An improved iterated greedy algorithm for the distributed flow shop scheduling problem with sequence-dependent setup times

Abstract—In various flow shop scheduling problems, it is very common that a large-scale production is done. Under this situation, more factories are of more practical interest than a single factory. Thus, the distributed permutation flow shop scheduling problems (DPFSPs) have been attracted much attention by researchers. However, the DPFSP is more complicated than the traditional flow shop scheduling problems. It considers not only the processing order of the jobs, but also how to distribute the jobs to multiple parallel factories for parallel processing. In addition, the sequence-dependent setup time (SDST) constraint of machines is taken into account to well study the above DPFSP with SDST. This paper presents a simple and effective iterated greedy algorithm. It is proposed to replace the traditional insertion-based local search with exchange-based local search, which greatly improves the search efficiency. The proposed new iterated greedy (NIG) algorithm is applied to test instances, and compares with the state-of-the-art algorithms. Our empirical results demonstrate that the proposed algorithm outperforms the compared algorithms and can obtain the best solution of DPFSP.

Keywords—distributed permutation flow shop, iterated greedy algorithm, local search, swap

I. INTRODUCTION

With the continuous development of global economy, a company often needs several production centers, it is necessary to set up a distributed production mode [1]. In recent years, the distributed permutation flow shop problem (DPFSP) has gradually attracted much attention by researchers. DPFSP can be described as follows: a total of $J$ jobs need to be processed in $F$ factories, each factory contains $M$ serial machines, each factory can independently complete the processing of the job. DPFSP was first proposed by Naderi and Ruiz [1], who used an iterated greedy algorithm to solve DPFSP with makespan criteria. They proposed six different mixed integer linear programming (MILP) models, and two simple factory assignment rules together with 14 heuristics based on the dispatching rules, the effective constructive heuristics and variable neighborhood descent methods. Bargauti et al. designed a CRO algorithm with good single point crossover and effective greedy strategy embedding. The algorithm uses an effective NEH heuristic method to generate initial solution groups [3]. Fernandez-Viagas and Perez-Gonzalez proposed 18 constructive heuristic methods and an improved iterated algorithm to obtain high-quality solutions [4]. Meng et al. developed three meta-heuristic methods to solve the DPFSP with customer order constraints [5]. In [6], Jing et al. proposed an iterated greedy (IG) algorithm with free time insertion evaluation to solve DPFSP with time window.

In practical production, different jobs will face operations such as tool replacement and equipment inspection when they are processed on the same machine. The time spent by these operations is often not only related to the job to be processed, but also related to the job previously processed on the submachine. Therefore, a sequence-dependent setup times (SDST) will be generated. About SDST / DPFSP, Huang gave a complete description in [2] and proposed an improved IG algorithm to solve this problem. Later, for the same problem, Huang integrated three constructive heuristic methods into a discrete artificial bee colony (DABC) algorithm to generate good scheduling solutions [7].

In this paper, the iterated greedy (IG) algorithm with excellent performance in DPFSP is selected. Compared with other algorithms based on metaphor or inspired by nature, the IG algorithm is simple in structure and easy to understand [15]. IG algorithm is used in the study of multi-objective flow shop scheduling problem after continuous expansion and improvement [9]. In References, IG algorithm was applied to the no-idle flow shop scheduling problem [10]. In addition, the IG with taboo reconstruction strategy is used to solve the FSP with...
Ruiz improves the performance of IG algorithm by improving initialization, destruction, construction and local search [12]. Karabulut proposed an IG algorithm based on the temperature calculation formula as the acceptance criterion, and hybridized it with a random search algorithm [13]. Li et al. proposed a new simulated annealing algorithm with four domain structures [14]. Recently, Huang proposed a restart scheme with six different operators in [2].

Our contribution is that an improve IG algorithm is proposed, in which the key factory operation exchange strategy and LS-N method based on the exchange strategy are adopted to replace the insertion strategy used in the traditional IG algorithm. Through comparison experiments with other algorithms, it is proved that the performance of the proposed algorithm is significantly improved.

II. THE SDST/DPFSP PROBLEM

SDST/DPFSP described as follows: There are $n$ jobs, $J = (J_1, J_2, ..., J_n)$, which need to be processed in $f$ identical factories, $F = (F_1, F_2, ..., F_f)$, and each factory has $m$ machines, $M = (M_1, M_2, ..., M_m)$. Each job $J_i$ can be processed in any factory $f$. The job is processed on the machine in the order from the first machine to the last machine, and the factory cannot be replaced during processing.

The processing time of job $J_i$ on machine $M_j$ is $p_{ij}$. When Job $J_i$ is on machine $M_j$, when $J_j$ is the previous Job processed on the machine, SDST is $s_{ij,j}$. Each machine can only process one Job at any time, and one Job can only be processed in the same factory. Once a factory is selected, the factory cannot be replaced during processing. All operations are independent, and all factories start processing from 0 when processing the Job.

The purpose of SDST / DPFSP in this paper is to assign jobs reasonably to the factory and find a job sequence to minimize makespan ($C_{max}$). The mathematical model of the problem is described as follows:

Notations:
- $f$: Number of factories.
- $m$: Number of machines in each factory.
- $n$: Number of jobs needed to be processed.
- $J = (J_1, J_2, ..., J_n)$: The set of $n$ jobs to be processed.
- $M = (M_1, M_2, ..., M_m)$: The set of $m$ machines, where $M_j$ is the $j^{th}$ machine used to complete the $i^{th}$ process of jobs, $M_i \in M$.
- $F = (F_1, F_2, ..., F_f)$: The set of $F$ parallel factories, where $F_j$ is the $j^{th}$ factory from set $F$, $F_j \in F$.
- $o_{ij}$: The operation of job $J_i$ on machine $M_j$.
- $p_{ij}$: The processing time of job $J_i$ on machine $M_j$.
- $s_{ij,j'}$: The setup time of job $J_i$ on machine $M_j$, When the job is the first job processed on machine $M_j$, then $j' = j$.

Decision variables:
- $y_{ij,q}$: when job $J_i$ is the $q^{th}$ job in factory $F_j$, the value of the decision variable is 1, otherwise it is 0.
- $y_{ij}$: when job $J_i$ is processed in factory $F_j$, the value of the decision variable is 1, otherwise it is 0.

Objective:

Min $C_{max}$

Subject to:

1. $y_{ij,q} = \sum_{q=1}^{n} x_{j,i,j,q} \quad \forall J_i \in J, \forall M_j \in M, \forall F_j \in F$ (2)
2. $\sum_{q=1}^{n} x_{j,i,j,q} \leq 1, \forall F_j \in F, \forall M_j \in M, \forall q \in \{1, 2, ..., n\}$ (3)
3. $\sum_{q=1}^{n} x_{j,i,j,q} \geq \sum_{q=1}^{n} y_{j,i,j,q} \quad \forall F_j \in F, \forall M_j \in M, \forall q \in \{1, 2, ..., n-1\}$ (4)
4. $ST_{i,j} \leq CT_{i,j} \quad \forall J_i \in J, \forall M_j \in M$ (5)
5. $MCT_{i,j} = MST_{i,j} + \sum_{j=1}^{n} p_{i,j} x_{j,i,j,q} \quad \forall M_j \in M, \forall F_j \in F, \forall q \in \{1, 2, ..., n\}$ (6)
6. $MST_{i,j} \geq MST_{i,j,q} \quad \forall M_j \in M, \forall F_j \in F, \forall q \in \{1, 2, ..., n-1\}$ (7)
7. $MST_{i,j} + G(1-x_{j,i,j,q}) \geq MST_{i,j,q} + \sum_{j=1}^{n} y_{j,i,j,q} \quad \forall M_j \in M, \forall F_j \in F, \forall q \in \{1, 2, ..., n-1\}$ (8)
8. $MST_{i,j} + G(1-x_{j,i,j,q}) \geq s_{j,i,j} \quad \forall J_i \in J, \forall M_j \in M, \forall F_j \in F$ (9)
9. $CT_{i,j} = ST_{i,j} + p_{i,j} \quad \forall J_i \in J, \forall M_j \in M, \forall F_j \in F$ (10)
10. $MST_{i,j} \geq 0, \forall M_j \in M, \forall F_j \in F, \forall q \in \{1, 2, ..., n\}$ (11)
11. $ST_{i,j} \geq 0, \forall J_i \in J, \forall M_j \in M$ (12)

Constraint (1) indicates that the indicator of the problem studied in this paper is to minimize makespan. Constraint (2) means that each job can only be processed on one machine in a factory at a time, and constraint (3) means that each machine can only process one job at a time. Constraint (4) states that the processing of operations on the machine can only be performed sequentially, and the processing time cannot be overlapped. Constraint (5) stipulates that the processing sequence of the job cannot be changed. Constraint (6) describes the start time and completion time of a workpiece processing. Constraint (7)
represents the start time must be equal or greater than the completion time of two adjacent jobs on a certain machine. Constraint (8) describes the constraints between the start time and completion time of the job including preparation time. Constraint (9) refers to the situation when the job is first processed on the machine. In constraint (10), the completion time of a job is the sum of the start time and processing time of the job. Constraints (11) and (12) indicate that the start time of each machine and each job is not less than 0, respectively.

Next, we use an example to illustrate how to calculate makespan. The example includes two factories \((f = 2)\), two machines \((m = 2)\) and five jobs \((n = 5)\). Table 1 and Table 2 show the processing time and setup time. It is assumed that a solution is that jobs 1, 2 and 4 are processed in factory 1, and jobs 3 and 5 are processed in factory 2. The value of decision variables can be obtained as follows: \(x_{1,1,1}=1\), \(x_{2,2,1}=1\), \(x_{3,1,1}=1\), \(x_{4,2,1}=1\), \(x_{5,2,1}=1\), \(x_{1,2,2}=1\), \(x_{2,2,2}=1\), \(y_{1,1}=1\), \(y_{2,1}=1\), \(y_{3,1}=1\), \(y_{5,1}=1\). The remaining decision variables are 0. The objective value is \(C_{\text{max}} = 333\). The scheduling Gantt is shown in Fig. 1.

### Table I. Processing Times of Jobs on Machines \(M_1\) and \(M_2\)

<table>
<thead>
<tr>
<th>Job</th>
<th>(M_1)</th>
<th>(M_2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(J_1)</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>(J_2)</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>(J_3)</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>(J_4)</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>(J_5)</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

### Table II. Sequence-Dependent Setup Times of Jobs on Machines \(M_1\) and \(M_2\)

<table>
<thead>
<tr>
<th>Job</th>
<th>(J_1)</th>
<th>(J_2)</th>
<th>(J_3)</th>
<th>(J_4)</th>
<th>(J_5)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(M_1)</td>
<td>6</td>
<td>7</td>
<td>9</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>(M_2)</td>
<td>4</td>
<td>2</td>
<td>6</td>
<td>3</td>
<td>7</td>
</tr>
</tbody>
</table>

III. PROPOSED IG ALGORITHM FOR THE SDST/DPFSP

In this paper, we propose an improved iterated greedy algorithm with simple structure. IG algorithm has shown excellent performance in solving the flow shop scheduling problem [15]. For example, Ding and Song used tabu-based reconstruction strategy to enhance the search ability of the algorithm in [16]. Ruiz and Pan improved the local search, Destruction and Construction of IG algorithm, which greatly improved the performance of the algorithm [10]. The traditional IG algorithm uses a heuristic method, generally NEH, to create the initial solution. The iterative content includes four parts: destruction, reconstruction, local search and acceptance criteria. When the termination condition is satisfied, the iteration stops.

The traditional IG algorithm framework is as follows:

#### Algorithm 1 The traditional IG algorithm

01: Begin:
02: Set the parameters: \(T\)
03: \(\pi^0 = \text{InitialSolution} \ M_1\)
04: \(\pi^0 = \text{LocalSearch}(\pi^0)\)
05: while termination criterion is not satisfied do
06: \(\pi^0, \pi^k = \text{Destruction}(\pi^k)\)
07: \(\pi = \text{Construction}(\pi^0, \pi^k)\)
08: \(\pi = \text{LocalSearch}(\pi)\)
09: \(\pi^0 = \text{AcceptanceCriterion}(\pi, \pi^0, T)\)
10: Endwhile
11: End

At the same time, we use the flowchart to show the running process of the IG algorithm, as shown in Fig. 2:

### A. Initial Solution

The generation of initial solution is the first step of the program, and its quality is crucial. We retained NEH2_en in [12] to generate the initial solution that meets the requirements. Firstly, all jobs are sorted in descending order according to the sum of processing time on all machines to generate sequence.
seeds, \( r = \{r_1, r_2, ..., r_n \} \). Then, the first job in the sequence is assigned to the first factory, and the second to the second factory. After all the factories are assigned, the remaining jobs are taken out one by one, and then the best position is tried and found in all sequences for insertion. After the insertion operation, the former or latter job of the insertion position is taken out, and tested at all positions of the same factory, and the insertion operation is performed at the optimal position. In addition, referring to the previous DPFSP literature, all the insertion processes adopt the well-known Taillard acceleration of the insertion neighborhood to improve the insertion speed.

The pseudocode of NEH2 en is as follows:

Algorithm 2 NEH2 en

01: Begin:
02: Compute \( P = \sum_{i=1}^{n} P_{ij}, J_j \in J \) % \( P_j \) is the total processing time of job \( J_j \).
03: \( \tau = \{r_1, r_2, ..., r_n \} \leftarrow \) sort jobs in descending order of \( P_j \).
04: for \( j = 1 \) to \( f \) do
05: Assign job \( r_j \) in \( \tau \) to plant \( F_j \).
06: endfor
07: for \( j = f + 1 \) to \( n \) do
08: for \( l = 1 \) to \( f \) do
09: Test \( r_j \) in all possible positions in \( \pi_j \), % Taillard acceleration is applied
10: Get the lowest makespan \( C'_{\pi_j} \) in factory \( F_j \).
11: Get the position of \( C'_{\pi_j} \) is \( pos_{\pi_j} \).
12: endfor
13: \( I^* = \arg(\min_{\pi_j} C'_{\pi_j}) \)
14: Insert \( r_j \) in the sequence \( \pi_{\pi_j} \) at position \( pos_{\pi_j} \).
15: Extract at random job \( h \) from position \( pos_{\pi_j} - 1 \) or \( pos_{\pi_j} + 1 \) from \( \pi_{\pi_j} \).
16: Insert job \( h \) in \( \pi_{\pi_j} \) at the position resulting in the lowest makespan
17: endfor
18: Output \( \pi \)

B. Local search

Local search in IG algorithm is an important factor to determine the quality of solution. Insertion operation is generally used in the local search of traditional IG algorithm. For example, for the case of \( n \) jobs, insertion attempts are generally required \( n + 1 \) locations, and the time complexity is \( O(n(n+1)) \). If the selected location is \( p \), the number of jobs that need to be moved is \( n - p + 1 \), and the time complexity becomes \( O(n(n+1)(n-p+1)) \). If the result of the execution does not make the target value improved, the above operation will be repeated until the target value is improved, which makes the number of effective executions in a certain period of time reduced. Therefore, in this paper, we abandon the previous local operation dominated by the insertion strategy, and choose the swapping strategy based on the previous LS_3 \[12\]. We call it LS_N, and its time complexity is only \( O(n^3) \). The operation steps of LS_N are as follows.

Firstly, find out the key factories in all factories, that is, the factory with the largest makespan, and then find out the factory with the largest makespan except the key factory. Then, two jobs are randomly selected in two factories to test the makespan \( C'_{\max} \) after the exchange. If the makespan \( C'_{\max} \) is less than the original \( C_{\max} \), the exchange is retained. Otherwise, the solution remains unchanged. Then continue to select the job until all the jobs in the key factory are traversed. LS_N algorithm framework is as follows:

Algorithm 3 LS_N

01: Begin
02: Find the critical factory \( F_{c1} \) with the \( C_{\max} \)
03: Find the critical factory \( F_{c2} \) with the \( C'_{\max} \) % \( C'_{\max} \) is a value only less than \( C_{\max} \)
04: \( Cnt = 0 \)
05: While \( Cnt < n \) do % \( n \) is the number of jobs in factory \( F_{c1} \)
06: \( \pi_{\text{initial}} = \pi \)
07: \( r^* = \) randomly selected job in \( F_{c1} \)
08: \( \tau^* = \) randomly selected job in \( F_{c2} \)
09: Test job \( r^* \) and \( \tau^* \) for swap
10: if \( C_{\max} > C'_{\max} \)
11: \( \pi = \pi_{\text{initial}} \)
12: \( Cnt = Cnt + 1 \)
13: elseif
14: Swap job \( r^* \) and \( \tau^* \)
15: \( C_{\max} = \max_{f=1}^{n} (C_{\max}(\pi_1), C_{\max}(\pi_2), ..., C_{\max}(\pi_f)) \)
16: Find the critical factory \( F_{c1} \) with the \( C_{\max} \)
17: Find the critical factory \( F_{c2} \) with the \( C'_{\max} \)
18: \( Cnt = 0 \)
19: endif
20: endwhile
21: Output \( \pi \)

C. Destruction, construction and acceptance criteria

In the loop phase of IG algorithm, we repeat the four steps of Destruction, Construction, local search and acceptance criteria until the quality of the solution is improved. In the Destruction phase, first select a factory containing \( n(n > d) \) jobs, then randomly select a job from it, add it to the sequence \( \pi^0 \) (stored deleted jobs), and delete the job from the original sequence. Repeat the last operation \( d \) times to get two sequences, sequence \( \pi^0 \) with \( d \) jobs and sequence \( \pi^* \) (store the remaining sequence of jobs without changing the sequence) with \( n-d \) jobs.

The Destruction pseudo-code is given in Algorithm 4.
Algorithm 4 Destruction

01: Begin
02: Procedure Destruction(π, d)
03: πₚ = π
04: Cnt = 0, πₚ = ∅
05: while Cnt < d do
06:   l = rand()%f + 1
07:   if n > 1 then
08:     j = rand()%n+1
09:     πₚ = πₚ ∪ {πₚₖₗ}
10:    Delete job πₚₖₗ from πₚ
11:   endif
12:   Cnt++
13: endwhile
14: Output πₚ, πₚ

The reconstruction phase is, firstly, the first job is taken out from πₚ, and it is tried in all positions until the best insertion position is found, that is, the position with the smallest makespan. Then, the second job is taken out, and the above operation is performed... until all the jobs in πₚ are taken out and inserted.

The Construction pseudocode is given in algorithm 5.

Algorithm 5 Construction

01: Begin
02: Procedure Construction(πₚ, πₚ)
03: for j = 1 to d
04:   Job = the jth job of πₚ
05:   for l = 1 to f do
06:     Test Job is inserted in all positions of πₚₖ
07:     Get the lowest makespan Cₖ in factory Fₖ
08:     Get the position of Cₖ is posₖ
09:   endfor
10:   lₚ = arg(minₖ Cₖ)
11:   Insert Job in the sequence πₚₖ at position posₖ
12: endfor
13: π = πₚ
14: Output π

Since the code referenced in this paper is the code in Ruiz's [12], the code in the 'Receiving Criteria' section still chooses Ruiz and Stützle (2007), a constant temperature acceptance criterion based on parameter T, as follows:

\[ Temperature = T \cdot \frac{\sum_{i=1}^{n} \sum_{j=1}^{m} P_{i,j}}{n \cdot m \cdot 10} \]  \hspace{1cm} (14)

T needs calibration, but has shown robustness (most values are not zero and not too high).

IV. EXPERIMENT AND ANALYSIS

The performance of the NIG algorithm is evaluated using 270 examples. In the example, n, m and f are the number of jobs, machines and factories, respectively. Their values are as follows: n ∈ {100, 200, 300, 400, 500}, m ∈ {5, 8, 10}, f ∈ {2, 3, 4, 5, 6, 7}. We randomly combine their values to generate instances of different sizes. In the example, the data are generated in a random way. The distribution range of processing time is [1, 99], and the value range of preparation time is (1+rand()%99)×factor/100. In this experiment, we repeat each instance five times, and each instance takes the minimum value in five runs. The index of this experiment is to minimize makespan (Cₘ₉₉).

All test algorithms are compiled and coded by Visual Studio 2019, C++, running on Microsoft Windows 10 operating system, 16GB DDR4 memory and 1.00 GHZ Intel Core i5 - 1035G1 processor.

A. Computational evaluation

The DPFSP is a NP-hard problem, thus we cannot obtain the optimal solution. Therefore, in this paper, we will use the performance evaluation method of the single objective optimization algorithm commonly used in the literature to evaluate the performance of the algorithm, namely the percentage relative deviation RPI. The calculation method of RPI is as follows:

\[ RPI = \frac{M_i - M_{best}}{M_{best}} \times 100\% \]  \hspace{1cm} (15)

Among them, Mᵢ is the minimum makespan of the algorithm running five times in an instance, and Mₙₑᵢ is the minimum makespan of all the algorithms running five times in this instance. It can be seen from the formula that in the comparison algorithm, the better the performance is, the closer the RPI is to 0, and the algorithm with RPI equals to 0 has the best performance in the current comparison algorithm.

In this paper, five algorithms are selected to compare with NIG, namely discrete artificial bee colony algorithm (DABC) [17], artificial chemical reaction optimization (CRO) [3], discrete differential evolution algorithm (DDE) [18], and improved iterated greedy algorithm IGA [12] and IGR [2]. For the 270 examples mentioned above, all algorithms are executed five times in comparison. In order to obtain fair results, we set the termination condition as:

\[ TimeLimit = CPU \times n \times m \]  \hspace{1cm} (16)

n and m are the number of jobs and the number of machines in the current instance respectively.

B. RPI comparison

Table 3 shows the RPI under different instances, and the average RPI of each algorithm under different factory numbers. According to the above formula for calculating RPI, the performance of NIG algorithm is significantly better than that of the other five algorithms. With the increase of examples, the performance of DABC, DDE and IGA algorithms gradually deteriorates, while the performance of IGR gradually improves. Overall, the performance of DDE and IGR is good, but the performance of NIG is the best.
### TABLE III.
RPI OF COMPARISON ALGORITHM WHEN CPU = 10

<table>
<thead>
<tr>
<th>Factory</th>
<th>J*M</th>
<th>DABC</th>
<th>CRO</th>
<th>DDE</th>
<th>IGA</th>
<th>IGM</th>
<th>NIG</th>
</tr>
</thead>
<tbody>
<tr>
<td>100*5</td>
<td>0.054</td>
<td>0.049</td>
<td>0.015</td>
<td>0.039</td>
<td>0.018</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>100*8</td>
<td>0.053</td>
<td>0.077</td>
<td>0.015</td>
<td>0.079</td>
<td>0.046</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>100*10</td>
<td>0.082</td>
<td>0.064</td>
<td>0.021</td>
<td>0.050</td>
<td>0.031</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>200*5</td>
<td>0.062</td>
<td>0.028</td>
<td>0.022</td>
<td>0.053</td>
<td>0.030</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>200*8</td>
<td>0.067</td>
<td>0.035</td>
<td>0.010</td>
<td>0.052</td>
<td>0.033</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>200*10</td>
<td>0.067</td>
<td>0.053</td>
<td>0.009</td>
<td>0.069</td>
<td>0.039</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>300*5</td>
<td>0.051</td>
<td>0.038</td>
<td>0.011</td>
<td>0.064</td>
<td>0.033</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>300*8</td>
<td>0.051</td>
<td>0.038</td>
<td>0.011</td>
<td>0.064</td>
<td>0.033</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>300*10</td>
<td>0.055</td>
<td>0.028</td>
<td>0.008</td>
<td>0.037</td>
<td>0.034</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>400*5</td>
<td>0.055</td>
<td>0.040</td>
<td>0.018</td>
<td>0.081</td>
<td>0.041</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>400*8</td>
<td>0.043</td>
<td>0.028</td>
<td>0.010</td>
<td>0.060</td>
<td>0.033</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>400*10</td>
<td>0.037</td>
<td>0.026</td>
<td>0.011</td>
<td>0.040</td>
<td>0.029</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>500*5</td>
<td>0.038</td>
<td>0.030</td>
<td>0.015</td>
<td>0.044</td>
<td>0.024</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>500*8</td>
<td>0.043</td>
<td>0.024</td>
<td>0.012</td>
<td>0.047</td>
<td>0.031</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>500*10</td>
<td>0.040</td>
<td>0.031</td>
<td>0.005</td>
<td>0.038</td>
<td>0.028</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>100*5</td>
<td>0.052</td>
<td>0.057</td>
<td>0.003</td>
<td>0.043</td>
<td>0.031</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>100*8</td>
<td>0.105</td>
<td>0.097</td>
<td>0.026</td>
<td>0.070</td>
<td>0.060</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>100*10</td>
<td>0.073</td>
<td>0.082</td>
<td>0.021</td>
<td>0.055</td>
<td>0.034</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>200*5</td>
<td>0.046</td>
<td>0.042</td>
<td>0.018</td>
<td>0.046</td>
<td>0.024</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>200*8</td>
<td>0.076</td>
<td>0.061</td>
<td>0.005</td>
<td>0.066</td>
<td>0.038</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>200*10</td>
<td>0.066</td>
<td>0.055</td>
<td>0.009</td>
<td>0.073</td>
<td>0.041</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>300*5</td>
<td>0.060</td>
<td>0.036</td>
<td>0.006</td>
<td>0.032</td>
<td>0.022</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>300*8</td>
<td>0.043</td>
<td>0.057</td>
<td>0.007</td>
<td>0.062</td>
<td>0.043</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>300*10</td>
<td>0.054</td>
<td>0.031</td>
<td>0.004</td>
<td>0.051</td>
<td>0.033</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>400*5</td>
<td>0.036</td>
<td>0.025</td>
<td>0.008</td>
<td>0.040</td>
<td>0.025</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>400*8</td>
<td>0.057</td>
<td>0.042</td>
<td>0.004</td>
<td>0.055</td>
<td>0.035</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>400*10</td>
<td>0.055</td>
<td>0.047</td>
<td>0.001</td>
<td>0.044</td>
<td>0.029</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>500*5</td>
<td>0.026</td>
<td>0.029</td>
<td>0.006</td>
<td>0.035</td>
<td>0.019</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>500*8</td>
<td>0.038</td>
<td>0.043</td>
<td>0.000</td>
<td>0.061</td>
<td>0.036</td>
<td>0.001</td>
<td></td>
</tr>
<tr>
<td>500*10</td>
<td>0.043</td>
<td>0.021</td>
<td>0.005</td>
<td>0.047</td>
<td>0.029</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>mean</td>
<td>0.053</td>
<td>0.039</td>
<td>0.013</td>
<td>0.053</td>
<td>0.032</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>100*5</td>
<td>0.058</td>
<td>0.049</td>
<td>0.010</td>
<td>0.040</td>
<td>0.019</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>100*8</td>
<td>0.067</td>
<td>0.086</td>
<td>0.019</td>
<td>0.057</td>
<td>0.024</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>100*10</td>
<td>0.082</td>
<td>0.085</td>
<td>0.026</td>
<td>0.047</td>
<td>0.039</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>200*5</td>
<td>0.054</td>
<td>0.047</td>
<td>0.008</td>
<td>0.034</td>
<td>0.017</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>200*8</td>
<td>0.051</td>
<td>0.065</td>
<td>0.0003</td>
<td>0.061</td>
<td>0.039</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>200*10</td>
<td>0.083</td>
<td>0.071</td>
<td>0.008</td>
<td>0.049</td>
<td>0.040</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>300*5</td>
<td>0.068</td>
<td>0.055</td>
<td>0.013</td>
<td>0.055</td>
<td>0.026</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>300*8</td>
<td>0.065</td>
<td>0.047</td>
<td>0.011</td>
<td>0.078</td>
<td>0.036</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>300*10</td>
<td>0.066</td>
<td>0.036</td>
<td>0.006</td>
<td>0.047</td>
<td>0.033</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>400*5</td>
<td>0.039</td>
<td>0.044</td>
<td>0.009</td>
<td>0.037</td>
<td>0.016</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>400*8</td>
<td>0.073</td>
<td>0.042</td>
<td>0.003</td>
<td>0.048</td>
<td>0.030</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>400*10</td>
<td>0.060</td>
<td>0.046</td>
<td>0.006</td>
<td>0.050</td>
<td>0.035</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>500*5</td>
<td>0.053</td>
<td>0.038</td>
<td>0.016</td>
<td>0.054</td>
<td>0.030</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>500*8</td>
<td>0.066</td>
<td>0.037</td>
<td>0.018</td>
<td>0.052</td>
<td>0.028</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>500*10</td>
<td>0.056</td>
<td>0.047</td>
<td>0.028</td>
<td>0.066</td>
<td>0.040</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>mean</td>
<td>0.063</td>
<td>0.053</td>
<td>0.012</td>
<td>0.052</td>
<td>0.030</td>
<td>0.000</td>
<td></td>
</tr>
<tr>
<td>100*5</td>
<td>0.054</td>
<td>0.027</td>
<td>0.009</td>
<td>0.018</td>
<td>0.007</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>100*8</td>
<td>0.114</td>
<td>0.080</td>
<td>0.026</td>
<td>0.053</td>
<td>0.029</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>100*10</td>
<td>0.116</td>
<td>0.115</td>
<td>0.060</td>
<td>0.061</td>
<td>0.051</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>200*5</td>
<td>0.074</td>
<td>0.083</td>
<td>0.040</td>
<td>0.081</td>
<td>0.041</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>200*8</td>
<td>0.095</td>
<td>0.069</td>
<td>0.000</td>
<td>0.056</td>
<td>0.045</td>
<td>0.005</td>
<td></td>
</tr>
<tr>
<td>mean</td>
<td>0.063</td>
<td>0.053</td>
<td>0.012</td>
<td>0.052</td>
<td>0.030</td>
<td>0.000</td>
<td></td>
</tr>
</tbody>
</table>
To identify the acquired data more clearly, we also use confidence intervals to analyze the data, as shown in Fig. 3. It can be seen from the graph that the performance of NIG algorithm is significantly better than other algorithms. In addition, DDE algorithm and IGR algorithm are also better, while DABC and IGA are relatively worse.

Next, we further analyze the convergence of the algorithm. We have selected four algorithms, DDE, IGA, IGR, and NIG, which perform well in the problem, as shown in Fig. 4. In order to make the expression clearer, we have selected three question examples of 300*5*3, 400*5*6, 500*5*6. Since the termination condition of the programming operation is $TimeLimit = CPU \times n \times m$, we can change the running time of the algorithm by changing the value of the CPU. Therefore, the X axis in Fig. 3 is the value of the CPU, the optimal solution (makespan) obtained by running the algorithm five times is the Y axis. It can be seen from the three figures a, b, and c that DDE is easy to fall into the local optimum. The performance of the IGR algorithm gradually increases with the increase of the instance size. The NIG algorithm performs best in these four algorithms.
V. CONCLUSIONS

In this paper, the DPFSP with sequence-dependent preparation time is studied and solved by the improved IG algorithm. Firstly, the mathematical model of SDST / DPFSP is described. Then, this paper proposes a local search based on exchange strategy, which improves the search efficiency. Through compared with other existing algorithms, the performance has been significantly improved.

In the future, we will do more research on problem-oriented strategies, focus more on algorithm thinking and consider different optimization objectives. For example, the green energy saving index is added to the problem of distributed displacement flow shop, and more attention is paid to the thinking of problems related to real life.

ACKNOWLEDGMENT

This work was jointly supported by National Natural Science Foundation of China with grant No. 61803192, 61973203, 61966012, 61773192, 61603169, 61773246, and 71533001. Thanks for the support of Shandong province colleges and universities youth innovation talent introduction and education program.

REFERENCES


