mathematics

MDPI

Article

A Variant Iterated Greedy Algorithm Integrating Multiple Decoding Rules for Hybrid Blocking Flow Shop Scheduling Problem

Yong Wang, Yuting Wang * and Yuyan Han *

School of Computer Science, Liaocheng University, Liaocheng 252059, China; 2110170110@stu.lcu.edu.cn
* Correspondence: wangyuting@lcu.edu.cn (Y.W.); hanyuyan@lcu.edu.cn (Y.H.);
Tel.: +86-156-6635-1136 (Y.W.); +86-188-6497-4734 (Y.H.)

Abstract: This paper studies the hybrid flow shop scheduling problem with blocking constraints (BHFSP). To better understand the BHFSP, we will construct its mixed integer linear programming (MILP) model and use the Gurobi solver to demonstrate its correctness. Since the BHFSP exists parallel machines in some processing stages, different decoding strategies can obtain different makespan values for a given job sequence and multiple decoding strategies can assist the algorithm to find the optimal value. In view of this, we propose a hybrid decoding strategy that combines both forward decoding and backward decoding to select the minimal objective function value. In addition, a hybrid decoding-assisted variant iterated greedy (VIG) algorithm to solve the above MILP model. The main novelties of VIG are a new reconstruction mechanism based on the hybrid decoding strategy and a swap-based local reinforcement strategy, which can enrich the diversity of solutions and explore local neighborhoods more deeply. This evaluation is conducted against the VIG and six state-of-the-art algorithms from the literature. The 100 tests showed that the average makespan and the relative percentage increase (RPI) values of VIG are 1.00% and 89.6% better than the six comparison algorithms on average, respectively. Therefore, VIG is more suitable to solve the studied BHFSP.

Keywords: blocking; hybrid decoding; hybrid flow shop scheduling; iterated greedy

MSC: 93B28

Citation: Wang, Y.; Wang, Y.; Han, Y. A Variant Iterated Greedy Algorithm Integrating Multiple Decoding Rules for Hybrid Blocking Flow Shop Scheduling Problem. *Mathematics* **2023**, *11*, 2453. https://doi.org/10.3390/math11112453

Academic Editors: Ana M. Madureira, Joao Ferreira and André Santos

Received: 3 April 2023
Revised: 17 May 2023
Accepted: 23 May 2023
Published: 25 May 2023

1. Introduction

The flow shop scheduling problem (FSSP) is prevalent in a variety of industries, including semiconductor, automobile, and textile [1–3]. To enhance production efficiency, some identical parallel machines are considered in the flow shop. This production pattern is hybrid or flexible and forms a new scheduling problem, called the hybrid flow shop scheduling problem (HFSP). In HFSP, 'stages' refers to the processing centers or machining operations. Each job must go through all stages in a predetermined sequence. Parallel machines may exist in each processing stage, allowing HFSP to process multiple jobs simultaneously. Therefore, HFSP has found wide applications in industries such as steel-making, electronics production, and chemistry [3–6]. In the literature, FSSP has been certified that it is an NP-hard problem [7]. However, HFSP is much more complex than FSSP. Unlike FSSP, which only considers the ordering of jobs, HFSP requires a more comprehensive approach that balances both job sequencing and machine assignment.

In HFSP, at least one processing stage contains identical parallel machines, and each job can be assigned to any one of these parallel machines for processing. HFSP can be divided into two types based on cost-related restrictions: those with infinite buffers and those with finite buffers. The HFSP with infinite buffers cannot cause job blocking due to sufficient buffers between stages that can deposit completed jobs. However, the HFSP with finite buffers may result in job blocking because of finite intermediate buffers. The

term "blocking" refers to the situation where a completed job in the current stage is held up on the machine because all parallel machines in the subsequent stage are busy [8]. The blocking constraint may increase the complete time of jobs and reduce production efficiency. Therefore, it is necessary to search for an optimal scheduling sequence that minimizes the blocking of jobs. Based on this, the present study considers the absence of buffer constraints in HFSP, resulting in the formation of a new problem known as the blocking HFSP (BHFSP). Furthermore, the optimization objective of our study is the makespan. It is a very important indicator for actual production. The shorter the makespan, the faster the jobs are completed. A longer makespan, on the other hand, means that jobs are taking longer to complete, which can result in delays and lower productivity. Therefore, by reducing the makespan, manufacturers can enhance resource utilization and meet customer demands.

As mentioned above, when a blocked state occurs, the job is blocked on the current machines until a machine is available in the next stage. Therefore, before designing corresponding strategies for BHFSP, it is necessary to analyze the challenges of the problem. The details are as follows:

(1) Due to there being no buffers between two adjacent stages, once a job is blocked on the current stage, it cannot be processed until the machine at the next stage is available, which may increase the wait time of jobs, prolong the completion time of jobs, and reduce the production efficiency.
(2) The HFSP in production is often a large-scale problem, involving a large number of jobs and machines, and it is difficult to generate an optimal scheduling solution in a short time.
(3) The solution may fall into a local optimum when the job is blocked on the machine. Therefore, it is crucial to design some appropriate and effective strategies to reduce the blocking time.

The iterated greedy (IG) algorithm is an efficient and simple iterative method when optimizing FSS problems [9,10]. It usually uses heuristics to obtain a better initial solution, and then adopts a destruction and reconstruction strategy to enhance the quality of the initial solution. The IG algorithm is a simple and practical heuristic algorithm that can be applied to real-world problems. It is a single-objective optimization algorithm that can be enhanced by utilizing previous search results to improve search efficiency and accuracy. However, the limitation of the IG algorithm is also noticeable, as it is susceptible to getting stuck in local optima during the iteration process. However, the IG algorithm still performs well as long as we make appropriate improvements to IG based on the problem characteristics. For example, the improved IG algorithm can effectively solve the blocking FSSP by considering its specific characteristics [11]. To sum up, we propose a variant IG (VIG) to optimize BHFSP. Our innovations are summarized as follows:

(1) We construct a mixed integer linear programming (MILP) model and adopt the Gurobi solver to demonstrate its correctness.
(2) Two different decoding strategies have been designed to calculate the objective in BHFSP. By adopting the hybrid decoding strategy, it is possible to find a smaller maximum completion time.
(3) A new reconstruction mechanism based on forward and backward decoding strategies is proposed to enrich the diversity of solutions.
(4) A swap-based local reinforcement strategy is developed to explore local neighborhoods more deeply.
(5) Abundant simulation experiments have been performed and demonstrated that the VIG shows superiority in solving BHFSP compared with the six algorithms in existence on 100 test instances.

The rest of this study can be stated as follows: Section 2 puts forward the literature related to BHFSP. Section 3 formulates the BHFSP. In Section 4, a variant IG algorithm is designed. Section 5 lists and discusses the simulation experiments. In the end, a summary and future research points are presented in Section 6.

2. Literature Review

2.1. Hybrid Flow Shop Scheduling Problems

The methods commonly used to solve HFSP fall into three categories: exact methods [12], heuristic methods [13], and metaheuristic algorithms [14]. The first category mainly includes the branch-and-bound method [15] and the dynamic programming approach [16], which can obtain the optimal solution using solvers but are best suited for small-scale problems due to their long computation time. However, for HFSPs with large scales, it is hard to obtain an optimum solution within a limited time. In view of this, the second and third categories of methods, heuristic, and metaheuristic have been developed and applied to optimize large-scale problems. The NEH heuristic, initially proposed by Nawaz, Enscore, and Ham to optimize FSSP [17], has been further developed in subsequent studies. Ronconi et al. developed the combination of MinMax and NEH heuristics, called MME, to optimize the blocking FSSP [18]. Likewise, the author obtained PFE by combining the profile fitting (PF) heuristic with NEH. On this basis, Pan and Wang proposed a combined PE-NEH heuristic to solve FSSP with blocking constraints [19]. Fernandez-Viagas et al. used two memory-based composite heuristics to address the HFSP [20]. Öztop et al. presented a greedy adaptive disturbance method combining NEH (GRASP_NEH(x)) for the HFSP [2]. Although these heuristic methods may not be as effective as meta-heuristics in solving complex problems, they are often considered a useful initialization strategy due to their ability to produce high-quality initial solutions.

In recent years, HFSP and its extensions have attracted the attention of scholars. To solve the general HFSP, Xiao et al. studied the work shift HFSP and presented a modified genetic algorithm (GA) according to a scheduling rule to optimize the weighted makespan [21]. Jin et al. adopted a hybrid simulated annealing (SA) method combining different scheduling rules [22]. To effectively solve HFSP, Wang et al. proposed a distribution estimation algorithm (EDA) that uses a probabilistic model to find promising solutions [23]. Pan et al. employed a variety of heuristic methods to improve the performance of the initial solution and embedded them into the discrete artificial bee colony (DABC) to optimize the makespan of HFSP [24]. Due to the lack of local search ability of the swarm intelligence method, Li et al. designed a variety of neighborhood structures to overcome this challenge and utilized the advantages of EDA and chemical-reaction optimization (CRO) to design a hybrid variable neighborhood search (HVNS) algorithm to solve HFSP [25]. To enrich the method of solving HFSP, Lin et al. proposed a chaos-enhanced SA (CSA) algorithm to optimize makespan [26]. To fill the gap in energy conservation studies for the five phases of HFSP, Utama et al. designed a hybrid Aquila Optimizer (HAO) to optimize the total energy consumption [27]. Subsequently, Utama et al. created a hybrid Archimedes optimization algorithm (HAOA), which hopefully will help provide new insights into advanced HAOA methods for solving HFSP [28]. In the real world, the expansion of HFSP with production limitations is more realistic than the general HFSP. Zhang et al. studied the HFSP with lot streaming to improve production efficiency and presented an effective modified migrating birds optimization (EMBO) algorithm to solve it [29]. In addition, the collaborative mechanism is considered. Zhang et al. proposed a collaborative variable neighborhood descent (CVND) for HFSP with consistent sublots [30]. To overcome the drawback that IG tends to fall into a local optimum, Li et al. introduced a restart strategy and proposed an adaptive IG to solve the hybrid no-idle flow shop scheduling problem [31]. In a multiple-factory environment, Cui et al. proposed an improved multi-population genetic algorithm (MPGA) that considers inter-factory neighborhood structure to solve the HFSP [32]. Considering group scheduling, Qin et al. designed a block-based neighborhood selection strategy according to the characteristics of the problem [33]. In addition, Qin et al. also studied the distributed HFSP, whose optimization goal is to minimize the total energy consumption [34]. For the energy consumption metric, Wang et al. investigated the energy-efficient fuzzy HFSP (EFHFSP) with the variable machine speed and extended the existing non-dominated sorting genetic algorithm-II (NSGA-II) [35]. According to the above literature, we selected some representative literature related to our considered problem

and stated their problems and algorithms in Table 1. Through an analysis of problems and algorithms, we found that most of the literature does not consider the blocking constraint and adopts swarm intelligence algorithms, i.e., GA, SA, EDA, DABC, EMBO, etc. These algorithms can provide multiple solutions to improve the diversity of solutions. However, they have more parameters and complicated structures. In the exploration of a single solution neighborhood, these swarm intelligence algorithms are slightly less effective than the IG algorithm [36]. Thus, a modified IG algorithm is adopted to solve the BHFSP due to the fact that it can more intensively explore a single solution and is very effective at improving the reinforcement of the solution.

Table 1. Review of the studies on HFSP.

Authors	HFSP with Blocking	Objective (Minimizing)	Algorithms	MILP Model
Xiao et al. [21]	No	Weighted makespan	GA	No
Jin et al. [22]	No	Makespan	SA	Yes
Wang et al. [23]	No	Makespan	EDA	Yes
Pan et al. [24]	No	Makespan	DABC	Yes
Li et al. [25]	No	Makespan	HVNS	No
Lin et al. [26]	No	Makespan	CSA	Yes
Utama et al.[27]	No	Total energy consumption	HAO	Yes
Utama et al. [28]	No	Total energy consumption	HAOA	No
Zhang et al. [29]	No	Total flow time	EMBO	Yes
Zhang et al. [30]	No	Makespan	CVND	Yes
Li et al. [31]	No	Total flow time	adaptive IG(AIG)	Yes
Cui et al. [32]	No	Makespan	MPGA	No
Qin et al. [33]	Yes	Makespan	IG	Yes
Qin et al. [34]	Yes	Total energy consumption	IG	Yes
Wang et al. [35]	No	Total energy consumption	NSGA-II	Yes
This research	Yes	Makespan	VIG	Yes

Regarding the blocking constraints, we note the following research. Pan et al. analyzed the problem characteristics of FSSP with blocking constraints (BFSP) and developed several effective heuristics to expand the problem-solving methods [19]. Shao et al. proposed a discrete invasive weed optimization method (DIWO) to solve the BFSP with the makespan criterion [37]. Ribas et al. considered realistic production setup times and proposed an IG algorithm to handle the BFSP [11]. Considering the distributed environment, Han et al. proposed an effective IG by combining learning-based selecting multiple neighborhood constructions to solve the discrete BFSP [38]. Meanwhile, Qin et al. used a collaborative IG to solve the HFSP without buffers [39]. Zhang et al. used a discrete whale swarm algorithm to solve HFSP with limited buffers [40]. Considering the machine energy consumption, Qin et al. proposed an improved IG to address the HFSP including energy consumption and blocking constraints [36].

From the literature above, it is evident that research on blocking constraints is still ongoing, and there are many practical applications for BHFSP, such as container trailer scheduling problems [41], train track scheduling [42], ship manufacturing [43], and concrete blocks [44]. Therefore, studying BHFSP has practical significance.

2.2. Iterated Greedy Algorithm

IG is widely used to solve FSSP because of its simple structure, few parameters, and high efficiency [9]. Thus, many scholars have proposed some improved IG algorithms to solve scheduling problems. Rodriguez et al. employed IG to address the FSSP of unrelated parallel machines [45]. Fernandez-Viagas et al. used the eight variants of IG based on some search-based heuristics to vary their population size and optimize the total delay criterion [46]. To minimize the makespan, Rubén Ruiz, Pan, and Naderi used an improved IG to optimize FSSP under the distributed environment [10]. The authors extended the destruction, reconstruction, and local search strategies to create a two-level IG based on

mixed properties of the problem. For the distributed blocking FSSP (DBFSP), Chen et al. described a population iterative greedy (PBIG) based on the advantages of the crowd search method and iterative greedy algorithm [47]. Pan and Ruiz [48] first constructed the MILP model for the mixed no-idle flow shop problem and designed an efficient IG combining a speed-up neighborhood insertion strategy. Qin et al. presented a double level mutation IG (IGDLM) to optimize HFSP [49]. Missaoui and Ruiz [50] proposed a parameter-less IG without calibration to address HFSP with setup time and lead-time window restraints.

From the literature cited in this paper, i.e., [7,40–45,48], we know that the IGA has shown good performance among the metaheuristics for many scheduling problems, especially in the flow shop scheduling field. For example, (1) as mentioned in the literature [48], unlike other metaphor-based algorithms, IGA is a simple iterated search method with no complex structures, which is easy to be coded and understood. Therefore, it is available to be replicated and used for other related problems. (2) In addition, the latest literature [44] proposed a modified IGA, named IGDLM algorithm that used to solve the BHFSP. This work compares many advanced algorithms, and simulation results have proved that the IGDLM outperforms the existing five compared algorithms. (3) Referring to [45], the authors compared the IG algorithm with different strategies and proved that it is feasible to use the IG algorithm to solve the HFSP and results show that the IG series algorithms generally obtain good solutions. The advantages of the IG algorithm are attributed to the simplicity of the algorithm framework with few parameters, ease of integration, and good reinforcement and local convergence performance. Based on the above analyses, we finally selected the IG algorithm to solve the BHFSP considered in this paper.

Although the above studies have used the IG algorithm to solve the scheduling problem, they have also revealed its limitations. Specifically, the IG algorithm has a strong local perturbation ability, but correspondingly, it is easy to make the solution converge prematurely and fall into a local optimum. Additionally, from the existing literature, the IG algorithm only optimizes a single solution, leading to poor diversity of solutions. To better address the BHFSP problem, we design corresponding strategies to overcome these limitations. We consider using parallel optimization to simultaneously optimize two solutions and using a collaborative mechanism to interact between the two solutions, increasing their diversity. Due to the characteristics of the HFSP, there are different decoding methods available. Therefore, we incorporate a hybrid decoding strategy into the parallel framework. This framework includes a new reconstruction mechanism that combines forward and backward decoding strategies to increase the diversity of solutions. Additionally, we incorporate a swap-based local reinforcement strategy to enable a deeper exploration of local neighborhoods. The proposed strategies can decrease the computational time of the objective function and raise the diversity of solutions.

3. Problem Statement

BHFSP is described as follows: Each job within a set having J jobs are processed at S sequential stages according to the same sequence. For each stage s $(s = 1, 2, \cdots, S)$, there exists M_s $(M_s \geq 1)$ identical parallel machines. Furthermore, there is at least one stage containing more than one identical parallel machine. We assume that there are no intermediate buffers between neighboring stages. The processing time of job j at stage s is denoted as $p_{j,s}$. Our optimization goal is to find a reasonable scheduling sequence with minimal makespan.

Typically, the following discussion of BHFSP is based on the following six hypotheses:

(1) At time 0, all jobs can be processed and machines are free.
(2) At any time, a machine can only process one job at a time, and a job cannot be processed by multiple machines at the same time.
(3) All jobs are continuously processed without any interruptions.
(4) Each job goes through all the processing stages in turn and is processed by only one machine at a stage.
(5) Skipping a stage or ending early is not allowed.

(6) No intermediate buffer exists in any neighboring stages. If all machines at the next stage are busy after a job is processed at the current machine, the finished job will be blocked until one machine at the downstream stage is free.

3.1. Mathematical Model

Objective:

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

Constraints:

$$\sum_{m=1}^{M_s} y_{j,s,m} = 1, \forall j \in \{1, 2, \cdots, J\}, \forall s \in \{1, 2, \cdots, S\} \tag{2}$$

$$z_{j,j',s} + z_{j',j,s} \leq 1, \forall j, j' \in \{1, 2, \cdots, J\}, j < j', \forall s \in \{1, 2, \cdots, S\} \tag{3}$$

$$\begin{aligned} &z_{j,j',s} + z_{j',j,s} \geq y_{j,s,m} + y_{j',s,m} - 1, \forall j, j' \in \{1, 2, \cdots, J\}, j < j', \\ &\forall s \in \{1, 2, \cdots, S\}, \forall m \in \{1, 2, \cdots, M_s\} \end{aligned} \tag{4}$$

$$C_{j,s} \geq p_{j,s}, \forall j \in \{1, 2, \cdots, J\}, \forall s \in \{1, 2, \cdots, S\} \tag{5}$$

$$\begin{aligned} &C_{j',s} \geq D_{j,s} + p_{j',s} + \left(y_{j,s,m} + y_{j',s,m} + z_{j,j's} - 3\right) \cdot U, \forall j, j' \in \{1, 2, \cdots, J\}, j \neq j', \\ &\forall s \in \{1, 2, \cdots, S\}, \forall m \in \{1, 2, \cdots, M_s\} \end{aligned} \tag{6}$$

$$C_{j,s+1} = D_{j,s} + p_{j,s+1}, \forall j \in \{1, 2, \cdots, J\}, \forall s \in \{1, 2, \cdots, S-1\} \tag{7}$$

$$D_{j,s} \geq C_{j,s}, \forall j \in \{1, 2, \cdots, J\}, \forall s \in \{1, 2, \cdots, S\} \tag{8}$$

$$C_{max} \geq D_{j,S}, \forall j \in \{1, 2, \cdots, J\} \tag{9}$$

Constraint (1) is the optimized makespan. Constraint (2) guarantees a job is processed on only one machine of each stage. Constraint (3) refers to that $z_{j,j',s}$ and $z_{j,j',s}$ cannot equal 1 at the same time. Constraint (4) enforces that if jobs j and j' can be processed on the same machine at stage s, one of $z_{j,j',s}$ and $z_{j,j',s}$ must be equal 1 and the other equal 0. Constraint (5) guarantees that the completion time of job j at stage s is larger than or equal to that of the processing time of job j at stage s. For job j and job j', if job j is processed on the same machine before j' at stage s, the completion time of job j' at stage s must be not less than the departure time of job j at stage s plus $p_{j',s}$, ensured by Constraint (6). Constraint (7) defines that the completion time of a job at a stage is not less than the processing time at the same stage plus its departure time at the previous one. Constraint (8) ensures that at each stage, the departure time of a job is larger than or equal to its completion time. Constraint (9) guarantees that the makespan is equal to or larger than the departure time of all jobs at the last stage.

3.2. Example Instance

Consider an example with $J = 6$, $S = 2$, $M_1 = 2$ and $M_2 = 2$. Table 2 lists all $p_{j,s}$. One possible solution of the example is $z_{1,3,1} = 1$, $z_{1,5,1} = 1$, $z_{3,5,1} = 1$, $z_{2,4,1} = 1$, $z_{2,6,1} = 1$, $z_{4,6,1} = 1$, $z_{1,3,2} = 1$, $z_{1,4,2} = 1$, $z_{1,5,2} = 1$, $z_{3,4,2} = 1$, $z_{3,6,2} = 1$, $z_{4,6,2} = 1$, $z_{2,5,2} = 1$, $y_{1,1,1} = 1$, $y_{3,1,1} = 1$, $y_{5,1,1} = 1$, $y_{2,1,2} = 1$, $y_{4,1,2} = 1$, $y_{6,1,2} = 1$, $y_{1,2,1} = 1$, $y_{3,2,1} = 1$, $y_{4,2,1} = 1$, $y_{6,2,1} = 1$, $y_{2,2,2} = 1$. A scheduling Gantt chart of the given example is displayed in Figure 1.

Table 2. Processing times of jobs at stage 1 and stage 2.

Job	Stage 1	Stage 2
1	1	3
2	2	4
3	2	4
4	3	1
5	2	3
6	1	2

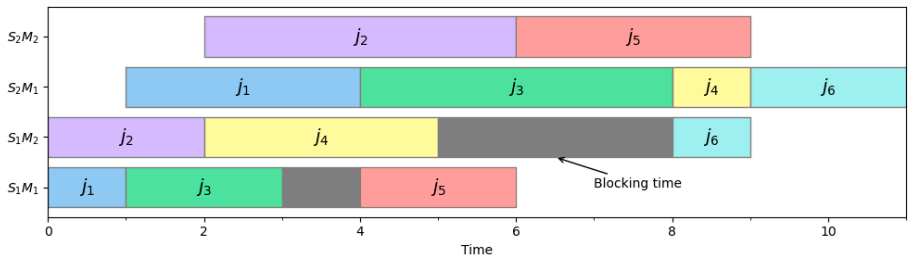

Figure 1. The scheduling Gantt chart of the given example.

In Figure 1, the makespan of above example is 11. At moment 3, job 3 has been finished on machine 1 at stage 1. However, job 3 must be blocked on machine 1 until moment 4 because all machines in the next stage are busy. Similarly, at moment 5, job 4 remained on machine 2 at stage 1 until moment 8.

4. VIG Algorithm for BHFSP

Ruiz and Stützle first proposed an IG algorithm for solving the FSP [9]. The framework of the basic IG is shown in Algorithm 1 and consists of four phases: initialization phase, destruction and reconstruction (DR) phase, local search, and acceptance criteria. First, a high-quality solution is obtained during the initialization phase. Second, for the destruction operation, we select some elements at random from the current solution π and delete them. All the deleted elements are put into π_d, and the rest elements forms π_r. For the reconstruction operation, each element is removed one by one from π_d and greedily reinserted into π_r until a complete solution is obtained. Next, a local reinforcement approach is adopted to further enhance the algorithm's performance. Finally, the solution is updated by using the acceptance criteria.

Algorithm 1. Basic IG algorithm

01: Generate an initial solution π_0
02: $\boldsymbol{\pi}$ = Local Search (π_0)
03: **while** (not meet termination criteria) **do**
04: (π_d, π_r) = *Destruction* (π)
05: π' = *Reconstruction* (π_d, π_r)
06: π'' = *Local Search* (π')
07: π = *AcceptanceCriterion* (π'', π_0)
08: **end while**

For BHFSP, two questions must be considered simultaneously: how to assign a job to a particular machine at each stage and how to determine the sequence of jobs to be processed at each machine. For the basic IG, only one solution is continually optimized during the iteration process, which reduces the diversity of solution. Therefore, considering the characteristics of BHFSP and the flaw of the basic IG, we proposed a variant IG (VIG). In the proposed VIG, two decoding strategies, i.e., forward decoding, and backward decoding, are devised to solve the machine allocation issue. To enhance the diversity of VIG, we devise a parallel mechanism based on a hybrid decoding strategy. This allows us to obtain

two optimized solutions, and their interaction is carried out by crossover operator. The above parallel and cooperation mechanism can increase the opportunity of obtaining a good solution to some extent.

Algorithm 2 describes the proposed VIG algorithm framework. First, two initial solutions are generated by using NEH [17] and MME [18] (see line 1 in Algorithm 2). Then, within the while loop, the DR_LS_SA strategy (see lines 6–7 in Algorithm 2) leads to the parallel evolution of the two solutions. As can be seen from the name of the DR_LS_SA function, this part includes DR, local search, and SA strategies. Algorithm 3 describes these three parts in detail. The DR strategy (see line 1 in Algorithm 3) and local search strategy (see line 5 in Algorithm 3) based on two decoding methods are proposed to improve the quality of the solutions and increase their diversity. If only the improved solution is accepted, the algorithm may converge prematurely and lose diversity. The diversity of solutions is enhanced in our algorithms by utilizing SA. If the quality of the currently generated solution is worse than the quality of the original solution, this research still chooses to accept the current solution with a certain probability (see line 12 in Algorithm 3), instead of directly discarding it, where $T = \sum_{j=1}^{J} \sum_{S=1}^{S} p_{j,s}/(10 \times J \times S) \times \tau$, where J is the number of jobs, S is the number of stages, $p_{j,s}$ is the processing time of job j at stage s, and τ ($_{\tau \in}$(0,1)) is a temperature parameter to be adjusted. To avoid the VIG from being trapped in a local optimum, we use a cooperation mechanism based on crossover operator to interact with two solutions, which further enhances the diversity of solutions (see lines 14–17 in Algorithm 2). The parameter *ward* in Algorithm 3 has two values, 0 and 1, which represent forward decoding and backward decoding, respectively (see lines 6–7 in Algorithm 2). In Algorithm 2, the number of times the best solution has not been improved is represented by parameter $fail\_count$. The parameter *crossover_threshold* represents the value of the condition threshold. When $fail\_count$ is equal to *crossover_threshold*, VIG implements a collaborative mechanism based on crossover operation to increase the diversity of the solution. To help readers better understand, Figure 2 illustrates the specific process of VIG, which includes DR, local search, and cooperation mechanisms.

Algorithm 2. The framework of the VIG algorithm

Input: π_1 and π_2 are two empty scheduling sequences, d is the number of removed and reinserted jobs in DR
01: $\pi_1 = NEH(\pi_1)$, $\pi_2 = MME(\pi_2)$
02: $\pi^{best1} = \pi_1$, $\pi^{temp1} = \pi_1$, $\pi^{best2} = \pi_2$, $\pi^{temp2} = \pi_2$
03: $fail\_count = 0$ %%$fail\_count$ is a counter
04: $\pi^{best} = \min(\pi^{best1}, \pi^{best2})$
05: **while** (termination criterion is not satisfied) **do**
06: $\pi^{temp1}, \pi^{best1}, \pi_1 = DR\_LS\_SA(\pi^{temp1}, \pi^{best1}, \pi_1, d, 0)$ %%including DR, local search, and SA
07: $\pi^{temp2}, \pi^{best2}, \pi_2 = DR\_LS\_SA(\pi^{temp2}, \pi^{best2}, \pi_2, d, 1)$
08: **if** (π^{best1} or π^{best2} is better than π^{best})
09: $\pi^{best} = \min(\pi^{best1}, \pi^{best2})$
10: $fail\_count = 0$
11: **else**
12: $fail\_count$++
13: **end if**
14: **if** ($fail\_count = crossover\_threshold$)
15: $\pi^{temp1}, \pi^{temp2} = Crossover(\pi^{temp1}, \pi^{temp2})$ %% Cooperative mechanism
16: $fail\_count = 0$
17: **end if**
18: **end while**
Output: π^{best}

Algorithm 3. *DR_LS_SA* (π^{temp}, π^{best}, π, d, $ward$)

Input: π^{temp}, π^{best}, π, d, $ward$
$\pi^{temp} = Destruction\_Reconstruction(\pi^{temp}, d, ward)$
if (π^{temp} is better than π^{best})
    $\pi^{best} = \pi^{temp}$
end if
$\pi^{temp} = LocalSearchStrategy(\pi^{temp}, ward)$ %% based on swap strategy
if (π^{temp} is better than π)
    $\pi = \pi^{temp}$
    if (π^{temp} is better than π^{best})
        $\pi^{best} = \pi^{temp}$
    end if
else %% SA acceptance criterion
    if ($rand() \leq \exp(-(C_{max}(\pi^{temp}) - C_{max}(\pi))/T)$)
        $\pi = \pi^{temp}$
    else
        $\pi^{temp} = \pi$
    end if
end if
Output: π^{temp}, π^{best}, π

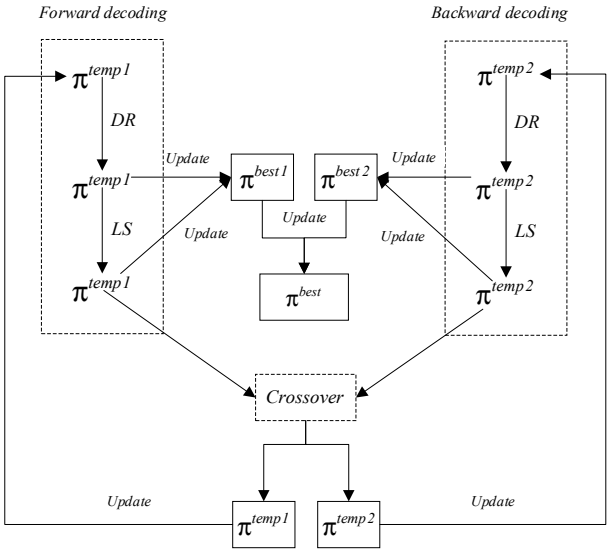

Figure 2. The cooperative process of two solutions.

4.1. Encoding and Decoding Scheme

There are two key issues that need to be addressed before designing an algorithm for BHFSP: encoding and decoding. Given the discrete nature of the problem, the job sequence corresponding to a solution is represented using a discrete integer encoding. For the decoding issue, its purpose is to obtain the real scheduling and evaluate the makespan. The regular FSSPs are known to be reversible. That is if the jobs go through the flow shop in the opposite direction and in the reverse process, the makespan does not change [24]. However, for BHFSP, due to there being parallel machines in some stages, it does not have the above property. Thus, different makespan values can be obtained for a given job sequence using forward and backward decoding methods. By different decoding strategies, we can select the minimal objective function value as the final makespan. Multiple decoding strategies can assist the algorithm to find the optimal value. Based on this, we propose a hybrid decoding strategy including forward and backward decoding. Suppose there are J jobs, and each job goes through S stages for processing. The processing sequence is [1, 2, . . . , J]. [j] represents the index of the jth job. In forward decoding, $fc_{[j],s}$ represents the completion

time of the job and $fmt_{s,m}$ is denoted as the idle time of the machine m at stage s. m^* represents the machine with the earliest idle time at stage s. $fc_{[j],s}$ is calculated according to Equations (10)–(14). Equation (15) represents the makespan of the forward calculation.

$$fc_{[j],0} = 0,\ j = 1,\ 2,\ \ldots, J \tag{10}$$

$$fmt_{s,m} = 0, s = 1, 2, \cdots, S, m = 1, 2, \cdots, M_s \tag{11}$$

$$m^* = \underset{m=1,2,\cdots,M_s}{argmin} \{fmt_{s,m}\}, s = 1, 2, \cdots, S \tag{12}$$

$$fc_{[j],s} = max\left\{fmt_{s,m^*}, fc_{[j],s-1}\right\} + p_{j,s}, j = 1, 2, \cdots, J, s = 1, 2, \cdots, S \tag{13}$$

$$fmt_{s,m^*} = \begin{cases} fc_{[j],s+1} - p_{[j],s+1}, s = 1, 2, \cdots, S-1 \\ fc_{[j],s}, s = S \end{cases} \tag{14}$$

$$C_{max} = maxf_{[j],s}, j = 1, 2, \cdots, J, s = S \tag{15}$$

In the backward decoding, $bc_{[j],s}$ represents the backward completion time of the job and bmt_{sm} is denoted as idle time of the machine m at stage s during the backward calculation. m^* represents the machine with the earliest idle time at stage s. $bc_{[j],s}$ is calculated according to Equations (16)–(20). Equation (21) is of the makespan obtained by the backward calculation.

$$bc_{[j],S+1} = 0, j = J, J-1, \cdots, 1 \tag{16}$$

$$bmt_{s,m} = 0, s = S, S-1, \cdots, 1, m = M_s, M_s - 1, \cdots, 1 \tag{17}$$

$$m^* = \underset{m=M_{s,M,-1},\cdots,1}{argmin} \{bmt_{s,m}\}, s = S, S-1, \cdots, 1 \tag{18}$$

$$bc_{[j],s} = max\left\{bmt_{s,m^*}, bc_{[j],s+1}\right\} + p_{j,s}, j = J, J-1, \cdots, 1, s = S, S-1, \cdots, 1 \tag{19}$$

$$bmt_{s,m^*} = \begin{cases} bc_{[j],s-1} - p_{[j],s-1}, s = S, S-1, \cdots, 2 \\ bc_{[j],s}, s = 1 \end{cases} \tag{20}$$

$$C_{max} = maxbc_{[j],s}, j = 1, 2, \cdots, J, s = 1 \tag{21}$$

In the following, we illustrate the forward and backward decoding methods by an example in which there are 3 stages and 4 jobs. There are two identical parallel machines at stage 1 and stage 2, and one machine at stage 3. Suppose that a particular job sequence π = (1, 2, 3, 4), and the processing time of each job at each stage is as follows.

$$p_{j,s} = \begin{bmatrix} 2 & 2 & 1 \\ 1 & 1 & 2 \\ 2 & 1 & 2 \\ 4 & 5 & 1 \end{bmatrix}$$

4.1.1. Forward Decoding Method

Forward decoding means arranging the first job in the sequence to each stage, then the second job, and so forth, until the last job. At each stage, the jobs are scheduled to the first earliest available machine. After a job is processed on the current machine, if no machine

is available at the next stage, the job is blocked on the current machine until a machine is available at the next stage. For a given job sequence $\pi = (1, 2, 3, 4)$ in the above example, we decode it according to the forward encoding method, and the procedure is as follows:

(1) First schedule job 1. Job 1 is arranged to stage 1, stage 2, and stage 3 according to the process sequence. As described above, at each stage we select the first earliest available machine for job 1. Figure 3a shows the Gantt chart after arranging job 1.
(2) Then job 2 is arranged to each stage according to the scheduling process of job 1. Note that on the second machine at the second stage, job 2 is blocked on the current machine until moment 4. Figure 3b shows the Gantt chart after job 2 is scheduled.
(3) Job 3 and job 4 are scheduled by using the same method as job 1 and job 2, the Gantt charts that job 3 and job 4 have been scheduled are shown in Figure 3c,d, respectively. We finally obtain real scheduling with a makespan that is equal to 12.

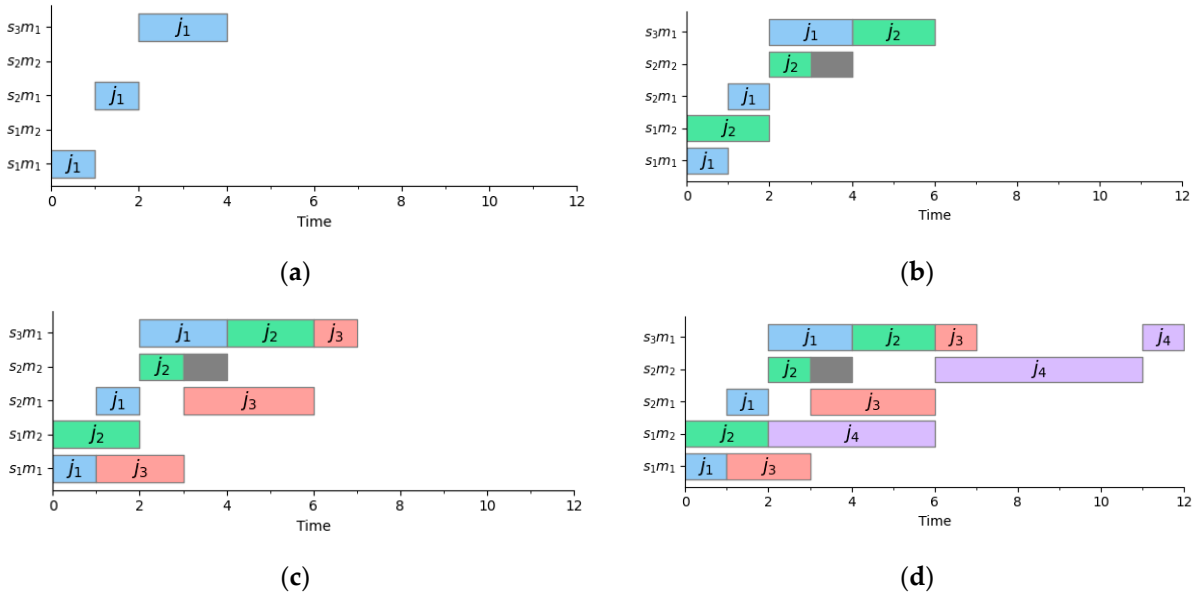

Figure 3. Forward decoding process for job sequence (the gray area in the graph is the blocking time). (**a**) Decoding process of job 1; (**b**) Decoding process of job 1 and job 2; (**c**) Decoding process of job 1, job 2, and job 3; (**d**) Decoding process of job 1, job 2, job 3, and job 4.

4.1.2. Backward Decoding Method

For a given sequence of jobs, real scheduling can also be obtained by traversing the jobs in the opposite order, that is, the last job is considered first. The jobs are assigned to the first available machine in a backward manner from the last stage to the first stage. Considering the above example, we arrange the jobs by the backward strategy as follows.

(1) First arrange job 4. Job 4 is arranged to stage 3, stage 2, and stage 1 according to the opposite process sequence. At each stage, we select the earliest available machine for job 4 in a backward manner. Figure 4a shows the Gantt chart for arranging job 4 by using the backward decoding method.
(2) Then consider job 3. In the same way as job 4 is scheduled, job 2 is arranged to each stage according to the opposite process sequence. Figure 4b shows the Gantt chart that job 2 is scheduled by using the backward decoding method.
(3) Job 2 and job 1 are scheduled by using the same method as job 4 and job 3. Note that blocking may also occur during the backward decoding process. Figure 4c,d give the Gantt charts that job 2 and job 1 have been scheduled by using the backward decoding method, respectively. Finally, we obtain real scheduling whose makespan is equal to 10.

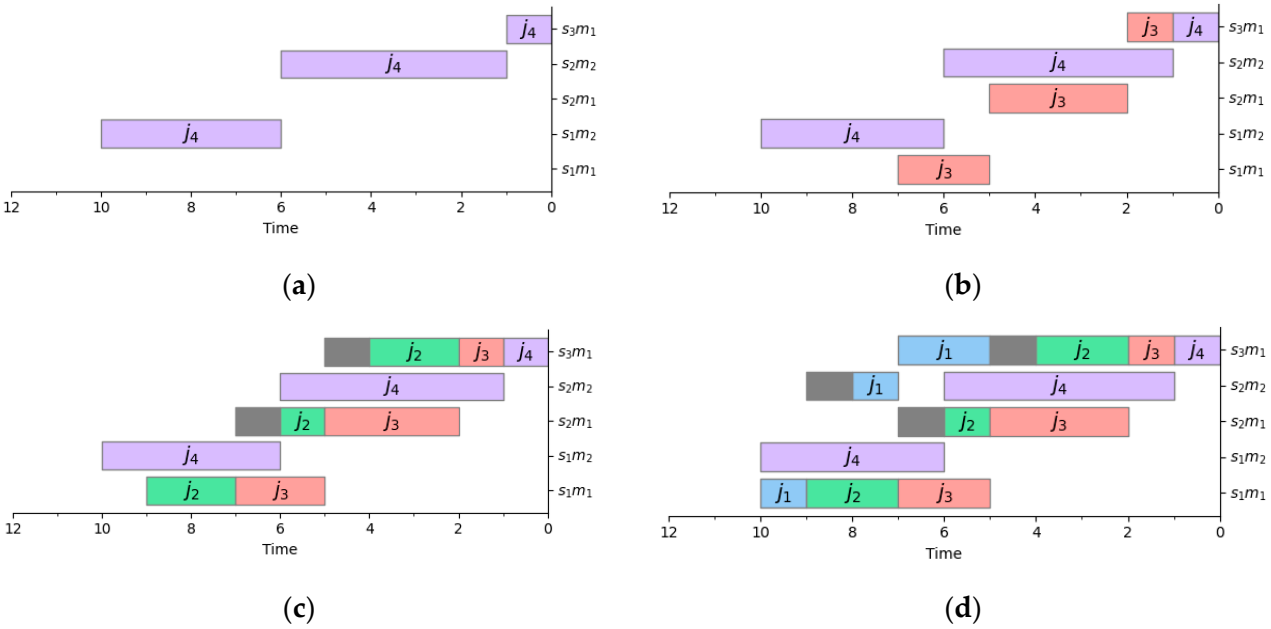

Figure 4. Backward decoding process for job sequence (the gray area in the graph is the blocking time). (**a**) Decoding process of job 1; (**b**) Decoding process of job 1 and job 2; (**c**) Decoding process of job 1, job 2, and job 3; (**d**) Decoding process of job 1, job 2, job 3, and job 4.

4.2. The Initialization Strategy

The performance of the IG algorithm is strongly impacted by the initialization strategy. Many scholars have proven that NEH is an efficient initialization method for discrete scheduling problems [17]. MME has a good performance in solving the BFSP [18]. Considering the blocking characteristic of BHFSP, both of the above two heuristics are used in this paper as the initialization strategies to solve BHFSP to improve the diversity of VIG.

In our paper, we attempt to enhance the diversity of VIG by generating multiple solutions. Therefore, we use different initialization strategies to solve BHFSP: NEH with a forward decoding strategy to calculate the makespan of job sequences and MME with a backward decoding strategy to calculate the makespan. At this time, we can guarantee that two high-quality initial solutions are obtained. In the next loop, we can optimize the two initial solutions in different directions, which may broaden the search space and increase the diversity of solutions.

4.3. Destruction and Construction

In traditional IG algorithms, the destruction and reconstruction operators can strongly disturb the current solution, which increases the diversity of solution. In the destruction phase, we randomly delete d jobs from π^{temp} and put them into π^{remove}. The remaining jobs compose of π^{temp} ($\pi^{temp} = \pi^{temp} \backslash \pi^{remove}$) (see lines 2–6). In the reconstruction phase, we extracted job π_j^{remove} from π^{remove}, one by one, and then try to insert them into all positions of π^{temp}. We adopt forward and backward decoding strategies to calculate the makespan, respectively. If $ward = 0$, forward decoding strategy is adopted to evaluate the objective function. If $ward = 1$, backward decoding strategy is adopted (see lines 11–15). Repeated the above process until all the jobs in sequence π^{remove} have been removed, and find the position with minimal $C_{\max}$. Algorithm 4 shows the pseudocode of the destruction and reconstruction strategy, $Destruction\_Reconstruction(\pi^{temp}, d, ward)$, which d is the number of jobs to be removed from π^{temp}, and $ward$ is the parameter that decides the decoding strategy.

Algorithm 4. *Destruction_Reconstruction*$(\pi^{temp},\ d,\ ward)$

Input: π^{temp}, parameter d, $ward$
01: $\pi^{remove} = \varnothing$, $\pi^{temp'} = \pi^{temp}$
02: **while** ($\pi^{remove}.size < d$) %% *Destruction*
03: $pt = rand()\%J + 1$
04: Extract $\pi_{pt}^{temp'}$ from $\pi^{temp'}$ and put it into π^{remove}
05: $\pi^{temp'} = \pi^{temp'} \backslash \pi_{pt}^{temp'}$
06: **end while**
07: **for** j =1 **to** d %% *reconstruction*
08: pos = 0
09: **while** ($pos \leq \pi^{temp}.size$)
10: Insert π_j^{remove} into position pos of $\pi^{temp'}$
11: **if** ($ward$ = 0)
12: Calculate $C_{\max}$ using forward decoding strategy
13: **else if** ($ward$ = 1)
14: Calculate $C_{\max}$ using backward decoding strategy
15: **end if**
16: pos++
17: **end while**
18: Select the best position pos with minimal $C_{\max}$ and insert π_j^{remove} to position pos of $\pi^{temp'}$
19: **end for**
Output: $\pi^{temp'}$

4.4. Local Search Strategy

Following the destruction and reconstruction phases, it is common to perform a local search operation on the solution to improve the quality of the current solution. The local search strategy in the traditional IG algorithm usually uses the insert operator. Assuming that there are J jobs, each job needs to be inserted into J positions, so the time complexity of using the insertion operator is $O(J^2)$. However, by replacing the insert operator with the swap operator, the time complexity can be reduced to $O(J)$. To reduce time complexity and explore local neighborhoods more deeply, we propose a swap-based local reinforcement strategy.

In Algorithm 5, function $swap(\pi_k^{temp}, \pi_{pt}^{temp})$ refers to exchange job π_k^{temp} and π_{pt}^{temp}, and obtain a new sequence $\pi^{temp'}$. $Forward\_Fit(\pi^{temp'})$ in line 6 means that the $\pi^{temp'}$ adopts the forward decoding strategy to calculate the makespan, $C_{\max'}$. Similarly, $Backward\_Fit(\pi^{temp'})$ in line 8 represents that the $\pi^{temp'}$ uses backward decoding to calculate $C_{\max'}$. If the makespan of $\pi^{temp'}$, $C_{\max'}$, is less than the makespan of π^{temp}, $C_{\max}$, then π^{temp} is updated with $\pi^{temp'}$; otherwise, $\pi^{temp'}$ is updated with π^{temp}.

Algorithm 5. *LocalSearchStrategy*$(\pi^{temp}, ward)$

Input: π^{temp}, $C_{\max}$ is the target value of π^{temp}, $ward$
01: k = 0, $\pi^{temp'} = \pi^{temp}$
02: **while** ($k < \pi^{temp}.size$)
03: **for** $pt = k + 1$ **to** J
04: $\pi^{temp'} = swap(\pi_k^{temp}, \pi_{pt}^{temp})$
05: **if** ($ward$ = 0)
06: $C_{\max'} = Forward\_Fit(\pi^{temp'})$
07: **else if** ($ward$ = 1)
08: $C_{\max'} = Backward\_Fit(\pi^{temp'})$
09: **end if**
10: **if** ($C_{\max'} < C_{\max}$)
11: $\pi^{temp} = \pi^{temp'}$

Algorithm 5. *Cont.*

```
        else
            π^temp' = π^temp
        end if
    end for
    k++
end while
```
Output: $\pi^{temp'}$

5. Simulation Experiments and Analysis

All the algorithms are coded in Visual Studio 2019 using C++. The same library functions are employed to make fair comparisons. All the instances are executed five independent replications on the same PC with an Intel Core i7 Pentium processor @ 3.60 GHz and 32 GB RAM, whose operating system is Windows 10 X64. For all the algorithms, an elapsed CPU time is adopted as the termination criterion.

5.1. Test Data and Performance Metric

To demonstrate the effectiveness of the proposed algorithm, we have conducted extensive simulation experiments. We use the methods of generating test instances provided by the literature [49] to obtain the test data. That is, the processing time of each job on each machine is obtained randomly from the interval [1, 99]. The number of machines at each stage is randomly generated from the range [1,5]. J belongs to {20, 40, 60, 80, 100}, and S belongs to {5, 10}. This will result in 10 ($|J|\times|S|$) different combinations, in which each combination has 10 test instances. Thus, 100 test instances are generated. To ensure fairness, each instance is independently run 30 times, and all algorithms set the same CPU runtime as the termination criteria, denoted as $TimeLimit = J \times S \times CPU$ (millisecond), in which CPU is set as 10 and 15, respectively. The source code in this paper can be found on GitHub (https://github.com/nideluckily/VIG.git (accessed on 17 May 2023)).

All performance comparisons are conducted using the relative percentage increase (RPI) as shown in Equation (22).

$$RPI = \frac{1}{10}\sum_{instance=1}^{10}\frac{c_i - c_{min}}{c_{min}} \times 100 \tag{22}$$

where, for each test instance, c_i is the average of makespan obtained by the i^{th} algorithm that independently runs 30 times. c_{min} is the best makespan found by all seven algorithms that independently run 30 times. The average RPI (ARPI) of 10 different combinations (scales) is calculated.

5.2. Validation of MILP Model

The MILP model mentioned in Section 2 will be evaluated and nine small-scale instances are randomly selected to demonstrate its correctness by using a Gurobi solver [38]. The maximum termination criteria of the Gurobi solver and VIG algorithm are set to 3600 s and $TimeLimit = J \times S \times CPU$ (millisecond), respectively. Each instance of VIG algorithm runs independently 30 times, and the average value of the makespan of 20 times is obtained as the final objective value. Table 3 shows the makespan and runtime obtained by the MILP model and VIG.

In Table 3, the MILP model performs better for small test instances, such as 8_2, 8_3, 8_4, 13_3, and 18_2 instances. However, as instance sizes increase, the performance of MILP gradually decreases compared to VIG, such as 13_4, 18_3, and 18_4 test instances. Table 3 also reveals that the time taken for MILP to solve the problem is much higher than that of VIG. Furthermore, for large-scale instances, the MILP model is difficult to provide a good solution in a short time. In practical production, it is necessary to find an approximate

optimal solution within a reasonable time. In contrast, the proposed VIG is more suitable for practical applications.

Table 3. Makespan values for the MILP Model and the VIG algorithm.

J_S	MILP		VIG	
	Makespan	**Time (s)**	**Makespan**	**Time (s)**
8_2	**74**	2.28	**74**	0.17
8_3	**154**	1.13	158	0.25
8_4	**185**	5.66	186	0.33
13_2	**194**	935.10	196	0.27
13_3	**193**	3600	196	0.39
13_4	247	3600	**241**	0.53
18_2	**243**	3600	**243**	0.38
18_3	298	3600	**296**	0.55
18_4	341	3600	**328**	0.73

Best values are indicated in bold.

5.3. Calibration of Parameters

In the IG algorithm, different parameter configurations may have an impact on the final results. In this subsection, we have calibrated two key parameters, d and *crossover_threshold*, using Taguchi experimental method. For parameter d, it plays an important role in balancing the local and global search abilities of an algorithm and coordinating its diversity. A too large d may increase the perturbation for the current sequence and devote more time. However, a too small d will result in a small perturbation and potentially insufficient exploration of the solution. For parameter *crossover_threshold*, it is used to interact with two solutions obtained by DR and local search strategies. The cooperation strategy based on crossover operator can increase the diversity with a certain probability. In other words, if the number of times that the makespan is not minimized reaches the threshold *crossover_threshold*, the algorithm may converge prematurely. Thus, the value of *crossover_threshold* is related to the scale of test instances. In this paper, the calculation formula of *crossover_threshold* is α/J, where α = {1600, 1800, 2000, 2200, 2400}. Thus, the calibration of *crossover_threshold* in this paper is the calibration of α.

In the calibration of parameters, preliminary experiments are conducted under the terminal condition $TimeLimit = J \times S \times CPU$ (millisecond). Each parameter is set to have five factor levels, such as d = {2, 3, 4, 5, 6} and α = {1600, 1800, 2000, 2200, 2400}, respectively. We obtained 25 different configurations using the orthogonal table shown in Table 4. To ensure fair, four different instances, such as 20 × 5, 60 × 10, 80 × 5, and 100 × 10, are randomly selected. The mean RPI values of different combinations are integrated. Based on the yielded RPI, the trend of the level of the factor is plotted in Figure 5.

Figure 5 shows the Taguchi analysis of the ARPI for each parameter. ARPI value is the smallest when d = 3, suggesting that VIG has the best performance with this parameter setting. Then as d becomes larger, the value of RPI gradually becomes larger as well. This is because as the value of d increasing, the number of the reinsertion operator used in the DR strategy also increases, which takes more time. As a result, the number of iterations for VIG is reduced, and the algorithm loses opportunities to search for potential solutions. Parameter α has little influence on VIG. When α = 2200, VIG demonstrates dominance. Based on the above experimental analysis, we can conclude that d has a significant impact on VIG, while α has only a minor impact. Therefore, we set the parameters as d = 3 and α = 2200.

Table 5 reports the significance level of parameter d and α in terms of the ARPI value of all scale instances, where delta measures the magnitude of the effect by calculating the difference between the maximum and minimum ARPI values among the five levels. Delta is larger or the rank indicator is smaller, suggesting that the parameter has an obvious influence. Thus, from Table 5, d and α have great and small impacts on the algorithm, respectively.

Table 4. Orthogonal table and response values.

Combination	Parameters		ARPI
	d	α	
1	2	1600	0.747758
2	2	1800	0.720745
3	2	2000	0.729445
4	2	2200	0.739731
5	2	2400	0.697866
6	3	1600	0.551798
7	3	1800	0.556757
8	3	2000	0.555334
9	3	2200	0.582645
10	3	2400	0.567974
11	4	1600	0.60351
12	4	1800	0.587508
13	4	2000	0.581717
14	4	2200	0.524173
15	4	2400	0.57228
16	5	1600	0.631847
17	5	1800	0.641343
18	5	2000	0.655224
19	5	2200	0.64794
20	5	2400	0.639212
21	6	1600	0.734407
22	6	1800	0.691219
23	6	2000	0.719173
24	6	2200	0.669324
25	6	2400	0.696156

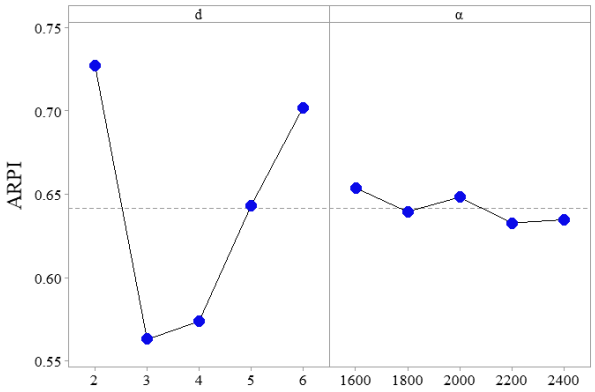

Figure 5. Taguchi analysis for all the factors for the VIG calibration.

Table 5. The ARPI response values of each parameter.

Level	d	α
1	0.7271	0.6539
2	0.5629	0.6395
3	0.5738	0.6482
4	0.6431	0.6328
5	0.7021	0.6347
Delta	0.1642	0.0211
Rank	1	2

5.4. Evaluating Different Strategies

This section will evaluate and analyze the algorithm components. Two sets of experiments, including comparisons of different decoding strategies and comparisons of local search strategies, are performed to verify their effectiveness.

As mentioned in Section 4, we know that for a given sequence of jobs, the objective values calculated by forward decoding and backward decoding, respectively, may be different [24]. To obtain the minimal makespan, we adopted a hybrid decoding strategy, which combines forward decoding and backward decoding. To verify the superiority of our hybrid decoding strategy, we tested the performance of VIG with only backward decoding, VIG with only forward decoding, and VIG with hybrid decoding. The three algorithms were run under the same experimental environment. For each compared strategy, we adopted 100 test instances, each one ran independently 30 times, and the ARPI values are shown in Figure 6. In Figure 6, the hybrid decoding strategy combining forward and backward is superior to the single decoding strategy. The reason may be that different decoding strategies can obtain different makespans, and multiple decoding strategies provide more opportunities to obtain minimal makespans than single decoding strategy. Therefore, the hybrid decoding strategy is suited for solving BHFSP.

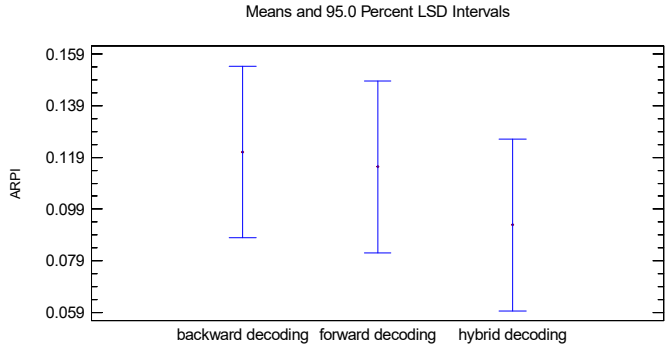

Figure 6. Means plots for different decoding strategies.

In traditional IG algorithms, the insertion operator is often considered to disturb the current solution. Local search using the insertion operator typically takes more time than the one based on the swap operator. Therefore, we adopted a swap-based local search strategy instead of the insertion operator. To verify its effectiveness, the strategies with the insertion operator and with the swap operator, respectively, have been conducted under the terminal condition $J \times S \times 10$ and ran five times independently on 100 instances. From Figure 7, we observed that the swap-based operator is obviously more effective than the insertion operator. The reason may be that the time complexity of the former is lower than that of the latter. Thus, in the same terminal time, the number of iterations of the algorithm using a swap operator is much more than that of an insertion operator, resulting that VIG is enabled to obtain more possible approximate solutions or explore more neighborhoods. Therefore, the swap-based local search can enhance the performance of the algorithm, which can effectively explore undiscovered neighborhoods and search for more solutions with high quality.

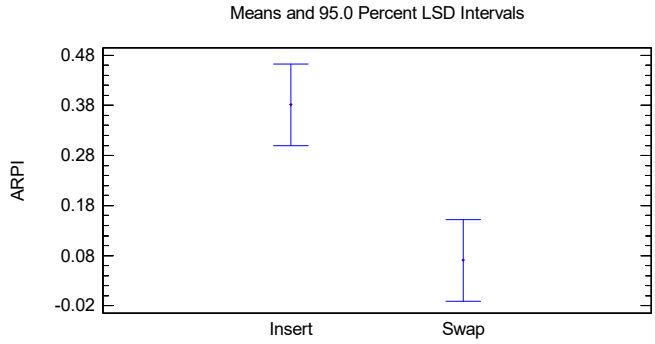

Figure 7. Means plots for different local search strategies.

5.5. Performance Evaluation of Comparative Algorithms

To analyze the capability of VIG and prove its effectiveness, we compared VIG with six algorithms in this section. The comparison algorithms include four population-based intelligence optimization methods, such as GA [51], DABC [24], EMBO [29], and discrete particle swarm optimization algorithm (DPSO) [52], and two improved IG algorithms, i.e., IGTALL [2] and double level mutation iterated greedy (IGDLM) [49]. GA is a meta-heuristic algorithm that is influenced by natural selection and generates new candidate solutions by using operators such as crossover and mutation. DABC is a classical swarm intelligence algorithm that uses heuristics to generate the initial population and then optimizes the solution through three stages: employed bees, onlooker bees, and scout bees. EMBO uses two competitive mechanisms to improve the probability of finding a better solution at the front end of the flock. DPSO combines the variable neighborhood search (VNS) algorithm with the particle swarm optimization algorithm to reduce the computation time and obtain the optimal solution. IGTALL adds a local search strategy after the destruction phase to improve the quality of some solutions and further expand the ability of the algorithm to explore the solution space. IGDLD reduces the computation time and helps the algorithm explore the solution space more deeply by using two mutation operations. The above six comparison algorithms are all proposed for HFSP and have been proven to have excellent performance. Thus, it is equitable to consider them as contrastive algorithms. In addition, to be fair, we modified them appropriately to adapt our problem. To be fair, the parameters of all compared algorithms were set to the values recommended by the original literature and shown in Table 6. All algorithms were set up in the same experimental environment to run and use the same termination criterion $TimeLimit = J \times S \times CPU$ for 100 test instances. Each instance ran independently 30 times.

Table 6. Parameters of the comparison algorithms.

Algorithm	Population Size	Number of Destruction Jobs	Crossover Rate	Variation Rate	Constant Factor	Temperature Coefficient
	Psize	***d***	***Pc***	***Pr***	***α***	***T***
DABC	20	/	/	/	30	/
DPSO	100	/	/	/	200	/
EMBO	25	/	/	/	10	/
GA	100	/	0.7	0.1	4	0.85
IGDLM	/	4	/	0.3	/	0.5
IGTALL	/	3	/	/	10	0.5
VIG	/	3	/	/	/	0.5

Tables 7 and 8 give the experimental results, i.e., average makespan (AVG) and RPI values obtained by all algorithms on 10 different scales when $CPU = 10$ and $CPU = 15$, respectively. In Tables 7 and 8, '$J \times S$' represents the scale of the problem, where J and S refer to the number of jobs and stages, respectively. Meanwhile, to visually illustrate the advantages of all comparison algorithms, Figures 8 and 9 show the means and 95% least significant difference (LSD) confidence intervals of all test algorithms.

In Tables 7 and 8, the proposed VIG algorithm shows extremely superior performance in solving 10 instance sets of different scales. We can observe that the number of the best average makespan generated by VIG is 10, followed by DPSO (2), IGDLM (1), IGTALL (0), GA (0), EMBO (0), and DABC (0). Meanwhile, the RPI values obtained by VIG are the best among all the algorithms. Similarly, VIG obtains the best average makespan and RPI values for all the instances when CPU = 15. Overall, VIG outperforms the comparative algorithms in solving the BHFSP, which may be attributed to our hybrid decoding strategy that can efficiently identify and select a small makespan value. Additionally, the swap-based local search can explore undiscovered neighborhoods more effectively and potentially find more feasible solutions.

Table 7. The values of average makespan and RPI for all comparison algorithms when CPU = 10.

$J \times S$	VIG		IGTALL		IGDLM		GA		EMBO		DPSO		DABC	
	AVG	RPI	AVG	RPI	AVG	RPI	AVG	RPI	AVG	RPI	AVG	RPI	AVG	RPI
20 × 5	**794**	**0.08**	798	0.72	796	0.37	810	2.41	795	0.18	796	0.33	796	0.47
20 × 10	**1350**	**0.11**	1357	0.61	1351	0.20	1389	3.04	1352	0.27	**1350**	0.14	1355	0.52
40 × 5	**1363**	**0.31**	1386	2.36	1380	1.83	1403	3.82	1376	1.48	1393	2.99	1393	2.97
40 × 10	**2215**	**0.13**	2235	1.01	2224	0.52	2271	2.66	2229	0.74	2242	1.27	2255	1.86
60 × 5	**2761**	**0**	2762	0.03	**2761**	0.01	2770	0.35	2762	0.04	**2761**	0.02	2764	0.14
60 × 10	**3175**	**0.21**	3200	1.13	3192	0.87	3221	1.80	3191	0.81	3225	2.00	3231	2.09
80 × 5	**3400**	**0.16**	3416	1.11	3408	0.63	3427	1.50	3409	0.61	3430	1.89	3435	1.90
80 × 10	**4272**	**0.05**	4286	0.36	4280	0.22	4321	1.17	4283	0.30	4332	1.39	4353	1.89
100 × 5	**3904**	**0.08**	3917	0.88	3911	0.50	3925	1.20	3913	0.58	3932	1.67	3934	1.69
100 × 10	**5490**	**0.15**	5526	0.81	5516	0.62	5549	1.22	5504	0.41	5667	3.37	5714	4.24
mean	**2872.4**	**0.13**	2888.3	0.92	2881.9	0.60	2908.6	1.87	2881.4	0.58	2913	1.62	2923	1.91

Best values are indicated in bold.

Table 8. The values of average makespan and RPI for all comparison algorithms when CPU = 15.

$J \times S$	VIG		IGTALL		IGDLM		GA		EMBO		DPSO		DABC	
	AVG	RPI	AVG	RPI	AVG	RPI	AVG	RPI	AVG	RPI	AVG	RPI	AVG	RPI
20 × 5	**794**	**0.07**	798	0.67	796	0.34	811	2.62	**794**	0.16	795	0.31	797	0.53
20 × 10	**1349**	**0.08**	1356	0.60	1351	0.16	1387	2.90	1352	0.26	1350	0.10	1356	0.55
40 × 5	**1363**	**0.31**	1385	2.30	1380	1.84	1401	3.71	1376	1.43	1392	2.96	1393	2.98
40 × 10	**2214**	**0.11**	2233	0.93	2224	0.50	2274	2.79	2229	0.74	2240	1.21	2256	1.87
60 × 5	**2761**	**0.00**	2762	0.02	**2761**	0.00	2771	0.39	2762	0.03	**2761**	0.02	2764	0.12
60 × 10	**3175**	**0.17**	3198	1.07	3190	0.79	3224	1.86	3190	0.75	3224	1.94	3229	2.01
80 × 5	**3400**	**0.10**	3415	1.05	3408	0.63	3428	1.58	3409	0.60	3429	1.83	3434	1.83
80 × 10	**4271**	**0.02**	4285	0.35	4279	0.21	4317	1.08	4283	0.29	4330	1.33	4350	1.81
100 × 5	**3904**	**0.10**	3916	0.85	3910	0.49	3926	1.21	3913	0.59	3931	1.64	3934	1.68
100 × 10	**5489**	**0.12**	5525	0.79	5515	0.59	5553	1.29	5505	0.42	5662	3.28	5712	4.19
mean	**2872**	**0.11**	2887.3	0.86	2881.4	0.56	2909.2	1.94	2881.3	0.53	2911	1.46	2922.5	1.76

Best values are indicated in bold.

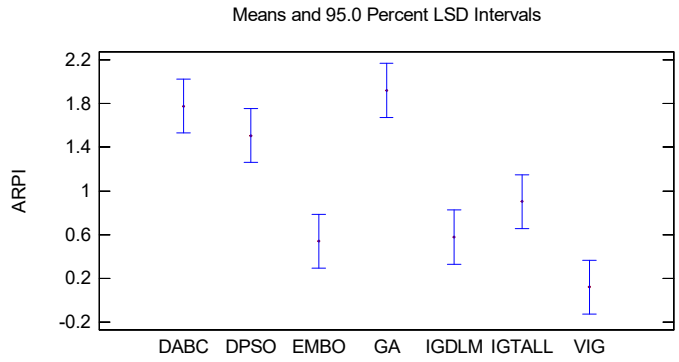

Figure 8. Confidence intervals for all comparison algorithms when CPU = 10.

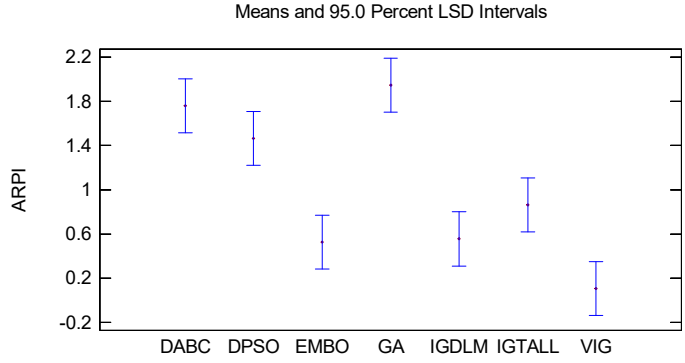

Figure 9. Confidence intervals for all comparison algorithms when CPU = 15.

To intuitively display the performance of all comparison algorithms, Figures 8 and 9 show the means and 95% LSD confidence intervals of all test algorithms. We can observe

that under the different CPU running times, the VIG algorithm outperforms other comparison algorithms, followed by IGDLM, EMBO and IGTALL, DPSO, DABC, and GA algorithms. We can believe that the algorithm including the proposed hybrid decoding, local search, and interaction strategies can show superior performance in solving BHFSP.

To further demonstrate the superiority of VIG, we selected four instance sets of different scales, i.e., 20 × 10, 60 × 10, 80 × 5, and 100 × 5, and displayed their box plots in Figure 10. The makespan of the VIG algorithm was smaller than those of IGDLM, EMBO, IGTALL, DPSO, DABC, and GA. In the four randomly selected test instances, VIG stood out from the other comparison algorithms and exhibited a relatively stable state. This indicates that the quality of the solutions produced by our algorithm does not fluctuate much and the convergence of the algorithm is good when dealing with instances of different sizes. From the overall perspective, VIG obtained the best makespan value.

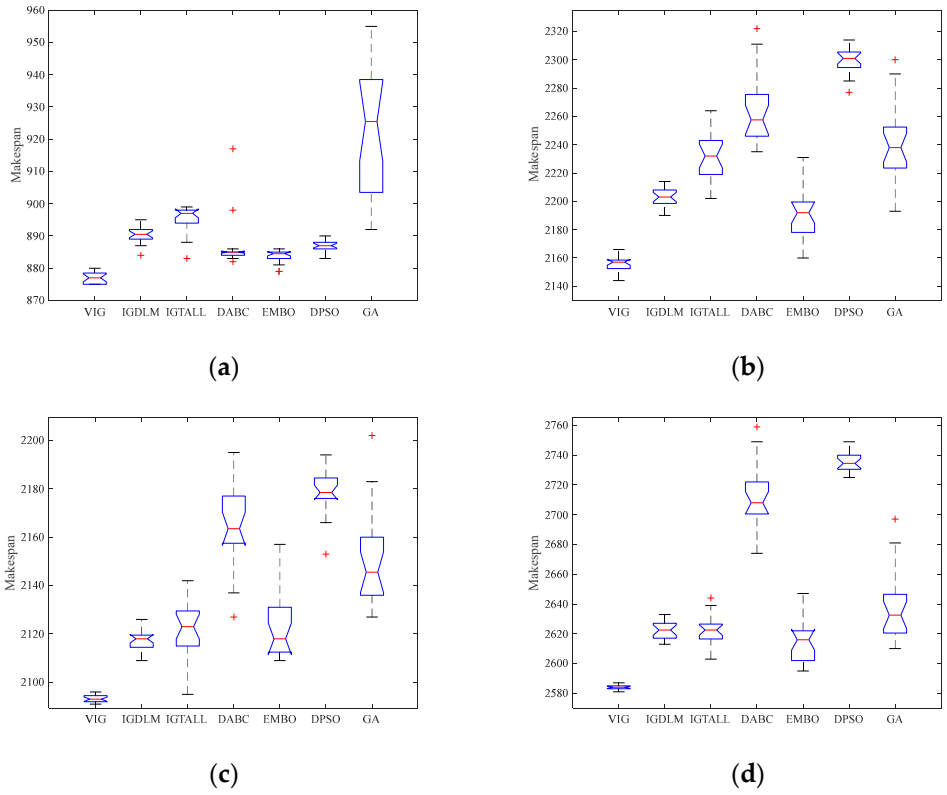

Figure 10. Box plot of all algorithms. (**a**) 20 × 10; (**b**) 60 × 10; (**c**) 80 × 5; (**d**) 100 × 5.

In addition, to enrich the experiments, we analyzed the convergence properties of the proposed algorithm. Four test instances, 40 × 5, 60 × 10, 80 × 5, and 100 × 10, were randomly selected to plot the evolutionary curves. All algorithms were performed in the same experimental environment, and the CPU is set to 10. In Figure 11, *Y*-axis is the makespan yielded by the algorithm during the evolutionary process, and the *X*-axis refers to the elapsing time of the algorithm. From Figure 11, it is clear that VIG has the lowest convergence curve and is the most rapid among the seven algorithms. VIG can use a hybrid decoding strategy to obtain different makespan values, increasing the diversity of solutions. At the same time, the use of local search can further increase the depth of neighborhood exploration and hopefully find better solutions. In addition, VIG can find better solutions for test instances of different scales and shows low convergence curves, which again demonstrates the good convergence of the algorithm. The above analysis demonstrates the effectiveness of VIG in solving the BHFSP.

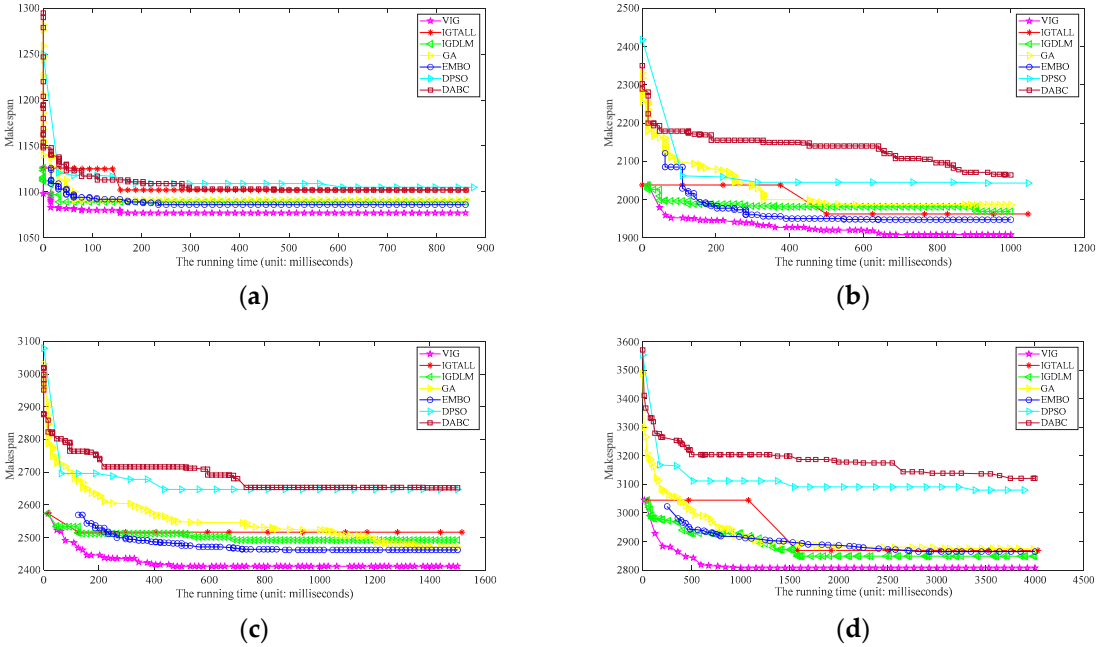

Figure 11. Evolution curve of all algorithms. (**a**) 40 × 5; (**b**) 60 × 10; (**c**) 80 × 5; (**d**) 100 × 10.

5.6. Friedman Test

These simulation results are used to test whether significant differences exist in multiple overall distributions [53]. We presume that all the compared methods are not significantly different from each other at every beginning. When the *p*-value is less than 0.05, the above assumption is unacceptable, and all methods are regarded to be significantly different. On the contrary, if the above assumption is accepted, it suggests that there are no significant differences between all methods. Tables 9 and 10 list the results of the Friedman test for 100 instances under CPU = 10 and CPU = 15, respectively. The *p*-value obtained by the Friedman test is equal to 0.000, which is less than the given level $\alpha = 0.050$, suggesting that VIG is significantly different from the six compared algorithms. For the values of ranks, the proposed VIG algorithm has the minimum rank value, e.g., 2.13 and 2.17 when CPU = 10 and CPU = 15, respectively. In addition, the maximum value ARPI (max), mean value ARPI (mean), and standard deviation (std. deviation) of VIG are 0.64, 0.061, and 0.1369, respectively. The above values are the smallest compared to other algorithms. Based on the above analysis, our VIG has an extremely high ability to solve BHFSP and is suitable for BHFSP.

Table 9. Friedman Test ($\alpha = 0.05$) when CPU = 10.

Algorithm	Ranks	CN	Min	Max	Mean	Std. Deviation
VIG	**2.13**	100	0.00	**0.64**	**0.061**	**0.1369**
IGTALL	3.84	100	0.00	5.88	0.827	1.4039
IGDLM	2.91	100	0.00	4.67	0.506	1.0183
GA	6.23	100	0.00	8.09	1.804	1.9808
EMBO	3.41	100	0.00	4.47	0.471	0.8354
DPSO	4.25	100	0.00	10.59	1.442	2.4597
DABC	5.26	100	0.00	10.18	1.692	2.4898
p-value	0.000					

Best values are indicated in bold.

Table 10. Friedman Test (α = 0.05) when CPU = 15.

Algorithm	Ranks	CN	Min	Max	Mean	Std. Deviation
VIG	**2.17**	100	0.00	**0.68**	**0.063**	**0.1366**
IGTALL	3.99	100	0.00	6.11	0.829	1.3963
IGDLM	2.95	100	0.00	4.72	0.512	1.0325
GA	6.23	100	0.00	7.92	1.764	1.9696
EMBO	3.23	100	0.00	4.11	0.460	0.8254
DPSO	4.26	100	0.00	10.22	1.416	2.4265
DABC	5.18	100	0.00	8.10	1.559	2.2845
p-value	0.000					

Best values are indicated in bold.

Remark 1. *Based on the experimental data and analysis reported in Sections 5.2–5.6, the proposed VIG algorithm has good convergence and is effective in solving BHFSP. The reasons are given as follows: (1) This paper adopts the parallel optimization route to obtain two solutions; one solution adopts forward decoding and the other adopts backward decoding, which can effectively calculate the objective value and select a minimal makespan. (2) Two efficient initialization methods based on different decoding strategies are used to generate high-quality solutions at the beginning, which can assist the algorithm to have more opportunities for probing the search space in depth. (3) The implementation of a swap-based local search strategy with low time complexity, which explores the neighborhood space of the solution very quickly. (4) The interaction of optimization solutions with some probability yields two new scheduling sequences, which improves the diversity of the solutions. Based on the analyses, the proposed hybrid decoding strategy and parallel optimization route are reasons for the outstanding performance of VIG. In addition, in real production, traditional manual scheduling methods often fail to obtain efficient solutions due to the complexity of scheduling problems. The intelligent optimization algorithm can find a solution close to the optimal solution within a reasonable time frame by the computing power and search technique of the computer. From the experimental results and analysis, our VIG algorithm is effective in solving the BHFSP, which can provide a promising solution for realistic production scheduling problems.*

6. Conclusions

We studied the BHFSP and now propose a VIG algorithm to optimize the makespan. First, a MILP model of BHFSP was built, and the Gurobi solver was adopted to demonstrate its correctness. Second, multiple decoding strategies based on forward and backward mechanisms were proposed to find a smaller completion time. Then a parallel evolution framework of two solutions was designed to broaden the search scope of VIG. In addition, the swap-based local disturbance strategy could quickly change the position of the job sequence, which saved the running time and increased the opportunities to probe the search space in depth, effectively enhancing the quality of the algorithm. Finally, the use of a crossover operator increased the diversity of solutions. Through the simulation experiments listed in Section 5, the proposed VIG algorithm shows superiority compared with the six existing algorithms.

This paper presents insights into the application of intelligent optimization algorithms in solving production scheduling problems. In practice, production scheduling involves the task of arranging production resources and job sequences in complex production environments to maximize production efficiency and meet production requirements. Due to the complexity of production scheduling problems and the diversity of constraints, traditional manual scheduling methods often fail to achieve optimal or efficient solutions. Intelligent optimization algorithms use the computing power and search techniques of computers to explore near-optimal solutions in large-scale problem spaces. The experimental results and analysis demonstrate the effectiveness of the proposed optimization approach. Decision makers can utilize our method to obtain high-quality solutions within a reasonable time frame, thereby improving production efficiency, reducing costs, enhancing competitiveness, and achieving sustainable development.

Although VIG demonstrates excellent performance in solving the BHFSP, there are still some related issues that require further research. Firstly, VIG performs well in solving the optimization objective of the maximum completion time, but it may not be suitable for other objectives or multi-objective scenarios. In future research, multi-objective optimization problems can be considered. Secondly, the production constraints currently considered are not comprehensive enough, and the next step of research can also consider multi-factory production, group problems, etc. Finally, considering the actual production demand, we should apply the proposed algorithm to real production scheduling problems.

Author Contributions: Y.W. (Yong Wang): conceptualization, methodology, data curation, software, validation, writing—original draft. Y.W. (Yuting Wang): conceptualization, methodology, software, validation, writing—original draft. Y.H.: conceptualization, methodology, software, validation, writing—original draft. All authors have read and agreed to the published version of the manuscript.

Funding: This work was jointly supported by the National Natural Science Foundation of China under grant numbers 61973203. We are grateful for Guangyue Youth Scholar Innovation Talent Program support received from Liaocheng University (LCUGYTD2022-03).

Data Availability Statement: The data that support the findings of this study are available from the corresponding author, upon reasonable request.

Conflicts of Interest: The authors declare that they have no conflict of interest.

Abbreviations

Notations:
J: The number of jobs.
S: The number of stages.
j: The index of jobs, $j \in \{1, 2, \ldots, J\}$.
s: The index of stages, $s \in \{1, 2, \ldots, S\}$.
M_s: The number of parallel machines at stage s.
m: The index of machines at stage s, $m \in \{1, 2, \ldots, M_s\}$.
U: A big positive number.
$p_{j,s}$ The processing time of job j at stage s.
Decision variables:
C_{max}: The makespan.
$C_{j,s}$: The completion time of job j at stage s.
$D_{j,s}$: The departure time of job j at stage s. The time that job j leaves the machine when it finishes processing at stage s.
$y_{j,s,m}$: Binary decision variable, 1 if job j is processed on machine m at stage s, 0 otherwise.
$z_{j,j',s}$: Binary decision variable, 1 if job j, is processed on the same machine before job j' at stage s, 0 otherwise.

References

1. Reza Hejazi, S.; Saghafian, S. Flowshop-Scheduling Problems with Makespan Criterion: A Review. *Int. J. Prod. Res.* **2005**, *43*, 2895–2929. [CrossRef]
2. Öztop, H.; Fatih Tasgetiren, M.; Eliiyi, D.T.; Pan, Q.-K. Metaheuristic Algorithms for the Hybrid Flowshop Scheduling Problem. *Comput. Oper. Res.* **2019**, *111*, 177–196. [CrossRef]
3. Kim, Y.-D.; Shim, S.-O.; Choi, B.; Hwang, H. Simplification Methods for Accelerating Simulation-Based Real-Time Scheduling in a Semiconductor Wafer Fabrication Facility. *IEEE Trans. Semicond. Manuf.* **2003**, *16*, 290–298.
4. Marichelvam, M.K.; Prabaharan, T.; Yang, X.S. A Discrete Firefly Algorithm for the Multi-Objective Hybrid Flowshop Scheduling Problems. *IEEE Trans. Evol. Comput.* **2014**, *18*, 301–305. [CrossRef]
5. Bruzzone, A.A.G.; Anghinolfi, D.; Paolucci, M.; Tonelli, F. Energy-Aware Scheduling for Improving Manufacturing Process Sustainability: A Mathematical Model for Flexible Flow Shops. *CIRP Ann.* **2012**, *61*, 459–462. [CrossRef]
6. Peng, K.; Pan, Q.-K.; Gao, L.; Zhang, B.; Pang, X. An Improved Artificial Bee Colony Algorithm for Real-World Hybrid Flowshop Rescheduling in Steelmaking-Refining-Continuous Casting Process. *Comput. Ind. Eng.* **2018**, *122*, 235–250. [CrossRef]
7. Fernandez-Viagas, V.; Ruiz, R.; Framinan, J.M. A New Vision of Approximate Methods for the Permutation Flowshop to Minimise Makespan: State-of-the-Art and Computational Evaluation. *Eur. J. Oper. Res.* **2017**, *257*, 707–721. [CrossRef]

8. Wardono, B.; Fathi, Y. A Tabu Search Algorithm for the Multi-Stage Parallel Machine Problem with Limited Buffer Capacities. *Eur. J. Oper. Res.* **2004**, *155*, 380–401. [CrossRef]
9. Ruiz, R.; Stützle, T. A Simple and Effective Iterated Greedy Algorithm for the Permutation Flowshop Scheduling Problem. *Eur. J. Oper. Res.* **2007**, *177*, 2033–2049. [CrossRef]
10. Ruiz, R.; Pan, Q.-K.; Naderi, B. Iterated Greedy Methods for the Distributed Permutation Flowshop Scheduling Problem. *Omega* **2019**, *83*, 213–222. [CrossRef]
11. Ribas, I.; Companys, R.; Tort-Martorell, X. An Iterated Greedy Algorithm for the Parallel Blocking Flow Shop Scheduling Problem and Sequence-Dependent Setup Times. *Expert Syst. Appl.* **2021**, *184*, 115535. [CrossRef]
12. Wang, S.; Liu, M.; Chu, C. A Branch-and-Bound Algorithm for Two-Stage No-Wait Hybrid Flow-Shop Scheduling. *Int. J. Prod. Res.* **2015**, *53*, 1143–1167. [CrossRef]
13. Riane, F.; Artiba, A.; Elmaghraby, S.E. Sequencing a Hybrid Two-Stage Flowshop with Dedicated Machines. *Int. J. Prod. Res.* **2002**, *40*, 4353–4380. [CrossRef]
14. Ruiz, R.; Vázquez-Rodríguez, J.A. The Hybrid Flow Shop Scheduling Problem. *Eur. J. Oper. Res.* **2010**, *205*, 24. [CrossRef]
15. Fattahi, P.; Hosseini, S.M.H.; Jolai, F.; Tavakkoli-Moghaddam, R. A Branch and Bound Algorithm for Hybrid Flow Shop Scheduling Problem with Setup Time and Assembly Operations. *Appl. Math. Model.* **2014**, *38*, 119–134. [CrossRef]
16. Xuan, H.; Tang, L. Scheduling a Hybrid Flowshop with Batch Production at the Last Stage. *Comput. Oper. Res.* **2007**, *34*, 2718–2733. [CrossRef]
17. Nawaz, M.; Enscore, E.E.; Ham, I. A Heuristic Algorithm for the M-Machine, n-Job Flow-Shop Sequencing Problem. *Omega* **1983**, *11*, 91–95. [CrossRef]
18. Ronconi, D.P. A Note on Constructive Heuristics for the Flowshop Problem with Blocking. *Int. J. Prod. Econ.* **2004**, *87*, 39–48. [CrossRef]
19. Pan, Q.-K.; Wang, L. Effective Heuristics for the Blocking Flowshop Scheduling Problem with Makespan Minimization. *Omega* **2012**, *40*, 218–229. [CrossRef]
20. Fernandez-Viagas, V.; Molina-Pariente, J.M.; Framinan, J.M. New Efficient Constructive Heuristics for the Hybrid Flowshop to Minimise Makespan: A Computational Evaluation of Heuristics. *Expert Syst. Appl.* **2018**, *114*, 345–356. [CrossRef]
21. Xiao, W.; Hao, P.; Zhang, S.; Xu, X. Hybrid Flow Shop Scheduling Using Genetic Algorithms. In Proceedings of the 3rd World Congress on Intelligent Control and Automation (Cat. No. 00EX393), Hefei, China, 28 June–2 July 2000; IEEE: New York, NY, USA, 2000; Volume 1, pp. 537–541.
22. Jin, Z.; Yang, Z.; Ito, T. Metaheuristic Algorithms for the Multistage Hybrid Flowshop Scheduling Problem. *Int. J. Prod. Econ.* **2006**, *100*, 322–334. [CrossRef]
23. Wang, S.; Wang, L.; Liu, M.; Xu, Y. An Enhanced Estimation of Distribution Algorithm for Solving Hybrid Flow-Shop Scheduling Problem with Identical Parallel Machines. *Int. J. Adv. Manuf. Technol.* **2013**, *68*, 2043–2056. [CrossRef]
24. Pan, Q.-K.; Wang, L.; Li, J.-Q.; Duan, J.-H. A Novel Discrete Artificial Bee Colony Algorithm for the Hybrid Flowshop Scheduling Problem with Makespan Minimisation. *Omega* **2014**, *45*, 42–56. [CrossRef]
25. Li, J.; Pan, Q.; Wang, F. A Hybrid Variable Neighborhood Search for Solving the Hybrid Flow Shop Scheduling Problem. *Appl. Soft Comput.* **2014**, *24*, 63–77. [CrossRef]
26. Lin, S.-W.; Cheng, C.-Y.; Pourhejazy, P.; Ying, K.-C.; Lee, C.-H. New Benchmark Algorithm for Hybrid Flowshop Scheduling with Identical Machines. *Expert Syst. Appl.* **2021**, *183*, 115422. [CrossRef]
27. Utama, D.M.; Primayesti, M.D. A Novel Hybrid Aquila Optimizer for Energy-Efficient Hybrid Flow Shop Scheduling. *Results Control. Optim.* **2022**, *9*, 100177. [CrossRef]
28. Utama, D.; Salima, A.; Setiya Widodo, D. A Novel Hybrid Archimedes Optimization Algorithm for Energy-Efficient Hybrid Flow Shop Scheduling. *Int. J. Adv. Intell. Inform.* **2022**, *8*, 237. [CrossRef]
29. Zhang, B.; Pan, Q.; Gao, L.; Zhang, X.; Sang, H.; Li, J. An Effective Modified Migrating Birds Optimization for Hybrid Flowshop Scheduling Problem with Lot Streaming. *Appl. Soft Comput.* **2017**, *52*, 14–27. [CrossRef]
30. Zhang, B.; Pan, Q.-K.; Meng, L.-L.; Zhang, X.-L.; Ren, Y.-P.; Li, J.-Q.; Jiang, X.-C. A Collaborative Variable Neighborhood Descent Algorithm for the Hybrid Flowshop Scheduling Problem with Consistent Sublots. *Appl. Soft Comput.* **2021**, *106*, 107305. [CrossRef]
31. Li, Y.-Z.; Pan, Q.-K.; Li, J.-Q.; Gao, L.; Tasgetiren, M.F. An Adaptive Iterated Greedy Algorithm for Distributed Mixed No-Idle Permutation Flowshop Scheduling Problems. *Swarm Evol. Comput.* **2021**, *63*, 100874. [CrossRef]
32. Cui, H.; Li, X.; Gao, L. An Improved Multi-Population Genetic Algorithm with a Greedy Job Insertion Inter-Factory Neighborhood Structure for Distributed Heterogeneous Hybrid Flow Shop Scheduling Problem. *Expert Syst. Appl.* **2023**, *222*, 119805. [CrossRef]
33. Qin, H.; Han, Y.; Wang, Y.; Liu, Y.; Li, J.; Pan, Q. Intelligent Optimization under Blocking Constraints: A Novel Iterated Greedy Algorithm for the Hybrid Flow Shop Group Scheduling Problem. *Knowl.-Based Syst.* **2022**, *258*, 109962. [CrossRef]
34. Qin, H.; Han, Y.; Chen, Q.; Wang, L.; Wang, Y.; Li, J.; Liu, Y. Energy-Efficient Iterative Greedy Algorithm for the Distributed Hybrid Flow Shop Scheduling with Blocking Constraints. *IEEE Trans. Emerg. Top. Comput. Intell.* **2023**, 1–16. [CrossRef]
35. Wang, Y.-J.; Wang, G.-G.; Tian, F.-M.; Gong, D.-W.; Pedrycz, W. Solving Energy-Efficient Fuzzy Hybrid Flow-Shop Scheduling Problem at a Variable Machine Speed Using an Extended NSGA-II. *Eng. Appl. Artif. Intell.* **2023**, *121*, 105977. [CrossRef]
36. Qin, H.-X.; Han, Y.-Y.; Zhang, B.; Meng, L.-L.; Liu, Y.-P.; Pan, Q.-K.; Gong, D.-W. An Improved Iterated Greedy Algorithm for the Energy-Efficient Blocking Hybrid Flow Shop Scheduling Problem. *Swarm Evol. Comput.* **2022**, *69*, 100992. [CrossRef]

37. Shao, Z.; Pi, D.; Shao, W.; Yuan, P. An Efficient Discrete Invasive Weed Optimization for Blocking Flow-Shop Scheduling Problem. *Eng. Appl. Artif. Intell.* **2019**, *78*, 124–141. [CrossRef]
38. Han, X.; Han, Y.; Zhang, B.; Qin, H.; Li, J.; Liu, Y.; Gong, D. An Effective Iterative Greedy Algorithm for Distributed Blocking Flowshop Scheduling Problem with Balanced Energy Costs Criterion. *Appl. Soft Comput.* **2022**, *129*, 109502. [CrossRef]
39. Qin, H.-X.; Han, Y.-Y.; Liu, Y.-P.; Li, J.-Q.; Pan, Q.-K.; Han, X. A Collaborative Iterative Greedy Algorithm for the Scheduling of Distributed Heterogeneous Hybrid Flow Shop with Blocking Constraints. *Expert Syst. Appl.* **2022**, *201*, 117256. [CrossRef]
40. Zhang, C.; Tan, J.; Peng, K.; Gao, L.; Shen, W.; Lian, K. A Discrete Whale Swarm Algorithm for Hybrid Flow-Shop Scheduling Problem with Limited Buffers. *Robot. Comput.-Integr. Manuf.* **2021**, *68*, 102081. [CrossRef]
41. Yu, Z.; Wang, S. The Research of Trailer Scheduling Based on the Hybrid Flow Shop Problem with Blocking. In Proceedings of the World Congress on Intelligent Control & Automation, Chongqing, China, 25–27 June 2008.
42. Zhang, Q.; Yu, Z. Population-Based Multi-Layer Iterated Greedy Algorithm for Solving Blocking Flow Shop Scheduling Problem. *Comput. Integr. Manuf. Syst.* **2016**, *22*, 2315–2322. [CrossRef]
43. Zheng, Y.; Mo, G.; Zhang, J. Blocking Flow Line Scheduling of Panel Block in Shipbuilding. *Comput. Integr. Manuf. Syst.* **2016**, *22*, 2305–2314. [CrossRef]
44. Riahi, V.; Newton, M.A.H.; Su, K.; Sattar, A. Constraint Guided Accelerated Search for Mixed Blocking Permutation Flowshop Scheduling. *Comput. Oper. Res.* **2019**, *102*, 102–120. [CrossRef]
45. Rodriguez, F.J.; Lozano, M.; Blum, C.; García-Martínez, C. An Iterated Greedy Algorithm for the Large-Scale Unrelated Parallel Machines Scheduling Problem. *Comput. Oper. Res.* **2013**, *40*, 1829–1841. [CrossRef]
46. Fernandez-Viagas, V.; Valente, J.M.S.; Framinan, J.M. Iterated-Greedy-Based Algorithms with Beam Search Initialization for the Permutation Flowshop to Minimise Total Tardiness. *Expert Syst. Appl.* **2018**, *94*, 58–69. [CrossRef]
47. Chen, S.; Pan, Q.-K.; Gao, L.; Sang, H. A Population-Based Iterated Greedy Algorithm to Minimize Total Flowtime for the Distributed Blocking Flowshop Scheduling Problem. *Eng. Appl. Artif. Intell.* **2021**, *104*, 104375. [CrossRef]
48. Pan, Q.-K.; Ruiz, R. An Effective Iterated Greedy Algorithm for the Mixed No-Idle Permutation Flowshop Scheduling Problem. *Omega* **2014**, *44*, 41–50. [CrossRef]
49. Qin, H.; Han, Y.; Chen, Q.; Li, J.; Sang, H. A Double Level Mutation Iterated Greedy Algorithm for Blocking Hybrid Flow Shop Scheduling. *Control. Decis.* **2022**, *37*, 2323–2332. [CrossRef]
50. Missaoui, A.; Ruiz, R. A Parameter-Less Iterated Greedy Method for the Hybrid Flowshop Scheduling Problem with Setup Times and Due Date Windows. *Eur. J. Oper. Res.* **2022**, *303*, 99–113. [CrossRef]
51. Nejati, M.; Mahdavi, I.; Hassanzadeh, R.; Mahdavi-Amiri, N.; Mojarad, M. Multi-Job Lot Streaming to Minimize the Weighted Completion Time in a Hybrid Flow Shop Scheduling Problem with Work Shift Constraint. *Int. J. Adv. Manuf. Technol.* **2014**, *70*, 501–514. [CrossRef]
52. Marichelvam, M.K.; Geetha, M.; Tosun, Ö. An Improved Particle Swarm Optimization Algorithm to Solve Hybrid Flowshop Scheduling Problems with the Effect of Human Factors—A Case Study. *Comput. Oper. Res.* **2020**, *114*, 104812. [CrossRef]
53. Huang, J.-P.; Pan, Q.-K.; Gao, L. An Effective Iterated Greedy Method for the Distributed Permutation Flowshop Scheduling Problem with Sequence-Dependent Setup Times. *Swarm Evol. Comput.* **2020**, *59*, 100742. [CrossRef]