi_i^*$;
8:    endif
9:  endfor
10: Find the best solution $\Pi_{best}$;
11: UpdateBestSolution $(\Pi_{best})$;
```

4.2.5 Scout bee phase

In the basic ABC, if certain food sources have not been updated within a specified number of iterations, scout bees generate new food sources randomly in the solution space. However, this approach often leads to solutions that lack clear problem-specific characteristics, causing the algorithm to become disoriented in the search space and making it difficult to find high-quality solutions efficiently. Additionally, it may generate many unrelated or ineffective solutions during the search, reducing the algorithm's efficiency. In this paper, an enhanced strategy is employed to generate new individuals (Pan et al., 2014). Instead of conducting random searches around solutions that haven't been updated within the specified iteration limit, the algorithm avoids exhaustive searching, thus improving the algorithm's guidance and efficiency. A control parameter φ is introduced to prevent the algorithm from getting stuck in local optima. If the iteration count of an individual is greater than φ, then discard that individual. The discarded individuals undergo multiple random insertion operations to generate several candidate individuals. The best solution among all candidate individuals is selected as the new solution for the scout bee. If the neighboring solution is better than the current best solution, then update the current best solution. The detailed process of the scout bee phase is as shown in Algorithm 3.

Algorithm3 Scout Bee Phase

```
Input: a set of non updated solution $\Pi_l = (\pi_{l,1}, \pi_{l,2}, \dots, \pi_{l,n})$,
    a parameter $L$ and $\varphi$;
Output: a new set of solution $\Pi_l = (\pi_{l,1}, \pi_{l,2}, \dots, \pi_{l,n})$;
1:  for l = 1 to PSize
2:    if $L(\Pi_l) > L$ then
3:      for j = 1 to $\varphi$
4:        $\Pi_{temp} = \Pi_l$;
5:        $\Pi_{temp}* \leftarrow insert(\Pi_{temp})$;
6:        $\Pi_{temp}** \leftarrow insert(\Pi_{temp}*)$;
7:        if $C_{max}(\Pi_{temp}**) < C_{max}(\Pi_l)$ then
8:          $\Pi_l = \Pi_{temp}**$;
9:        endif
10:     endfor
11:   endif
12: endfor
2.  13: Find the best solution $\boldsymbol{\Pi_{best}}$;
14: UpdateBestSolution ($\Pi_{best}$);
```

4.2.6 Problem-specific local search

During the encoding and decoding processes, there are instances where the completion times of multiple jobs are identical. In such situations, a job is randomly selected for processing as described in Section 4.2.1. However, this random selection method may inadvertently reduce the diversity of potential solutions, limiting the search space. To address this concern, LS that considers SDST has been incorporated. To illustrate this, let's take the forward decoding process as an example. When multiple jobs complete processing simultaneously in the preceding stage, a random selection of a single job is not opted for. Instead, each pending job is individually evaluated to calculate its makespan. Subsequently, the job with the smallest makespan is given priority for processing. This strategy is consistently applied in subsequent stages. Conversely, the backward decoding process follows a similar procedure but in the opposite direction. Specifically, in each iteration, the LS is only conducted on the best solution of the population. Moreover, the effectiveness of the LS is evaluated in Section 5.3.1.

The steps of the problem-specific local search are as follows:
Step 1: Set $k = 2$.
Step 2: Set $q = 1$ and generate the processing sequence $\Pi_l = (\pi_1, \pi_2, \dots, \pi_n)$ for this stage based on the completion times from the k-1 stage.
Step 3: Use formula (12) to select machine i^* for processing job π_q and identify all the jobs that have been completed in the previous stage after π_q; add them to the set $A = (\pi_{q+1}, \pi_{q+2}, \dots, \pi_{q'})$. If A is empty, proceed to Step 5; otherwise, continue with Step 4.
Step 4: Swap job π_q with the jobs in A to generate a new processing sequence Π^*. Calculate the completion time of sequence Π^*. Repeat Step 3 until all jobs in set A have been considered.
Step 5: Increment $q + 1$. If $q < n$, return to Step 3. Otherwise, proceed to Step 6.
Step 6: Increment $k + 1$. If $k < s$, return to Step 2. Otherwise, evaluate all the generated solutions and select the best solution Π_{best}.

An instance with 2 stages and 5 jobs is considered, with each stage equipped with 2 identical parallel machines. The processing times $p_{k,j}$ for the jobs, and the setup times $[st_{k,j,j'}]_{5*5}$ for the jobs in each stage are as follows:

$$p_{k,j} = \begin{Bmatrix} 2 & 1 & 6 & 1 & 1 \\ 5 & 5 & 1 & 2 & 6 \end{Bmatrix} \qquad [st_{1,j,j'}]_{5*5} = \begin{Bmatrix} 1 & 2 & 2 & 1 & 1 \\ 2 & 1 & 1 & 1 & 2 \\ 1 & 1 & 2 & 2 & 3 \\ 2 & 3 & 2 & 1 & 1 \\ 1 & 2 & 3 & 1 & 2 \end{Bmatrix} \qquad [st_{2,j,j'}]_{5*5} = \begin{Bmatrix} 1 & 1 & 2 & 1 & 3 \\ 2 & 1 & 2 & 1 & 1 \\ 2 & 2 & 1 & 2 & 1 \\ 1 & 1 & 1 & 2 & 1 \\ 1 & 2 & 1 & 2 & 3 \end{Bmatrix}$$

With the initial job processing sequence π = (1, 2, 3, 4, 5) and the flag bit set to 0, forward decoding is used to generate a complete solution. In the first stage, jobs 1, 4, and 5 are assigned to machine 1, while jobs 2 and 3 are processed on machine 2, following the given initial sequence. In the second stage, jobs are sorted based on their release times from the first stage, resulting in the order (2, 1, 4, 5, 3). The processing sequence on machine 1 becomes (2, 4, 5), and on machine 2, it becomes (1, 3). The specific scheduling Gantt chart is shown in the Fig.7 and Fig. 8 below:

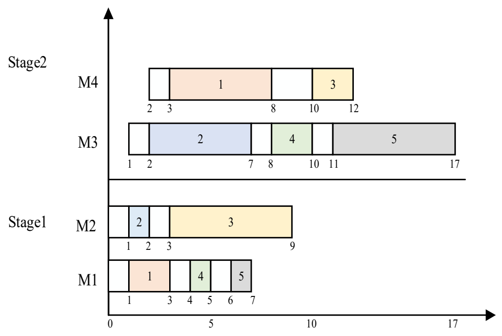

Fig. 7. An example of the forward schedule

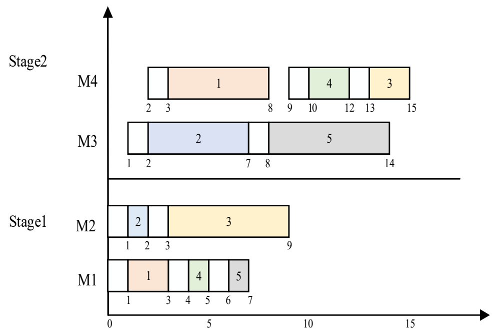

Fig. 8. A neighboring solution of **Fig.7**

4.2.7 The framework of IABC

Based on the design, Fig. 9 provides the overall framework of the IABC algorithm. The detailed IABC steps are as follows:

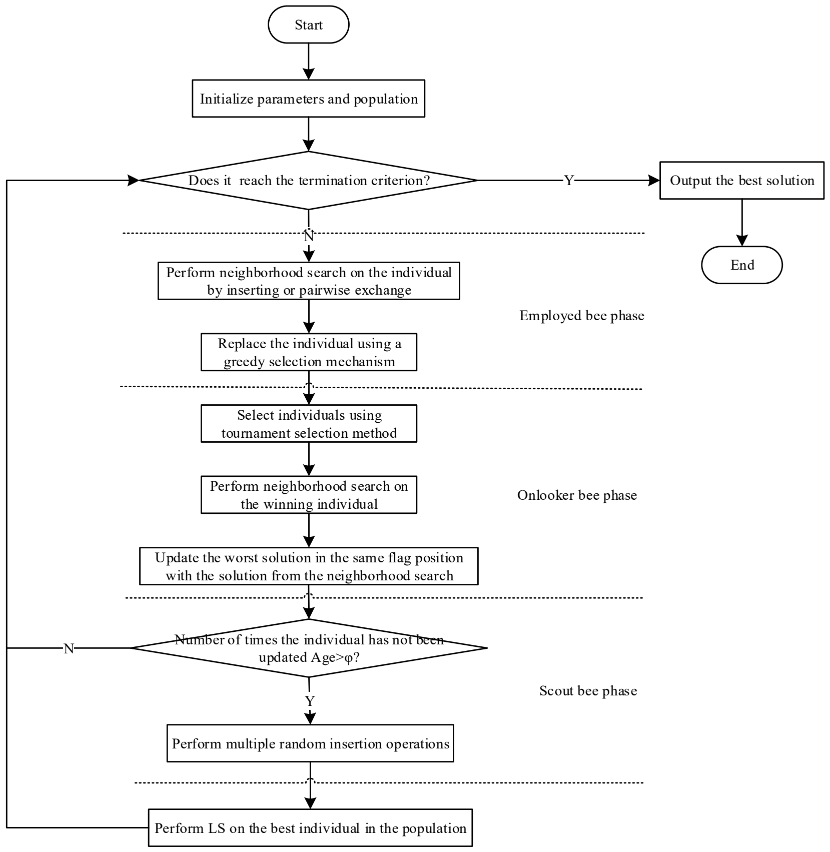
Start
Initialize parameters and population
Does it reach the termination criterion?
Y
Output the best solution
End
N
Perform neighborhood search on the individual by inserting or pairwise exchange
Replace the individual using a greedy selection mechanism
Employed bee phase
Select individuals using tournament selection method
Perform neighborhood search on the winning individual
Onlooker bee phase
Update the worst solution in the same flag position with the solution from the neighborhood search
Number of times the individual has not been updated Age>φ?
N
Y
Scout bee phase
Perform multiple random insertion operations
Perform LS on the best individual in the population

Fig. 9. Framework of IABC

Step 1: Initialize parameters and population. Set the population size ($PSize$), the number of iterations in the employed bee stage (α), the number of iterations a solution remains unchanged for an individual (L), and the number of neighborhood solutions generated by discarded individuals in the scout bee stage (φ). The population is initialized as $\{\Pi_1, \dots, \Pi_l, \dots, \Pi_{PSize}\}$ using the method from Section 4.2.2.

Step 2: Employed bee phase. For l = 1, 2, ..., $PSize$, the individual Π_l undergoes insert and pairwise exchange operations to generate a new individual Π_l*. If Π_l*is better than Π_l, let Π_l=Π_l*. Repeat this process until the employed bee stage is executed α times.

Step 3: Onlooker phase. For l=1,2, …, $PSize$, randomly select two individuals and use tournament selection to choose the better one. Generate a new individual for this selected individual using an insertion operation. If there is no identical individual in the population to the new individual, and the new individual is better than the worst individual in the population with the same flag, then replace that individual with the new one.

Step 4: Onlooker phase. For l=1,2, …, $PSize$, if the individual Π_l has not been updated in L consecutive iterations, discard that individual. Perform multiple insertion operations on the discarded individual to generate φ neighboring solutions and place the best neighboring individual back into the population.

Step 5: Perform LS on the best individual as described in Section 4.2.6.

Step 6: If the termination criterion is reached, output the best solution; otherwise, go to Step 3.

5. Calibration and Evaluation

This section aims to comprehensively evaluate the effectiveness of the MILP model and IABC algorithm through systematic experimental research. The instances used include two different scales. Large-scale instances are used to test the IABC algorithm, and the data are sourced from the test instances used by reference (Gómez-Gasquet, Andrés, & Lario, 2012). Unrelated parallel machines were used during the production phase. Unrelated parallel machines refer to machines that may exhibit variations in certain aspects, resulting in different job processing times on each machine. In this instance, the number of jobs, n∈{20, 50}, and the number of machines, m, is distributed in two scenarios: P13 (randomly distributed between one and three machines) and P3 (a constant number of three machines). The number of stages, s∈{5, 10, 20}. Processing times are uniformly distributed in the range [1,99]. Four variations of SDST, denoted as SSD10, SSD50, SSD100, and SSD125, correspond to 10%, 50%, 100%, and 125% of the job processing times. The DABC algorithm proposed by reference (Pan et al., 2017) in 2017 also used 240 benchmark examples from reference (Gómez-Gasquet et al., 2012) to solve related problems. In subsequent validation and assessment, only one load level (20 jobs) is considered as the data source to test the proposed algorithm, following the same generation process. Large-scale instances are used to test the MILP model, also sourced from the subset of the benchmark database. The number of jobs (n) is chosen from {10, 20}, the number of stages (s) from {5, 10}, and each stage has three machines with setup times set to 10% of the processing time. The MILP model is implemented in OPL language and solved using CPLEX, while the ABC algorithm is implemented in C++. Both are executed on a PC with an Intel i7-12700 CPU @ 2.1 GHz, Windows 10 operating system, and 16GB of memory.

5.1 MILP model validation

To validate the effectiveness of the MILP model, this section conducts verification using a subset of 20 instances from the benchmark library mentioned above, specifically from SSD_P3.

Table 1
Results of MILP model for minimizing makespan consumption

Instances	NBV	NIV	NC	Makespan	Time(s)	Gap (%)
ta002_10_5	825	101	1550	262	84.4	0
ta004_10_5	825	101	1550	268	37.8	0
ta006_10_5	825	101	1550	265	16.1	0
ta008_10_5	825	101	1550	220	6.6	0
ta010_10_5	825	101	1550	227	3.2	0
ta012_10_5	825	101	1550	268	24.0	0
ta014_10_5	825	101	1550	202	4.8	0
ta016_10_5	825	101	1550	195	20.6	0
ta018_10_5	825	101	1550	227	13.1	0
ta020_10_5	825	101	1550	263	99.7	0
ta012_10_10	1650	201	3100	422	600.1	6.2
ta014_10_10	1650	201	3100	256	5.1	0
ta016_10_10	1650	201	3100	362	600.0	1.4
ta018_10_10	1650	201	3100	366	600.0	0.5
ta020_10_10	1650	201	3100	459	600.0	4.6
ta002_20_5	3150	201	6100	362	600.0	31.8
ta004_20_5	3150	201	6100	381	604.1	40.2
ta006_20_5	3150	201	6100	370	600.1	28.6
ta008_20_5	3150	201	6100	369	600.3	43.4
ta010_20_5	3150	201	6100	335	600.1	35.2
ta012_20_10	6300	401	12200	565	600.1	29.9
ta014_20_10	6300	401	12200	458	600.1	22.9
ta016_20_10	6300	401	12200	548	600.1	37.0
ta018_20_10	6300	401	12200	544	600.2	33.1
ta020_20_10	6300	401	12200	587	600.1	28.4

Each instance consists of 10 stages, considering both 10 jobs and 20 jobs scenarios. The MILP model was implemented in the OPL language of the IBM CPLEX Studio IDE 12.7.1. The CPLEX solver used a branch-and-cut method combining the cutting plane and branch-and-bound methods (Meng et al., 2024). The maximum CPU runtime is set to 600 seconds, and the program operates in the same environment as mentioned above. Table 1 provides the results of running the model. In the table, NBV represents the number of binary decision variables, NIV represents the number of integer decision variables, NC represents the number of constraints, Times(s) represents CPU runtime, and Gap indicates the deviation from the best solution. When the Gap value is 0, the solution is optimal. It can be observed that the MILP model can only obtain the best solutions for small-scale instances, such as ta002_10_5- ta020_10_5, and ta014_10_10, within a reasonable time frame. However, as the problem size increases, the performance of MILP deteriorates, making it difficult to find the best solution within the given time constraints.

5.2 Parameters calibration of IABC

In this section, the Taguchi experimental design method (DOE) was employed to determine the values of four key parameters in the IABC algorithm. These four parameters are the population size (*PSize*), the maximum limit of the bee stages (α), the maximum number of consecutive update failures (L), and the number of neighborhood solutions generated in the scout bee stage (φ). At the beginning of the experiments, several typical values were selected for each parameter, as shown in **Table 2**. Based on four parameters with four factor levels, an orthogonal matrix L16 was selected, generating 16 different parameter combinations. For each parameter combination, instances with different problem sizes were randomly selected from the benchmark library. The instances included 9 different problem size combinations with the number of jobs $n \in \{20,50,100\}$ and the number of stages $s \in \{5, 10, 20\}$. For each problem size combination, 8 instances were randomly selected, covering 4 different setup time sizes and 2 different machine distribution scenarios mentioned earlier, resulting in a total of 8×9 = 72 instances. Each instance was independently run 10 times, with a maximum CPU time limit of 50×n×s milliseconds. Thus, each parameter combination was run independently 8×10×9 times, and its Relative Percentage Increase (*RPI*) was calculated. Finally, the average of the obtained *RPI* values (*ARPI*) was taken as the response result. The *ARPI* for 720 runs of different parameter combinations is presented in Table 3.

$$RPI(c) = (c - c_{\min}) / c_{\min} \times 100\% \tag{13}$$

where c is the objective value of the same instance with the same problem scale in the current parameter combination, which $c_{\min}$ is the smallest objective value of the same instance with the same problem scale among all parameter combinations.

Fig. 9 displays the factor level trend. The smaller value of the *ARPI*, the better the performance. It is evident that *PSize* = 10,20,30 has a smaller *ARPI* compared to *PSize* = 5; there is no significant difference in the impact of the number of cycles in the employee bee phase on the results; for the reconnaissance bee phase, the neighborhood solution quantity φ = 20, 30, 40 performs better than *PSize* = 5, demonstrating the effectiveness of the reconnaissance bee phase used. Finally, based on the above analysis, the parameters are set as follows: *PSize*=20; α=10; L=90; φ=30.

Table 2
Four sets of parameter values

Parameter	Parameter level			
	1	2	3	4
PSize	20	30	40	50
α	5	10	20	30
L	30	50	70	100
φ	5	20	30	40

Table 3
The ARPI values for different parameter combinations

PSize	α	L	φ	ARPI
5	5	30	5	13.5249
5	10	50	20	13.0132
5	20	70	30	12.5658
5	30	90	40	12.3250
10	5	50	30	10.8631
10	10	30	40	11.5896
10	20	90	5	11.8129
10	30	70	20	11.4534
20	5	70	40	11.1926
20	10	90	30	10.8382
20	20	30	20	11.1537
20	30	50	5	11.6913
30	5	90	20	11.3323
30	10	70	5	11.4203
30	20	50	40	11.4751
30	30	30	30	11.5497

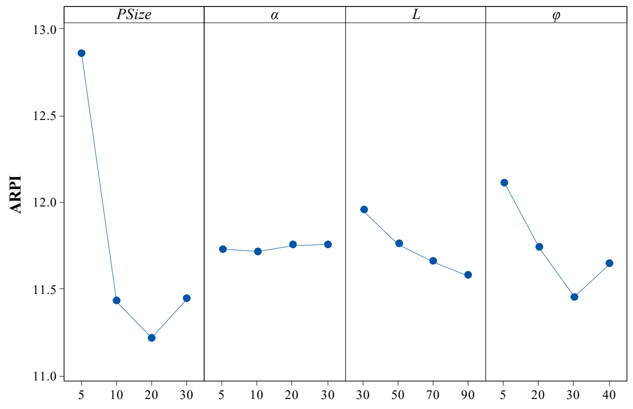

Fig. 9. The trend of the factor level

5.3 *Evaluation of IABC*

5.3.1 The evaluation of LS

To substantiate the effectiveness of the proposed LS strategy, verifications were conducted from both forward decoding and backward decoding perspectives. 24 instances were selected from different combinations of P3, P13, SSD10, SSD50, SSD100, and SSD125 among the 240 benchmark instances, representing various scales. For each instance where n=20 and $s\in\{5, 10, 20\}$, 10 executions were independently run, comparing the minimum makespan values from each run. The maximum CPU time limit for each run was set at $300\times n\times s$ milliseconds. The computational results are presented in Table 3 and Table 4 (numerical values are represented as before LS/after LS). Values in bold indicate that the makespan value improved after the LS. From the tables, it is evident that among the randomly selected 24 test instances, forward decoding further improved 11 instances beyond the original best solution, while 13 remained the same as before the LS. In the case of backward decoding, it further improved 13 instances, with 11 instances remaining unchanged from before the LS. It is evident from this data that the solutions after LS are either less than or equal to the original solutions. Therefore, the proposed LS strategy considering SDST enhances the current best-known solutions and is undeniably effective.

Table 4
Instances are based on LS with forward decoding

Instances	P13				P3			
(LS before/after)	SSD10	SSD50	SSD100	SSD125	SSD10	SSD50	SSD100	SSD125
ta006	1320/1320	**1417/1414**	1570/1570	**1691/1689**	344/344	436/436	550/550	588/588
ta016	1354/1354	**1549/1537**	1836/1836	2020/2020	**508/507**	**636/634**	778/778	**842/840**
ta026	**1841/1829**	**2124/2116**	2488/2488	2698/2698	818/818	**994/991**	**1180/1176**	**1252/1245**

Table 5
Instances are based on LS with backward decoding

Instances	P13				P3			
(LS before/after)	SSD10	SSD50	SSD100	SSD125	SSD10	SSD50	SSD100	SSD125
ta006	1314/1314	1387/1387	1530/1530	1618/1618	341/341	**438/437**	**522/520**	588/588
ta016	**1349/1347**	1536/1536	**1894/1882**	**2053/2048**	506/506	627/627	**774/768**	847/847
ta026	**1836/1835**	2126/2126	**2526/2499**	**2716/2700**	**822/813**	**981/979**	**1168/1163**	**1260/1251**

5.3.2 The evaluation of decoding method

In this paper, to better address the HFSP-SDST and enhance the exploration capability of the solution space, a backward decoding is introduced based on the original forward decoding. The feasibility of the backward decoding is demonstrated through an example in Section 4.2.

Table 5
Comparison of three decoding methods

Instances		SSD10	SSD50	SSD100	SSD125
Forward/ Backward/ Hybrid					
P13	ta006	1320/**1314/1314**	1417/**1387/1387**	1570/**1530/1530**	1691/1618/**1615**
	ta016	1354/**1349/1349**	1569/**1536**/1537	**1836**/1894/**1836**	**2020**/2053/**2020**
	ta026	1841/**1836/1836**	2124/**2128/2116**	**2488**/2526/**2488**	2698/2716/**2693**
P3	ta006	344/341/**340**	436/439/**434**	550/**522/522**	588/588/**575**
	ta016	508/506/**505**	636/**627**/631	778/774/**767**	842/847/**838**
	ta026	**818**/822/822	994/**981/981**	1180/1168/**1166**	1252/1260/**1244**

For convenience, verifications were conducted using the instances selected in the previous section. Similarly, each instance was independently executed 10 times, with a maximum CPU time limit of 300×*n*×*s* milliseconds, and the minimum makespan value was selected from each run. The results are presented in Table 5. Values in bold are the best among all the three decoding methods. It can be observed that forward decoding achieved the best solution in only 4 instances, backward decoding in 9 instances, while hybrid decoding achieved the best solution in 21 instances. The experiments demonstrate that the use of hybrid decoding significantly expands the solution space and outperforms using either forward or backward decoding alone. Therefore, the proposed hybrid decoding method holds considerable research value.

5.4 Comparison of the IABC with the state-of-art algorithms

In this section, 120 benchmark instances are used to compare the proposed IABC with DABC(Pan et al., 2017) and GA(Gómez-Gasquet et al., 2012). The same control parameters for the algorithm as those in Section 5.2 are employed, and the algorithm is run with a maximum elapsed CPU time limit of 300×n×s milliseconds. Table 6 displays the best-known values corresponding to the instances used in the experimental phase of this study. In 120 benchmark instances, the proposed IABC algorithm improved the known the best makespan for 61 instances. Among them, values in bold represent the best makespan achieved by IABC, and the other values indicate the currently known best makespan. Moreover, the improved solutions for 61 instances are given in Appendix.

Table 6
Best-known solutions for the benchmark instances

Instances	P13				P3			
	SSD10	SSD50	SSD100	SSD125	SSD10	SSD50	SSD100	SSD125
ta002	1025	**1185**	**1374**	**1463**	**346**	**447**	**547**	599
ta004	1178	1245	1338	1386	**349**	455	**560**	**604**
ta006	1314	**1387**	**1530**	**1615**	**340**	**434**	**520**	569
ta008	1067	1146	1230	1281	**334**	**428**	540	**582**
ta010	1031	**1160**	**1323**	1386	305	401	**493**	549
ta012	1439	1567	1818	1983	**525**	647	**792**	858
ta014	1325	**1493**	1704	1790	**443**	580	720	755
ta016	1344	1515	1804	1945	**497**	**629**	**767**	827
ta018	**1406**	**1636**	**1949**	2118	**502**	617	763	**829**
ta020	1315	1477	1677	**1788**	**527**	**639**	**780**	**839**
ta022	**1871**	2193	**2564**	**2733**	**757**	**910**	**1105**	**1201**
ta024	1845	**2087**	2471	2652	**759**	**915**	**1095**	**1193**
ta026	**1835**	2115	2474	2665	**813**	**979**	**1166**	**1239**
ta028	1842	2137	2490	2699	**840**	1002	**1199**	1289
ta030	**1691**	1977	**2340**	2521	**839**	**1005**	1185	1262

5.5 Comparisons of the IABC and MILP

In this section, the proposed MILP model is validated by using instances of four scales, namely 10×5、10×10、20×5 and 20×10. The instances used were derived from the benchmarks mentioned earlier.

Table 7
Comparison of IABC algorithm and MILP mode

Instances	MILP	IABC
ta002_10_5	**262***	263
ta004_10_5	**268***	273
ta006_10_5	**265***	**265***
ta008_10_5	**220***	227
ta010_10_5	**227***	229
ta012_10_5	**268***	279
ta014_10_5	**202***	216
ta016_10_5	**195***	201
ta018_10_5	**227***	230
ta020_10_5	**263***	279
ta012_10_10	**422***	449
ta014_10_10	**356***	365
ta016_10_10	**362***	382
ta018_10_10	**366***	380
ta020_10_10	**459***	467
ta002_20_5	362	**346**
ta004_20_5	381	**349**
ta006_20_5	370	**340**
ta008_20_5	369	**334**
ta010_20_5	335	**305**
ta012_20_10	565	**525**
ta014_20_10	458	**443**
ta016_20_10	548	**497**
ta018_20_10	544	**502**
ta020_20_10	587	**527**

The maximum CPU runtime limit is 300×n×s milliseconds. The operating environment remains consistent with the aforementioned. To evaluate the performance of the IABC, a comparison was made with the MILP model. The validation instances and operating environment remain consistent with the aforementioned. The results obtained are shown in Table 7, where values marked with * represent the optimal solution. From Table 7, it can be observed that the MILP model can the optimal solutions for 15 small-sized instances. However, as the problem size increases, the MILP model fails to reach the optimal solution. As shown in the graph, when n = 20, the MILP model cannot achieve the optimal solutions within the specified time, and this is determined by the NP-hard nature of the problem. With regard to IABC algorithm, it cannot obtain the optimal solutions even for very small-sized instances. Obviously, 14 solutions of IABC are worse than that obtained by MILP model for small-sized instances. For example, IABC can only obtain 263 for instance ta002_10_5 instead of the optimal solution 262. With regard to the relatively large-sized instances, the obtained solutions of IABC are better than MILP model, which shows its efficiency for large-sized instances.

6. Conclusion

This paper studies HFSP-SDST to minimize the makespan. A MILP model is proposed for obtaining the optimal solutions for small-scale instances, and the CPLEX solver is used to validate its effectiveness. Given the NP-hard characteristics of HFSP-SDST, an IABC is also presented. In IABC, a hybrid decoding method is proposed that combines the strengths of both forward and backward decoding methods. Furthermore, an effective LS procedure is introduced to enlarge the solution space. Experiment results show that the hybrid decoding method outperforms single decoding methods in obtaining superior solutions, and the LS procedure is effective in enlarging the solution space. Specifically, the proposed IABC outperforms the existing algorithms and improves 61 current best solutions of benchmark instances.

In future work, more efficient neighborhood search methods and problem-specific optimization techniques will be explored. The research can be extended to multi-objective problems, and real-world manufacturing constraints, such as energy consumption and distributed shop scheduling in large-scale production, will be considered. Additionally, the algorithm will be applied to other optimization problems, including batch hybrid flow shop scheduling, blocking flow shop problems, and more.

Acknowledgements

This research is supported by the Funds for the Natural Science Foundation of Shandong Province [grant numbers ZR2021QE195 and ZR2021QF036], the National Natural Science Foundation of China [grant numbers 52205529 and 62303204], the Youth Innovation Team Program of Shandong Higher Education Institution [2023KJ206] and the Guangyue Youth Scholar Innovation Talent Program support received from Liaocheng University [LCUGYTD2022-03].

References

Allahverdi, A. (2015). The third comprehensive survey on scheduling problems with setup times/costs. *European journal of operational research, 246*(2), 345-378.

Behnamian, J., & Ghomi, S. F. (2011). Hybrid flowshop scheduling with machine and resource-dependent processing times. *Applied Mathematical Modelling, 35*(3), 1107-1123.

Dai, L.-L., Pan, Q.-K., Miao, Z.-H., Suganthan, P. N., & Gao, K.-Z. (2023). Multi-Objective Multi-Picking-Robot Task Allocation: Mathematical Model and Discrete Artificial Bee Colony Algorithm. *IEEE Transactions on Intelligent Transportation Systems*.

Engin, O., & Döyen, A. (2004). A new approach to solve hybrid flow shop scheduling problems by artificial immune system. *Future generation computer systems, 20*(6), 1083-1095.

Fan, J., Li, Y., Xie, J., Zhang, C., Shen, W., & Gao, L. (2021). A hybrid evolutionary algorithm using two solution representations for hybrid flow-shop scheduling problem. *IEEE Transactions on Cybernetics*.

Fattahi, P., Hosseini, S. M. H., Jolai, F., & Tavakkoli-Moghaddam, R. (2014). A branch and bound algorithm for hybrid flow shop scheduling problem with setup time and assembly operations. *Applied Mathematical Modelling, 38*(1), 119-134.

Garey, M. R., & Johnson, D. S. (1978). ``strong"np-completeness results: Motivation, examples, and implications. *Journal of the ACM (JACM), 25*(3), 499-508.

Gómez-Gasquet, P., Andrés, C., & Lario, F.-C. (2012). An agent-based genetic algorithm for hybrid flowshops with sequence dependent setup times to minimise makespan. *Expert Systems with Applications, 39*(9), 8095-8107.

Gupta, J. N. (1986). Flowshop schedules with sequence dependent setup times. *Journal of the Operations Research Society of Japan, 29*(3), 206-219.

He, X., Pan, Q.-K., Gao, L., Neufeld, J. S., & Gupta, J. N. (2024). Historical information based iterated greedy algorithm for distributed flowshop group scheduling problem with sequence-dependent setup times. *Omega, 123*, 102997.

Jemmali, M., & Hidri, L. (2023). Hybrid Flow Shop with Setup Times Scheduling Problem. *Comput. Syst. Sci. Eng, 44*, 563-577.

Johnson, S. M. (1954). Optimal two - and three - stage production schedules with setup times included. *Naval research logistics quarterly, 1*(1), 61-68.

Jungwattanakit, J., Reodecha, M., Chaovalitwongse, P., & Werner, F. (2005). An evaluation of sequencing heuristics for flexible flowshop scheduling problems with unrelated parallel machines and dual criteria. *Otto-von-Guericke-Universitat*

Magdeburg, 28(05), 1-23.

Jungwattanakit, J., Reodecha, M., Chaovalitwongse, P., & Werner, F. (2008). Algorithms for flexible flow shop problems with unrelated parallel machines, setup times, and dual criteria. *The International Journal of Advanced Manufacturing Technology, 37*, 354-370.

Karaboga, D. (2005). *An idea based on honey bee swarm for numerical optimization*. Retrieved from

Karaboga, D., & Basturk, B. (2008). On the performance of artificial bee colony (ABC) algorithm. *Applied Soft Computing, 8*(1), 687-697.

Khare, A., & Agrawal, S. (2019). Scheduling hybrid flowshop with sequence-dependent setup times and due windows to minimize total weighted earliness and tardiness. *Computers & Industrial Engineering, 135*, 780-792.

Kurz, M. E., & Askin, R. G. (2003). Comparing scheduling rules for flexible flow lines. *International Journal of Production Economics, 85*(3), 371-388.

Lee, G.-C., Hong, J. M., & Choi, S.-H. (2015). Efficient heuristic algorithm for scheduling two-stage hybrid flowshop with sequence-dependent setup times. *Mathematical Problems in Engineering, 2015*.

Liao, C.-J., Tjandradjaja, E., & Chung, T.-P. (2012). An approach using particle swarm optimization and bottleneck heuristic to solve hybrid flow shop scheduling problem. *Applied Soft Computing, 12*(6), 1755-1764.

Liao, Y., Zhantao, L., Li, X., & Chenfeng, P. (2022). Heuristics for the Hybrid Flow Shop Scheduling Problem with Sequence-Dependent Setup times. *Mathematical Problems in Engineering, 2022*.

Meng, L., Duan, P., Gao, K., Zhang, B., Zou, W., Han, Y., & Zhang, C. (2024). MIP modeling of energy-conscious FJSP and its extended problems: From simplicity to complexity. *Expert Systems with Applications, 241*, 122594.

Meng, L., Gao, K., Ren, Y., Zhang, B., Sang, H., & Chaoyong, Z. (2022). Novel MILP and CP models for distributed hybrid flowshop scheduling problem with sequence-dependent setup times. *Swarm and Evolutionary Computation, 71*, 101058.

Meng, L., Zhang, C., Ren, Y., Zhang, B., & Lv, C. (2020). Mixed-integer linear programming and constraint programming formulations for solving distributed flexible job shop scheduling problem. *Computers & Industrial Engineering, 142*, 106347.

Meng, L., Zhang, C., Zhang, B., Gao, K., Ren, Y., & Sang, H. (2023). MILP modeling and optimization of multi-objective flexible job shop scheduling problem with controllable processing times. *Swarm and Evolutionary Computation, 82*, 101374.

Naderi, B., Zandieh, M., Balagh, A. K. G., & Roshanaei, V. (2009). An improved simulated annealing for hybrid flowshops with sequence-dependent setup and transportation times to minimize total completion time and total tardiness. *Expert Systems with Applications, 36*(6), 9625-9633.

Nishi, T., Hiranaka, Y., & Inuiguchi, M. (2010). Lagrangian relaxation with cut generation for hybrid flowshop scheduling problems to minimize the total weighted tardiness. *Computers & Operations Research, 37*(1), 189-198.

Ozsoydan, F. B., & Sağir, M. (2021). Iterated greedy algorithms enhanced by hyper-heuristic based learning for hybrid flexible flowshop scheduling problem with sequence dependent setup times: a case study at a manufacturing plant. *Computers & Operations Research, 125*, 105044.

Pan, Q.-K., & Dong, Y. (2014). An improved migrating birds optimisation for a hybrid flowshop scheduling with total flowtime minimisation. *Information Sciences, 277*, 643-655.

Pan, Q.-K., Gao, L., Li, X.-Y., & Gao, K.-Z. (2017). Effective metaheuristics for scheduling a hybrid flowshop with sequence-dependent setup times. *Applied Mathematics and Computation, 303*, 89-112.

Pan, Q.-K., Wang, L., Li, J.-Q., & Duan, J.-H. (2014). A novel discrete artificial bee colony algorithm for the hybrid flowshop scheduling problem with makespan minimisation. *Omega, 45*, 42-56.

Pan, Q.-K., Wang, L., Mao, K., Zhao, J.-H., & Zhang, M. (2012). An effective artificial bee colony algorithm for a real-world hybrid flowshop problem in steelmaking process. *IEEE Transactions on Automation Science and Engineering, 10*(2), 307-322.

Rashidi, E., Jahandar, M., & Zandieh, M. (2010). An improved hybrid multi-objective parallel genetic algorithm for hybrid flow shop scheduling with unrelated parallel machines. *The International Journal of Advanced Manufacturing Technology, 49*, 1129-1139.

Ruiz, R., & Maroto, C. (2006). A genetic algorithm for hybrid flowshops with sequence dependent setup times and machine eligibility. *European journal of operational research, 169*(3), 781-800.

Salvador, M. (1973). Asolution to a special class of flow shop scheduling problems. *Lecture Notes in Economics and Mathematical Systems, 86*, 83-91.

Tian, H., Li, K., & Liu, W. (2016). A pareto-based adaptive variable neighborhood search for biobjective hybrid flow shop scheduling problem with sequence-dependent setup time. *Mathematical Problems in Engineering, 2016*.

Xu, Y., & Wang, L. (2011). Differential evolution algorithm for hybrid flow-shop scheduling problems. *Journal of Systems Engineering and Electronics, 22*(5), 794-798.

Zandieh, M., Ghomi, S. F., & Husseini, S. M. (2006). An immune algorithm approach to hybrid flow shops scheduling with sequence-dependent setup times. *Applied Mathematics and Computation, 180*(1), 111-127.

Appendix

The improved best solutions are given as follows:

SSD10_P13 ta018:
makespan:1406
operation sequence:
6 8 14 17 4 7 1 11 5 3 13 18 9 15 10 19 0 2 16 12
SSD10_P13 ta022:
makespan:1871
operation sequence:
4 18 14 7 16 12 17 8 11 0 6 9 5 15 10 13 2 19 3 1
SSD10_P13 ta026:
makespan:1835
operation sequence:
10 0 18 3 19 5 11 2 7 17 12 6 4 13 9 1 14 15 8 16
SSD10_P13 ta030:
makespan:1691
operation sequence:
13 12 2 8 10 14 18 4 1 3 9 6 19 7 17 0 5 11 15 16
SSD50_P13 ta002:
makespan:1185
operation sequence:
14 7 3 19 18 17 4 1 11 6 13 16 2 8 15 10 5 9 0 12
SSD50_P13 ta006:
makespan:1387
operation sequence:
16 0 14 7 19 11 6 9 17 1 2 10 8 15 13 3 18 12 4 5
SSD50_P13 ta010:
makespan:1160
operation sequence:
9 15 12 14 5 6 7 16 19 2 11 4 1 13 10 8 18 0 3 17
SSD50_P13 ta014:
makespan:1493
operation sequence:
2 5 0 11 10 17 15 8 9 19 1 16 13 7 6 12 3 4 14 18
SSD50_P13 ta018:
makespan:1636
operation sequence:
4 8 14 11 5 6 13 18 16 1 7 17 2 9 15 10 3 0 19 12
SSD50_P13 ta024:
makespan:2087
operation sequence:
12 19 16 4 5 13 6 9 3 7 18 1 11 14 0 10 2 17 15 8
SSD100_P13 ta002:
makespan:1374
operation sequence:
6 3 2 1 13 4 18 17 5 9 7 8 19 0 14 16 11 15 10 12
SSD100_P13 ta006:
makespan:1530
operation sequence:
8 0 10 13 18 14 16 7 3 19 9 11 6 12 17 1 4 15 5 2
SSD100_P13 ta010:
makespan:1323
operation sequence:
9 10 4 1 13 8 18 0 15 12 14 5 6 7 16 19 2 11 3 17
SSD100_P13 ta018:
makespan:1949
operation sequence:
14 7 17 8 9 15 18 16 1 11 5 6 13 3 2 4 10 0 19 12
SSD100_P13 ta022:
makespan:2564
operation sequence:
7 12 4 16 10 11 18 9 2 6 0 5 15 13 1 14 17 19 8 3
SSD100_P13 ta030:
makespan:2340
operation sequence:
12 4 6 13 10 14 16 8 17 11 18 2 19 9 1 7 0 3 5 15
SSD125_P13 ta002:
makespan:1463
operation sequence:
3 9 7 8 13 4 18 17 5 19 0 14 16 11 6 2 1 15 10 12
SSD125_P13 ta006:
makespan:1615
operation sequence:
4 16 0 13 3 18 14 7 19 11 6 9 10 2 5 15 8 1 12 17
SSD125_P13 ta020:
makespan:1788
operation sequence:
2 17 18 5 6 10 3 1 15 16 14 12 8 0 19 13 7 9 11 4
SSD125_P13 ta022:

SSD10_P3 ta026:
makespan:813
operation sequence:
3 10 16 12 18 2 13 5 14 1 19 17 0 7 4 15 6 8 11 9
SSD10_P3 ta028:
makespan:840
operation sequence:
14 19 15 10 0 5 8 16 6 11 17 3 7 12 13 1 4 2 18 9
SSD10_P3 ta030:
makespan:839
operation sequence:
13 12 4 16 19 8 15 14 6 5 1 2 7 10 9 11 3 0 17 18
SSD50_P3 ta002:
makespan:447
operation sequence:
17 9 14 18 0 1 15 10 19 2 7 4 16 3 13 5 8 12 11 6
SSD50_P3 ta006:
makespan:434
operation sequence:
15 9 13 8 19 4 16 7 2 12 17 0 14 5 1 18 6 11 3 10
SSD50_P3 ta008:
makespan:428
operation sequence:
9 5 4 8 11 19 14 2 12 15 17 3 16 18 10 6 13 7 0 1
SSD50_P3 ta016:
makespan:629
operation sequence:
7 18 1 10 13 4 11 15 3 17 9 16 2 0 12 8 14 6 19 5
SSD50_P3 ta020:
makespan:639
operation sequence:
11 1 15 17 13 4 3 18 6 12 16 2 19 14 0 9 5 8 10 7
SSD50_P3 ta022:
makespan:910
operation sequence:
16 6 12 13 14 10 0 19 5 18 8 17 2 11 9 3 1 15 4 7
SSD50_P3 ta024:
makespan:915
operation sequence:
15 13 17 4 3 2 12 9 19 8 11 16 7 1 6 14 18 10 5 0
SSD50_P3 ta026:
makespan:979
operation sequence:
2 15 14 13 18 17 10 5 9 1 7 19 12 11 3 8 16 6 0 4
SSD50_P3 ta030:
makespan:1005
operation sequence:
15 16 14 5 12 19 17 13 6 2 4 8 9 10 18 3 1 7 0 11
SSD100_P3 ta002:
makespan:547
operation sequence:
5 13 16 4 3 7 18 9 2 19 10 17 0 8 11 12 1 6 15 14
SSD100_P3 ta004:
makespan:560
operation sequence:
12 9 8 15 14 0 16 13 18 19 2 5 7 6 1 11 4 3 10 17
SSD100_P3 ta006:
makespan:520
operation sequence:
5 15 13 16 2 17 3 19 0 8 4 9 1 6 12 7 18 11 10 14
SSD100_P3 ta010:
makespan:493
operation sequence:
6 3 15 14 12 16 5 8 18 19 11 13 0 4 10 1 7 17 2 9
SSD100_P3 ta012:
makespan:792
operation sequence:
7 3 16 9 19 1 17 5 14 15 4 13 0 2 8 6 10 11 18 12
SSD100_P3 ta016:
makespan:767
operation sequence:
12 19 7 10 4 2 0 6 13 1 18 17 8 14 3 15 9 5 11 16
SSD100_P3 ta020:
makespan:780
operation sequence:
11 2 15 17 12 18 6 13 19 3 4 1 14 7 9 0 16 10 8 5
SSD100_P3 ta022:

makespan:2733
operation sequence:
7 12 4 16 10 11 18 9 2 6 0 5 13 1 15 14 17 19 8 3
SSD10_P3 ta002:
makespan:346
operation sequence:
5 0 18 13 3 16 9 17 14 19 8 7 1 10 4 11 2 6 12 15
SSD10_P3 ta004:
makespan:349
operation sequence:
15 17 12 16 18 8 9 13 7 2 10 14 5 4 1 19 11 3 0 6
SSD10_P3 ta006:
makespan:340
operation sequence:
19 17 15 9 4 2 5 16 13 6 12 8 7 0 3 10 18 1 14 11
SSD10_P3 ta008:
makespan:334
operation sequence:
9 8 2 0 4 16 11 12 19 6 18 3 14 15 13 7 10 17 5 1
SSD10_P3 ta012:
makespan:525
operation sequence:
6 18 7 3 8 4 12 15 11 1 16 0 19 13 14 10 17 5 9 2
SSD10_P3 ta014:
makespan:443
operation sequence:
12 15 18 2 0 11 14 16 9 7 10 17 5 6 1 4 8 19 3 13
SSD10_P3 ta016:
makespan:497
operation sequence:
8 19 4 3 17 7 13 12 14 10 9 2 1 0 6 18 11 16 5 15
SSD10_P3 ta018:
makespan:502
operation sequence:
13 12 5 15 3 10 4 7 1 14 8 0 17 2 18 16 11 19 9 6
SSD10_P3 ta020:
makespan:527
operation sequence:
11 13 17 3 2 1 18 6 12 15 0 14 4 19 5 7 16 9 10 8
SSD10_P3 ta022:
makespan:757
operation sequence:
0 16 7 11 10 17 4 6 19 8 18 15 3 5 2 14 1 13 12 9
SSD10_P3 ta024:
makespan:759
operation sequence:
17 3 1 19 9 16 12 10 14 4 2 15 6 8 18 0 5 13 11 7

makespan:1105
operation sequence:
10 0 13 12 16 8 11 4 14 5 19 18 3 2 7 1 9 17 15 6
SSD100_P3 ta024:
makespan:1095
operation sequence:
17 19 12 4 10 7 15 1 11 3 2 8 9 14 16 13 6 5 18 0
SSD100_P3 ta026:
makespan:1166
operation sequence:
14 16 2 13 15 17 18 0 19 5 3 10 12 1 8 11 6 9 4 7
SSD100_P3 ta028:
makespan:1199
operation sequence:
19 15 2 0 14 9 12 6 5 8 11 1 13 17 16 3 10 7 18 4
SSD125_P3 ta004:
makespan:604
operation sequence:
18 9 16 17 12 19 15 1 14 8 10 6 11 13 7 3 4 5 0 2
SSD125_P3 ta008:
makespan:582
operation sequence:
4 11 9 2 16 8 15 3 13 0 18 14 6 12 19 17 7 5 10 1
SSD125_P3 ta018:
makespan:829
operation sequence:
5 9 15 18 16 1 8 3 4 10 12 11 13 7 0 2 14 19 6 17
SSD125_P3 ta020:
makespan:839
operation sequence:
11 15 13 12 18 6 2 17 7 1 3 4 0 14 19 16 9 10 5 8
SSD125_P3 ta022:
makespan:1201
operation sequence:
4 12 8 13 17 19 2 0 18 16 11 3 10 7 1 5 14 9 15 6
SSD125_P3 ta024:
makespan:1193
operation sequence:
13 2 19 10 17 4 1 12 9 3 15 6 8 14 7 11 0 18 5 16
SSD125_P3 ta026:
makespan:1239
operation sequence:
14 10 13 3 4 17 18 16 0 15 9 19 1 5 8 7 2 12 11 6