An integrated search heuristic for large-scale flexible job shop scheduling problems

2 authors, including:

Yuan Yuan
Michigan State University
32 PUBLICATIONS   **699** CITATIONS

SEE PROFILE

An integrated search heuristic for large-scale flexible job shop scheduling problems

Yuan Yuan, Hua Xu*

State Key Laboratory of Intelligent Technology and Systems, Tsinghua National Laboratory for Information Science and Technology, Department of Computer Science and Technology, Tsinghua University, Beijing 100084, PR China

ARTICLE INFO

ABSTRACT

The flexible job shop scheduling problem (FJSP) is a generalization of the classical job shop scheduling problem (JSP), where each operation is allowed to be processed by any machine from a given set, rather than one specified machine. In this paper, two algorithm modules, namely hybrid harmony search (HHS) and large neighborhood search (LNS), are developed for the FJSP with makespan criterion. The HHS is an evolutionary-based algorithm with the memetic paradigm, while the LNS is typical of constraint-based approaches. To form a stronger search mechanism, an integrated search heuristic, denoted as HHS/LNS, is proposed for the FJSP based on the two algorithms, which starts with the HHS, and then the solution is further improved by the LNS. Computational simulations and comparisons demonstrate that the proposed HHS/LNS shows competitive performance with state-of-the-art algorithms on large-scale FJSP problems, and some new upper bounds among the unsolved benchmark instances have even been found.

© 2013 Elsevier Ltd. All rights reserved.

1. Introduction

The classical job shop scheduling problem (JSP) is one of the most important and difficult problems in the field of production scheduling and has received an enormous amount of attention in the research literature [1–6]. The flexible job shop scheduling problem (FJSP) is a generalization of the classical JSP, where each operation is allowed to be processed by any machine from a given set, rather than one specified machine. Compared with the classical JSP, the FJSP is closer to a real production environment and has more practical applicability. However, it is more complicated than the classical JSP, because of its additional decision to assign each operation to the appropriate machine. It has been proved that the FJSP is strongly NP-hard even if each job has at most three operations and there are two machines [7].

Due to the computational complexity of the FJSP, exact algorithms are not effective, especially for instances on a large scale. So, meta-heuristics have become the main-stream of research for this problem over the past decade. In the initial study on this subject, tabu search (TS) was applied most successfully to the FJSP [8–11]. The TS proposed by Mastrolilli and Gambardella [11] still represents the state-of-the-art until now. One of the latest important work based on TS can be found in [12], in which a parallel double-level TS was proposed for the FJSP and could

obtain new best known solutions for the benchmark instances from the literature. However, in recent years, two categories of techniques have been more emphasized to address the FJSP, which are evolutionary-based and constraint-based approaches.

Evolutionary-based approaches attempt to solve the FJSP by using evolutionary algorithms. The basic mechanism for this type of approaches is to encode the scheduling solutions to some form of codes, then the corresponding decoding algorithm is carried out to evaluate the codes during the optimization process. Thus far, many evolutionary meta-heuristics have been studied for the FJSP, such as genetic algorithm (GA) [13,14], artificial immune algorithm (AIA) [15], ant colony optimization (ACO) [16], and artificial bee colony (ABC) algorithm [17]. Among these existing works, the memetic paradigm [18], which introduces the problem-dependent local search into the evolutionary algorithm, seems to be more promising to produce high quality solutions for the FJSP [14,17].

Constraint-based approaches are mainly built on a foundation of constraint programming (CP) techniques. Thus, to apply CP to the FJSP is natural because the FJSP is a constraint optimization problem in essence. However, the pure CP is only effective for some small size instances of the FJSP due to the exponentially growing search space. Several important advancements in CP, such as discrepancy search (DS) [19], large neighborhood search (LNS) [20] and iterative flattening search (IFS) [21], changed this situation. Most recently, these techniques have been well tested on the FJSP and achieved the excellent performance on some standard benchmarks [22–24].

Since both of the two categories of techniques have been applied successfully in the FJSP, therefore, the integration of them

* Corresponding author.
 E-mail address: xuhua@tsinghua.edu.cn (H. Xu).

to form a stronger search mechanism appears to be promising. In this paper, two algorithm modules for the FJSP with makespan criterion are developed: hybrid harmony search (HHS) and large neighborhood search (LNS), representing the evolutionary-based and constraint-based approaches, respectively. The HHS algorithm is designed with memetic paradigm, which explores the search space using harmony search (HS) [25], whereas a local search procedure based on the critical path is embedded in the HS to perform the exploitation. The LNS algorithm is devised to improve the current solution continuously by focusing on re-optimizing its subpart employing the CP-based search. Based on the two algorithms, an integrated search heuristic, namely HHS/LNS, is proposed for solving large-scale FJSP problems.

The integration of the HHS and LNS in this paper is motivated by the following aspects: the HHS algorithm could generate high quality solutions quickly; however, when the evolution procedure reaches a certain level, the solution is hard to improve further by tuning several important algorithm parameters; and the LNS has a strong ability to intensify during search, but the ability degrades along with the increase of problem space. Another defect of LNS is that it depends on the initial solution for dealing with some large-scale instances, and the inferior initial solution may lead to a large amount of computation time and poor quality results.

The remainder of this paper is organized as follows. Section 2 formulates the studied problem. Sections 3 and 4 introduce the HHS module and the LNS module, respectively. How to integrate the two modules into a framework is described in Section 5. Afterwards, experimental studies are presented in Section 6. Finally, the paper is summarized in Section 7.

2. Problem formulation

The FJSP is formally formulated as follows. There are a set of n independent jobs $J = \{J_1, J_2, \ldots, J_n\}$ and a set of m machines $M = \{M_1, M_2, \ldots, M_m\}$. Each job J_i consists of a sequence of precedence constrained operations $O_{i,1}$, $O_{i,2}$, ..., O_{i,n_i}. The job J_i is completed only when all its operations are executed in a given order, which can be represented as $O_{i,1} \rightarrow O_{i,2} \rightarrow \cdots \rightarrow O_{i,n_i}$. Each operation $O_{i,j}$, i.e. the jth operation of job J_i, can be executed on any machine selected among a given subset $M_{i,j} \subseteq M$. The processing time of each operation is machine dependent. We denote $p_{i,j,k}$ to be the processing time of $O_{i,j}$ on machine M_k. The scheduling consists of two subproblems: the routing subproblem that assigns each operation to an appropriate machine and the sequencing subproblem that determines a sequence of operations on all the machines. The objective is to find a schedule which minimizes the makespan. The makespan means the time needed to complete all the jobs and can be defined as $C_{\max} = \max_{1 \leq i \leq u}(C_i)$, where C_i is the completion time of job J_i.

Moreover, the following assumptions are made in this paper: all the machines are available at time 0; all the jobs are released at time 0; each machine can process only one operation at a time; each operation must be completed without interruption once it starts; the order of operations for each job is predefined and cannot be modified; the setting up time of machines and transfer time of operations are negligible.

For illustrating explicitly, a sample instance of FJSP is shown in Table 1, where rows correspond to operations and columns correspond to machines. Each entry of the input table denotes the processing time of that operation on the corresponding machine. In this table, the tag "–" means that a machine cannot execute the corresponding operation.

Finally, we summarize some notations used mostly throughout this paper in Table 2.

Table 1
Processing time table of an instance of FJSP.

Job	Operation	M_1	M_2	M_3
J_1	$O_{1,1}$	2	–	3
	$O_{1,2}$	4	1	3
J_2	$O_{2,1}$	–	2	3
	$O_{2,2}$	6	2	4
	$O_{2,3}$	3	–	–
J_3	$O_{3,1}$	1	5	2
	$O_{3,2}$	3	–	2

3. Hybrid harmony search

3.1. Outline of HS

The harmony search (HS) [25] is one of the latest population-based evolutionary meta-heuristics. It is originally designed for the continuous optimization problem, which is defined as minimize (or maximize) $f(X)$ such that $x(j) \in [x_{\min}(j), x_{\max}(j)]$, where $f(X)$ is the objective function, $X = \{x(1), x(2), \ldots, x(D)\}$ is a candidate solution consisting of D decision variables, and $x_{\min}(j)$ and $x_{\max}(j)$ are the lower and upper bounds for each decision variable, respectively. To solve the problem, the HS maintains a harmony memory (HM) which consists of harmony vectors and can be represented as $HM = \{X_1, X_2, \ldots, X_{HMS}\}$, where HMS denotes the harmony memory size (HMS), and $X_i = \{x_i(1), x_i(2), \ldots, x_i(D)\}$ is the ith harmony vector in the HM. The best and worst harmony vectors in the HM are separately labeled as X_{best} and X_{worst}, respectively. The workflow of HS is simply described as follows. First, the initial harmony memory is generated from a uniform distribution in the ranges $[x_{\min}(j), x_{\max}(j)]$, where $1 \leq j \leq D$. Then, a new candidate harmony is generated from the HM based on three rules: memory consideration, pitch adjustment and random selection. In this paper, the modified pitch adjustment rule [26] is adopted, which can well inherit a good solution structure of X_{best} and make the algorithm have fewer parameters. The pseudocode of generating a new candidate harmony, the so-called "improvisation" in the HS, is depicted in Algorithm 1, where HMCR is the harmony memory considering rate (HMCR), PAR is the pitch adjusting rate (PAR), and $rand(0, 1)$ is a random function returning a real number between 0 and 1 with uniform distribution. Following the improvisation, the HM is updated by replacing the worst harmony in the HM with the newly generated harmony, but only if its fitness (measured in terms of the objective function) is better than that of the worst harmony. The procedures of improvising and updating are repeated until the termination criterion is satisfied. For more details about HS, refer to [25,27].

Algorithm 1. Pseudocode of the improvisation.

```
1:  for each j∈[1, D] do
2:     if rand(0, 1) < HMCR then              ▹ memory
                                                  consideration
3:        x_new(j)←x_i(j), where
          i∈{1, 2, …, HMS};
4:        if rand(0, 1) < PAR then            ▹ pitch adjustment
5:           x_new(j)←x_best(j);
6:        end if
7:     else                                    ▹ random selection
8:        x_new(j)←x_new(j)∈[x_min(j), x_max(j)];
9:     end if
10: end for
```

3.2. Procedure of HHS

The procedure of the proposed HHS algorithm is based on the HS and its algorithmic flow is depicted in Algorithm 2. Unlike the basic HS, a local search procedure is performed to improve the harmony vector generated in the improvisation phase for stressing exploitation, then the improved harmony vector enters into the evolutionary process to replace the original one. In addition, our HHS adopts the total number of improvisations (NI) as the stopping criterion. In other words, the HHS will be terminated when the NI is reached.

Algorithm 2. Algorithmic flow of the proposed HHS algorithm.

1: Set the algorithm parameters and the stopping criterion.
2: Initialize the HM.
3: Evaluate each harmony vector in the HM and label the X_{best} and X_{worst}.
4: **while** the stopping criterion is not met **do**
5:     Improvise a new harmony vector X_{new} from the HM.
6:     Perform the local search to X_{new} and yield X'_{new}.
7:     Update the HM.
8: **end while**
9: **return** the best harmony vector found.

Table 2
Summary of notations.

Notation	Description
J	The set of all jobs
M	The set of all machines
n	The total number of jobs
m	The total number of machines
J_i	The ith job
n_i	The number of operations of job J_i
M_k	The kth machine
$O_{i,j}$	The jth operation of job J_i
$M_{i,j}$	The set of alternative machines of operation $O_{i,j}$
$p_{i,j,k}$	The processing time of $O_{i,j}$ on machine M_k
C_{max}	The time needed to complete all jobs
d	The total number of all operations
$l(j)$	The number of alternative machines of operation j
$\sigma_{i,j}$	The start time of operation $O_{i,j}$ in the schedule
$\mu_{i,j}$	The selected machine of operation $O_{i,j}$ in the schedule
G	The schedule represented by the disjunctive graph
X_i	The ith harmony vector in the harmony memory
$x_i(j)$	The jth decision variable of harmony vector X_i
$x_{min}(j)$	The lower bound for the decision variable $x_i(j)$
$x_{max}(j)$	The upper bound for the decision variable $x_i(j)$
D	The dimension of harmony vector
$X_{i,1}$	The first half part of harmony vector X_i
$X_{i,2}$	The second half part of harmony vector X_i
X_{best}	The best harmony vector in the harmony memory
X_{worst}	The worst harmony vector in the harmony memory
X_{new}	The new harmony vector obtained through improvisation in harmony search
X'_{new}	The harmony vector obtained after local search
δ	The bound factor
R_i	The machine assignment vector
$r_i(j)$	The jth decision variable of machine assignment vector R_i
S_i	The operation sequence vector
$s_i(j)$	The jth decision variable of operation sequence vector S_i
Ω	The set of operations chosen to relax in large neighborhood search
Υ	The subset of Ω in which each operation is fixed on the original machine

3.3. Adaptation of HHS to the FJSP

As can be seen from Algorithm 2, there exist four issues to adapt the proposed HHS to the FJSP: representation of a harmony vector, initialization of the HM, evaluation of a harmony vector, and how to apply the local search to a harmony vector.

The HS algorithm works on the continuous domain, so the representation of a harmony vector (see Section 3.3.1) in the HHS is described in terms of continuous values to make more efficient use of the searching mechanism of HS. The HM is just initialized randomly and uniformly (see Section 3.3.1) to maintain the diversity.

To evaluate a harmony vector, we should map it to a schedule of the FJSP, then its fitness is given the value of makespan for this schedule. However, the kind of mapping is not straightforward, due to the continuity of the harmony vector and the discreteness of the schedule. Thus, a kind of discrete two vector code (see Section 3.3.2) is adopted as the bridge. When evaluating a harmony vector, it is firstly converted to a two vector code, called forward conversion (see Section 3.3.3) in this paper, then the two-vector code is further decoded to an active schedule, just as depicted in Fig. 1.

As for the local search to a harmony vector, its computational flow is depicted in Fig. 2. Indeed, the local search is not directly applied to a harmony vector, but to the schedule corresponding to the harmony vector, which is helpful for introducing the problem-specific knowledge. As shown in Fig. 2, when performing the local search to a harmony vector, the operator of evaluation is firstly used to obtain the corresponding schedule, then the schedule is further improved by the local search (see Section 3.3.4). Afterwards, the improved schedule is encoded to a two vector code, which is finally converted to a harmony vector (improved harmony vector) by using backward conversion (see Section 3.3.3).

In the following, we will detail the adaptation of the proposed HHS to the FJSP. Section 3.3.1 introduces the representation of a harmony vector and initialization of the HM. In Section 3.3.2, the two-vector is illustrated including its encoding and decoding methods. In Section 3.3.3, the conversion techniques, forward conversion and backward conversion, are presented. The local search strategy is described in Section 3.3.4.

3.3.1. Representation and initialization

In the proposed HHS, a harmony vector, $X_i = \{x_i(1), x_i(2), ..., x_i(D)\}$, is represented as a D-dimensional real vector. The dimension D satisfies the constraint $D = 2d$, where d is the total number of all operations in the FJSP to solve. The first half of the harmony vector $X_{i,1} = \{x_i(1), x_i(2), ..., x_i(d)\}$ describes the information of machine assignment for each operation, while the last half of the harmony vector $X_{i,2} = \{x_i(d+1), x_i(d+2), ..., x_i(2d)\}$ presents the information of operations sequencing on all the machines. This design can correspond well to the two-vector code for the FJSP. Moreover, to deal with the problem conveniently, the intervals $[x_{min}(j), x_{max}(j)]$, $j = 1, 2, ..., D$, are all set as $[-\delta, \delta]$, $\delta > 0$, where δ is referred to as the bound factor in this paper.

The population is initialized randomly and uniformly. A harmony vector, $X_i = \{x_i(1), x_i(2), ..., x_i(D)\}$, is randomly produced according to the following formula:

$$x_i(j) = -\delta + 2\delta \times rand(0, 1), \quad j = 1, 2, ..., D \quad (1)$$

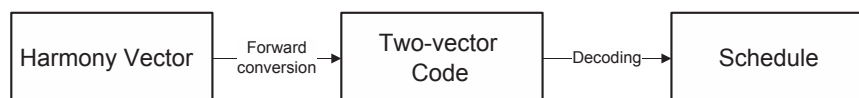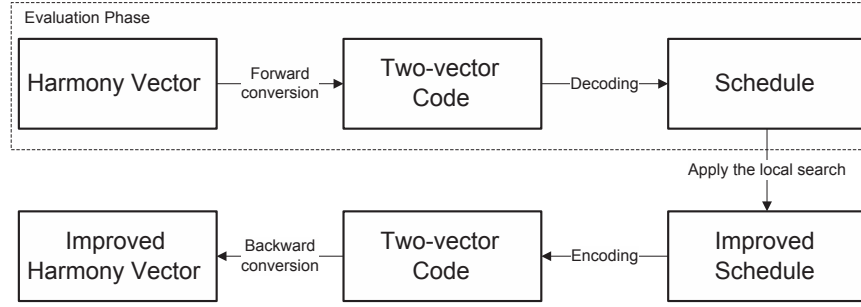**Fig. 1.** The computational flow of the evaluation.

Fig. 2. The computational flow of the local search to a harmony vector.

Table 3
Illustration of numbering scheme for operations.

Operation indicated	$O_{1,1}$	$O_{1,2}$	$O_{2,1}$	$O_{2,2}$	$O_{2,3}$	$O_{3,1}$	$O_{3,2}$
Fixed ID	1	2	3	4	5	6	7

3.3.2. Two-vector code

A two-vector code consists of two vectors: machine assignment vector and operation sequence vector, corresponding to two subproblems in the FJSP.

For explaining the two vectors, a fixed ID for each operation is first given in accordance with the job number and operation order within the job. This numbering scheme is illustrated in Table 3 for the instance shown in Table 1. After numbered, the operation can also be referred to by the fixed ID, for example, operation 6 has the same reference with the operation $O_{3,1}$ as shown in Table 3.

A machine assignment vector, denoted by $R_i = \{r_i(1), r_i(2), \ldots, r_i(d)\}$, is an array of d integer values. In the vector, $r_i(j)$, $1 \leq j \leq d$, represents the operation j chooses the $r_i(j)$th machine in its alternative machine set. For the problem in Table 1, a possible machine assignment vector is shown in Fig. 3 and its meaning is also revealed. For example, $r_i(1) = 2$ indicates that the operation $O_{1,1}$ chooses the second machine in its alternative machine set, that is machine M_3.

As for the operation sequence vector, expressed as $S_i = \{s_i(1), s_i(2), \ldots, s_i(d)\}$, it is the ID permutation of all the operations. The order of occurrence for each operation in S_i indicates its scheduling priority. Take the instance shown in Table 1 for example, a possible operation sequence vector is represented as $S_i = \{3, 1, 4, 2, 6, 5, 7\}$. The OS can be directly translated into a unique list of ordered operations: $O_{2,1} \succ O_{1,1} \succ O_{2,2} \succ O_{1,2} \succ O_{3,1} \succ O_{2,3} \succ O_{3,2}$. Operation $O_{2,1}$ has the highest priority and is scheduled first, then the operation $O_{1,1}$, and so on. It must be noted that not all the ID permutations are feasible for the operation sequence vector because of the designated priority of operations lying in an job. That is to say, the operations within a job should keep the relative priority order in S_i.

The decoding of the two-vector code is divided into two stages. The first step is to assign each operation to the selected machine according to R_i. Then the second is to treat all the operations one by one according to their order in S_i, each operation under treatment is allocated in the best available processing time for the corresponding machine. A schedule generated by this way can be guaranteed to be an active schedule [29]. To encode a schedule solution to a two-vector code is more direct, the vector R_i is obtained just through the machine assignment in the schedule, while the vector S_i is attained by sorting all the operations in the non-decreasing order of the earliest start time.

3.3.3. Conversion techniques

Conversions in the proposed HHS include two different types: forward conversion and backward conversion.
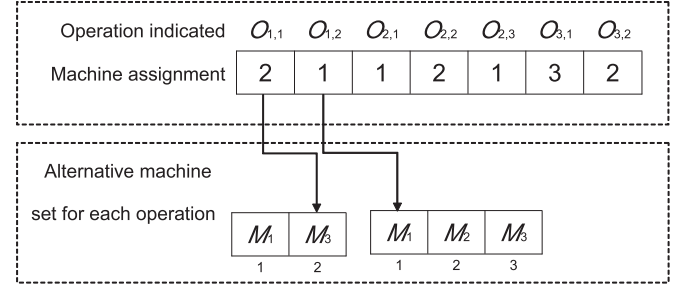

Fig. 3. Illustration of the machine assignment vector.

The forward conversion is to convert a harmony vector represented by the real-parameter vector $X_i = \{x_i(1), x_i(2), \ldots, x_i(d), x_i(d+1), x_i(d+2), \ldots, x_i(2d)\}$ to a two-vector code that consists of two integer-parameter vectors $R_i = \{r_i(1), r_i(2), \ldots, r_i(d)\}$ and $S_i = \{s_i(1), s_i(2), \ldots, s_i(d)\}$ and is divided into two separate parts. In the first part, we convert $X_{i,1} = \{x_i(1), x_i(2), \ldots, x_i(d)\}$ to the machine assignment vector $R_i = \{r_i(1), r_i(2), \ldots, r_i(d)\}$. Let $l(j)$ denote the number of alternative machines of operation j, where $j = 1, 2, \ldots, d$, and what the conversion needs to do is map the real number $x_i(j) \in [-\delta, \delta]$ to the integer $r_i(j) \in [1, l(j)]$. The concrete procedure is first convert $x_i(j)$ to a real number belong to $[1, l(j)]$ by linear transformation, then $r_i(j)$ is given the nearest integer value for the converted real number, which is shown in Eq. (2)

$$r_i(j) = round\left(\frac{1}{2\delta}(l(j)-1)(x_i(j)+\delta)+1\right), \quad j = 1, 2, \ldots, d \quad (2)$$

$round(x)$ is the function that rounds the number x to the nearest integer. In the second part, $X_{i,2} = \{x_i(d+1), x_i(d+2), \ldots, x_i(2d)\}$ is converted to the operation sequence vector $S_i = \{s_i(1), s_i(2), \ldots, s_i(d)\}$. To realize this transformation, the largest position value (LPV) rule [30] is used to construct an ID permutation of operations by ordering the operations in their non-increasing position value. However, as mentioned in Section 3.3.2, the obtained permutation may not be feasible for S_i. So, the repair procedure is further carried out to adjust the relative order of operations within a job in the permutation. Suppose that we have a vector $X_{i,2} = \{0.6, -0.4, 0.5, -0.2, 0.7, 0.3, -0.3\}$ for the instance in Table 1, then an example of conversion is illustrated in Fig. 4.

The backward conversion is to convert a two-vector code to a harmony vector, which also consists of two separate parts. In the first part related to machine assignment, the transformation is in fact an inverse linear transformation of Eq. (2). But the case of $l(j) = 1$ should be considered alone, and $x_i(j)$ should choose a random value between $[-\delta, \delta]$ when $l(j) = 1$. The transformation can be performed as follows:

$$x_i(j) = \begin{cases} \dfrac{2\delta}{l(j)-1}(r_i(j)-1)-\delta, & l(j) \neq 1 \\ x_i(j) \in [-\delta, \delta], & l(j) = 1 \end{cases} \quad (3)$$
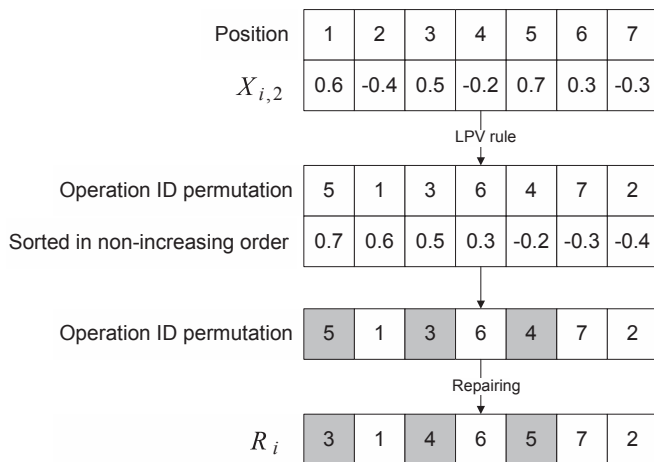
Fig. 4. The conversion from $X_{i,2}$ to the operation sequence vector.

where $j = 1, 2, \ldots, d$. For the second part, the vector $X_{i,2} = \{x_i(d + 1), x_i(d + 2), \ldots, x_i(2d)\}$ is obtained by rearranging elements in the old $X_{i,2}$ before improved. The rearrangement makes the new $X_{i,2}$ correspond to the operation sequence vector of the improved schedule according to the LPV rule.

3.3.4. Local search strategy

The local search is employed to enhance the local exploitation ability of HS, which is in fact applied to the schedule corresponding to the harmony vector just as depicted in Fig. 2.

A schedule of the FJSP could be represented by a disjunctive graph $G = (V, C \cup D)$ [8,28]. In the graph, V represents a set of all the nodes, each node denotes an operation in the FJSP (including dummy starting and terminating operations); C is the set of all the conjunctive arcs, these arcs connect two adjacent operations within one job and the directions of them represent the processing order between two connected operations; D means a set of all the disjunctive arcs, these arcs connect two adjacent operations performed on the same machine and their direction also show the processing order. The processing time for each operation is generally labeled above the corresponding node and regarded as the weight of the node. Take the problem shown in Table 1 for instance, a possible schedule represented by the disjunctive graph is illustrated in Fig. 5, in which $O_{3,1}$, $O_{2,3}$ are processed in succession on the machine M_1, $O_{2,1}$, $O_{1,2}$ are executed successively on the machine M_2, and $O_{1,1}$, $O_{2,2}$, $O_{3,2}$ are performed in turn on the machine M_3. The longest path from the starting node S to the ending node E is called the critical path, whose length defines the makespan for the schedule. Operations on the critical path are known as critical operations. In Fig. 5, the critical path is $S \rightarrow O_{1,1} \rightarrow O_{2,2} \rightarrow O_{2,3} \rightarrow E$ and the makespan equals 10, $O_{1,1}$, $O_{2,2}$ and $O_{2,3}$ are all critical operations.

Algorithm 3. Procedure of generating an acceptable neighborhood.

1:   Get the critical path in the current solution represented by the disjunctive graph G.
2: **for** each operation cp in the critical path **do**
3:     Delete cp from G to yield G'.
4:     **if** an available position is found for cp to insert into **then**
5:       Insert the operation cp to yield G''.
6:       **return** the disjunctive graph G''.
7:     **end if**
8: **end for**
9: **for** each operation cp in the critical path **do**
10:    Delete cp from G to yield G'.
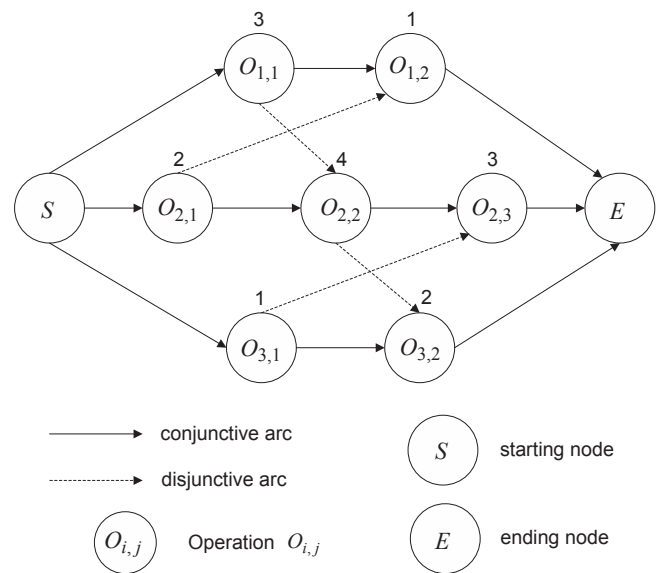11:   **for** each operation op in G' **do**

Fig. 5. Illustration of disjunctive graph.

12:      Delete op from G' to yield G''.
13:      **if** two available positions are found for cp and op to insert into **then**
14:        Insert cop and op to yield G'''.
15:        **return** the disjunctive graph G'''.
16:      **end if**
17:    **end for**
18: **end for**
19: **return** an empty disjunctive graph.

In the proposed HHS, a local search similar to the one used in [17] is adopted, in which the neighborhood of a schedule solution is obtained by moving one critical operation on a critical path in the disjunctive graph. The move of an operation is performed in two steps consisting of deletion and insertion, which means to delete an operation in the disjunctive graph, then insert it to an available position to yield no worse schedule. It is notable that the available position can be chosen on any alternative machine for an operation to be moved, so the machine assignment for this operation will be reallocated at the same time when moved to a different machine. Take the schedule shown in Fig. 5 for instance, when performing the local search on it, its critical path $S \rightarrow O_{1,1} \rightarrow O_{2,2} \rightarrow O_{2,3} \rightarrow E$ is first identified, then we try to move $O_{1,1}$, if $O_{1,1}$ is moved successfully (an available position is found for $O_{1,1}$ to insert into), an acceptable schedule is immediately found and is set as the current schedule to go on the next iteration of local improvement, otherwise $O_{2,2}$ is considered to be moved, until one operation on the critical path is moved successfully. If all the operations on the critical path fail to move, it indicates that the local search drops to local optimum of moving one critical operation.

In our local search procedure, the main difference lies in that when the local optimum of moving one critical operation is found, the current solution is further improved by moving two operations simultaneously, at least one of which is critical. Because moving two operations is more time consuming, it is only executed when it is failing to move one critical operation. In summary, the procedure of getting an acceptable neighborhood in the local search is described in Algorithm 3, where the keyword "return" within the loops means that the algorithm is terminated when it gets here and the content returned is regraded as the final result of the algorithm. For more details about how to move an operation,

refer to [17]. Our local search procedure is terminated when the maximal local iterations are met or all the specified moves fail.

4. Large neighborhood search

4.1. Outline of LNS

The large neighborhood search (LNS) [20] is a powerful technique, which combines constraint programming (CP) and local search to solve optimization problems. Unlike typical local search that makes small changes to the current solution, such as moving one or two operations in the scheduling, the LNS selects a subset of variables to relax from the problem. Once variables are chosen, unassign them while keeping the remaining variables fixed (destruction), and then search for a better solution by re-optimizing only the unassigned variables (construction). Steps of destruction and construction are iterated in the LNS until the termination condition is met. The basic architecture of LNS is outlined in Algorithm 4.

Algorithm 4. LNS architecture.

1:	Produce initial solution.
2:	**while** termination condition is not met **do**
3:	Choose a subset of variables to relax.
4:	Fix the remaining variables.
5:	**if** search finds improvement **then**
6:	Update current solution.
7:	**end if**
8:	**end while**

The main idea of LNS is simple in fact. Through the operator of destruction, the original problem is reduced, then the CP is employed to solve the reduced problem which can overcome the defect of CP existing in exploring the large search space. The key benefit from a CP perspective is that the strong CP propagation techniques can be exploited to prune the search space more effectively when compared with the pure tree search [31].

Our LNS is implemented on top of the COMET [32], an advanced optimization system that has been gradually adopted in the operations research. When using LNS in COMET, the user should first model the problem to be solved by establishing constraints; the additional constraints will be added continuously during Steps 4 and 5 in Algorithm 4; once a constraint has been posted, the CP propagation provided inside the COMET system will be triggered, which means that all the constraints posted until then, will participate in the filtering of the domains, one constraint after the other, until no more values can be removed from any of the domains. Moreover, every time a solution is found during the search, a constraint is dynamically added by the system, stating that the next solution found must be better. For more details about the working mechanism of the COMET and how it can be used to solve various combinatorial optimization applications, refer to [33]. Below, we will describe the implementation of our LNS for the FJSP based on the COMET, including the constraint-based model, destruction procedure and construction procedure.

4.2. Constraint-based model for the FJSP

To illustrate the constraint-based model for the FJSP, two more notations are defined based on Section 2. Let $\sigma_{i,j}$, $\mu_{i,j}$ denote the start time and the selected machine of operation $O_{i,j}$ in the schedule, respectively. Then, a solution to the FJSP is composed of pairs of values $(\sigma_{i,j}, \mu_{i,j})$ for all the operations. The solution is feasible if it satisfies the following three kinds of constraints:

- Precedence constraint: the operation within a job must satisfy the designated priority order. It is formulated as follows: $\forall i \in [1, n], \forall j \in [1, n_i - 1], \sigma_{i,j} + p_{i,j,\mu_{i,j}} \leq \sigma_{i,j+1}$.
- Resource constraint: one operation can only be processed on its available machines, that is $\forall O_{i,j}, \mu_{i,j} \in M_{i,j}$.
- Capacity constraint: the machine can only process one operation at a time, which is formulated like this: $\forall O_{x,y}, O_{\alpha,\beta}$, if $\mu_{x,y} = \mu_{\alpha,\beta}$, then $\sigma_{x,y} + p_{x,y,\mu_{x,y}} \leq \sigma_{\alpha,\beta}$ or $\sigma_{\alpha,\beta} + p_{\alpha,\beta,\mu_{\alpha,\beta}} \leq \sigma_{x,y}$.

The FJSP is to minimize the makespan $C_{\max} = \max_{1 \leq i \leq n, 1 \leq j \leq n_i} \{\sigma_{i,j} + p_{i,j,\mu_{i,j}}\}$ under the above three constraints.

4.3. Destruction procedure

In the destruction procedure, some variables are chosen to relax while the others are kept fixed. For the FJSP, the partial-order schedule (POS) relaxation [34] is adopted, which chooses a set Ω of operations first, then each of the remaining operations is fixed on the current machine and operations executed on the same machine are kept their relative precedence order in the current schedule. Let (σ, μ) is the current schedule solution and (σ', μ') is the one to be constructed. The POS relaxation can be formulated as follows: $\forall O_{x,y}, O_{\alpha,\beta} \notin R$ and $\mu_{x,y} = \mu_{\alpha,\beta}$, if $\sigma_{x,y} + p_{x,y,\mu_{x,y}} \leq \sigma_{\alpha,\beta}$, then $\sigma'_{x,y} + p_{x,y,\mu_{x,y}} \leq \sigma'_{\alpha,\beta}$ and $\mu'_{x,y} = \mu_{x,y}$, $\mu'_{\alpha,\beta} = \mu_{\alpha,\beta}$. It is beneficial for the POS relaxation to fix only the relative precedence order between the remaining operations, rather than the actual start time, which leaves more room for the re-optimization.

Let Υ be the subset of Ω where each operation is fixed on the original machine, then how to choose the set Υ is called the neighborhood heuristic in LNS. In this paper, the time-window neighborhood heuristic is adopted, which generates the time window $[t_{\min}, t_{\max}]$ randomly and Υ is the set of all operations processed between the interval $[t_{\min}, t_{\max}]$. To further intensify the search, an additional neighborhood is constructed from the time-window neighborhood by selecting a subset $\Upsilon \subseteq \Omega$ and the machine assignment of operations in Υ is fixed, that is $\forall O_{i,j} \in \Upsilon, \mu'_{i,j} = \mu_{i,j}$.

4.4. Construction procedure

The construction procedure is to form the new schedule solution based on the current one by using CP search. Our search method is simple and direct, which relies on the fact that once each operation is assigned to an available machine and operations within the same machine are completely ranked under constraints, the minimal makespan is equal to the sum of durations of the operations along the longest path of the precedence graph (disjunctive graph). Briefly, the search is performed as follows: assign operations to the chosen machines, considering the operations with the fewest machine selections first; and then rank operations on each machine separately, giving priority to those machines with the least slack; following the rank, the earliest and latest start time for each operation and the makespan are calculated according to the precedence graph; finally set the earliest start time as the start time for all operations. To avoid searching for too long, the failure limit is set for each construction procedure of LNS. Moreover, the deep first search is used as our search controller, and because the search is implemented on the constraint-based system, it can well inherit the benefit of constraint propagation in CP.

5. Integrated search heuristic: HHS/LNS

Given the complexities of the two existing algorithm modules, our method to integration in this paper is relatively naive. The HHS

module is executed at first. By the mechanism of HS that replaces the worst harmony in the HM at each iteration, the HM can also be regarded as an elite solution pool at the end of running the HHS algorithm (NI is reached). Because the assignment of machines to operations is critical for the FJSP, we can extract some good machine assignment information from the elite solutions in the HM before entering the LNS module. The procedure of extraction works as follows: for each operation, the machine selected by the best harmony in the HM is added to its available machine list, while the machine among the remaining that selected most frequently by the other elite solutions is added if its selected frequency is no less than τ, where τ is the adjustable parameter. So, each operation has a reduced set of machines to chose from in the extraction. Following the extraction, the best solution found by the HHS is used as the initial solution for the LNS, the LNS is then run for a specified CPU time to further improve it and return the best solution found.

As described above, our integration features two folds: first, the best solution found by the HHS provides a good starting point for the LNS and second, the extracted machine assignment information restricts the search of LNS to a more promising problem space, which could strengthen the ability of intensification.

Obviously, there is much more we could do. However, this simple integration directly and concisely achieves our motivations for this research, just as described in Section 1.

6. Experimental study

6.1. Experimental setup

To test the performance of the proposed algorithms, the HHS module is implemented in Java, while the LNS module is implemented in the COMET [32] language. Both algorithms are run on an Intel 2.83 GHz Xeon processor with 15.9 GB of RAM.

In our experiment, two well-known FJSP benchmark sets are mainly involved. One is the BRdata set of FJSP instances from Brandimarte [8], which is used to validate the effectiveness of the proposed HHS algorithm. The other is a set of harder and larger instances provided by Dauzére-Pérés and Paulli [10], referred as DPdata, which is adopted to illustrate the effectiveness of the HHS/LNS for solving hard and large-scale FJSP problems. Moreover, to have an overall evaluation of our proposed algorithms, we also summarize the results on another two benchmark data sets in the literature, which are BCdata set from Barnes and Chambers [35] and HUdata set from Hurink et al. [9]. Among them, HUdata is divided into three subsets: Edata, Rdata and Vdata. The best lower bound (LB) and upper bound (UB) of each benchmark instance quoted in this paper are taken from [36].

Due to the natural nondeterminacy of the proposed algorithms, we carry out five runs on each problem instance in order to obtain meaningful results, just as some existing literatures suggested [11,13,14]. Four metrics including the best makespan ($BC_{\max}$), the average makespan ($AV(C_{\max})$), the standard deviation of makespan (SD), and the average computational time in seconds ($AV(CPU)$) obtained among five runs are applied to describe the computational results.

To show the effectiveness, we compare the results obtained by the proposed algorithm with the existing algorithms in the literature, and $BC_{\max}$ and $AV(C_{\max})$ are used as the main comparison metrics. Relative deviation criterion represented by (dev) is employed for the comparison of the makespan, which is defined as

$$dev = [(MK_{comp} - MK_{proposed})/MK_{comp})] \times 100\% \qquad (4)$$

where $MK_{proposed}$ and MK_{comp} are the makespan values obtained by our method and the comparative algorithm, respectively.

The parameters in the HHS algorithm module include harmony memory size (HMS), harmony memory considering rate (HMCR), pitch adjusting rate (PAR), the bound factor (δ), the total number of improvisations (NI), and the maximum iterations of local search ($iter_{\max}$). The LNS algorithm module has three parameters: the failure limit for each construction procedure (maxFail), the probability of the operation in Ω belonging to Υ (P_l), and the maximum CPU time limit ($T_{\max}$). For the integrated method HHS/LNS, there is another parameter τ that is mentioned in Section 5 besides the parameters in the HHS and LNS. The parameters will be specified for the corresponding experiment, which are set in such a way that a relatively good trade-off between solution quality and computational time can be obtained.

6.2. Performance analysis of the HHS module

In this section, the performance of the HHS algorithm is analyzed. First, BRdata is investigated to validate the effectiveness of our proposed HHS. The parameters of the HHS algorithm for this set of instances are given in Table 4.

Our computational results for the HHS on BRdata are reported in Table 5. The first and second columns include the name and size of the instance, respectively. The third column lists the total number of all operations for the instance. In the fourth column, the average number of alternative machines for each operation is shown for each instance, which is called "flexibility" in the FJSP. In the fifth column, (LB, UB) stand for the best lower and upper bounds of the instance, respectively. The following columns describe the four metrics mentioned in Section 6.1.

In Table 6, the HHS algorithm is compared with several recently proposed evolutionary-based algorithms, including GA of Pezzella et al. [13], KBACO of Xing et al. [16], AIA of Bagheri et al. [15], and ABC of Wang et al. [17]. The $AV(C_{\max})$ of GA and AIA are not available. In Table 7, we compare our HHS algorithm with other categories of techniques, including TS of Mastrolilli and Gambardella [11], and CDDS of Hmida et al. [22]. The bold values indicate that the corresponding algorithm found to be the best among the referred approaches. dev is used for the comparing of the best

Table 4
Parameters setting for the HHS on BRdata.

Parameter	Description	Value
HMS	Harmony memory size	5
HMCR	Harmony memory considering rate	0.95
PAR	Pitch adjusting rate	0.3
δ	Bound factor	1.0
NI	The total number of improvisations	3000
$iter_{\max}$	The maximum iterations of local search	150

Table 5
Results of HHS on BRdata.

Instance	$n \times m$	d	Flex	(LB, UB)	HHS			
					$BC_{\max}$	$AV(C_{\max})$	SD	$AV(CPU)$
MK01	10×6	55	2.09	(36, 42)	40	40	0	3.87
MK02	10×6	58	4.10	(24, 32)	26	26.2	0.45	5.79
MK03	15×8	150	3.01	(204, 211)	204	204	0	36.60
MK04	15×8	90	1.91	(48, 81)	60	60	0	13.30
MK05	15×4	106	1.71	(168, 186)	172	172.8	0.45	35.78
MK06	10×15	150	3.27	(33, 86)	59	59.4	0.45	111.65
MK07	20×5	100	2.83	(133, 157)	139	139.8	0.45	26.16
MK08	20×10	225	1.43	523	523	523	0	171.10
MK09	20×10	240	2.53	(299, 369)	307	307	0	172.24
MK10	20×15	240	2.98	(165, 296)	202	203	1.00	437.69

Table 6
Comparison of the proposed HHS with four existing evolutionary-based algorithms on BRdata.

| Instance	HHS		GA		KBACO			AIA		ABC		
	BC_{max}	$AV(C_{max})$	BC_{max}	$dev(\%)$	BC_{max}	$AV(C_{max})$	$dev(\%)$	BC_{max}	$dev(\%)$	BC_{max}	$AV(C_{max})$	$dev(\%)$
MK01	40	40	40	0	**39**	39.8	−2.56	40	0	40	40	0
MK02	**26**	26.2	**26**	0	29	29.1	+10.34	**26**	0	**26**	26.5	0
MK03	**204**	204	**204**	0	**204**	204	0	**204**	0	**204**	204	0
MK04	**60**	60	**60**	0	65	66.1	+7.69	**60**	0	**60**	61.22	0
MK05	**172**	172.8	173	+0.58	173	173.8	+0.58	173	+0.58	**172**	172.98	0
MK06	**59**	59.4	63	+6.35	67	69.1	+11.94	63	+6.35	60	64.48	+1.67
MK07	**139**	139.8	**139**	0	144	145.4	+3.47	140	+0.71	**139**	141.42	0
MK08	**523**	523	**523**	0	**523**	523	0	**523**	0	**523**	523	0
MK09	**307**	307	311	+1.29	311	312.2	+1.29	312	+1.60	**307**	308.76	0
MK10	**202**	203	212	+4.72	229	233.7	+11.79	214	+5.61	208	212.84	+2.88
Average improvement				+1.29			+4.45		+2.09			+0.46

Table 7
Comparison of the proposed HHS with other categories of techniques (TS, CDDS) on BRdata.

Instance	HHS		TS			CDDS							
	BC_{max}	$AV(C_{max})$	BC_{max}	$AV(C_{max})$	$dev(\%)$	N_1	$dev(\%)$	N_2	$dev(\%)$	N_3	$dev(\%)$	N_4	$dev(\%)$
MK01	**40**	40	**40**	40	0	**40**	0	**40**	0	**40**	0	**40**	0
MK02	**26**	26.2	**26**	26	0	**26**	−0.77	**26**	−0.77	**26**	−0.77	**26**	−0.77
MK03	204	204	204	204	0	204	0	204	0	204	0	204	0
MK04	60	60	**60**	60	0	**60**	0	**60**	0	**60**	0	**60**	0
MK05	**172**	172.8	173	173	+0.58	175	+1.26	173	+0.12	173	+0.12	173	+0.12
MK06	59	59.4	**58**	58.4	−1.72	60	+1.00	59	−0.68	59	−0.68	**58**	−2.41
MK07	**139**	139.8	144	147	+3.47	**139**	−0.58	**139**	−0.58	**139**	−0.58	**139**	−0.58
MK08	523	523	523	523	0	523	0	523	0	523	0	523	0
MK09	307	307	**307**	307	307	**307**	0	**307**	0	**307**	0	**307**	0
MK10	202	203	198	199.2	−2.02	198	−2.53	197	−3.05	198	−2.53	198	−2.53
Average improvement					+0.03		−0.16		−0.50		−0.44		−0.62

makespan obtained between the HHS and other methods except for the CDDS. Different from the other aforesaid algorithms, the CDDS is deterministic by nature, and it is not a single algorithm. In fact, it consists of a group of four deterministic algorithms (CDDS-N_1, CDDS-N_2, CDDS-N_3, and CDDS-N_4). So, to make a fairer comparing, dev for the CDDS-N_1 (or CDDS-N_2, CDDS-N_3, CDDS-N_4) is computed by comparing HHS against CDDS-N_1 (or CDDS-N_2, CDDS-N_3, CDDS-N_4) based upon the average performance of HHS instead of using the best performance.

As can be seen from Table 6, the proposed HHS compares favorably with the other evolutionary-based algorithms on the BRdata. In fact, the HHS outperforms GA in 4 out of 10 instances, outperforms KBACO in 7 out of 10 instances, outperforms AIA in 5 out of 10 instances, and outperforms ABC in 2 out of 10 instances, in terms of BC_{max}. For the instances MK06 and MK10, the HHS obtains a better solution than any of them. Measures concerning average relative deviation also reveal that all the four existing evolutionary-based algorithms are dominated by the proposed HHS. From Table 7, the HHS is also comparable with TS and CDDS. In particular, the HHS obtains 8 best BC_{max} out of 10 instances among the three algorithms. Considering the average relative deviation, the HHS is a little better than the TS, but is slightly worse than all the four CDDS algorithms.

From Table 5, the HHS not only shows strong ability but also stability and efficiency. Half of the instances are solved with the SD is equal to 0, while the rest with small SD values. Moreover, all 10 instances are computed in no more than 3 min except for the instance MK10. Based on the simulation tests and comparison study, it is safe to conclude that the HHS is enough for solving some medium to large FJSP instances effectively, efficiently, and robustly.
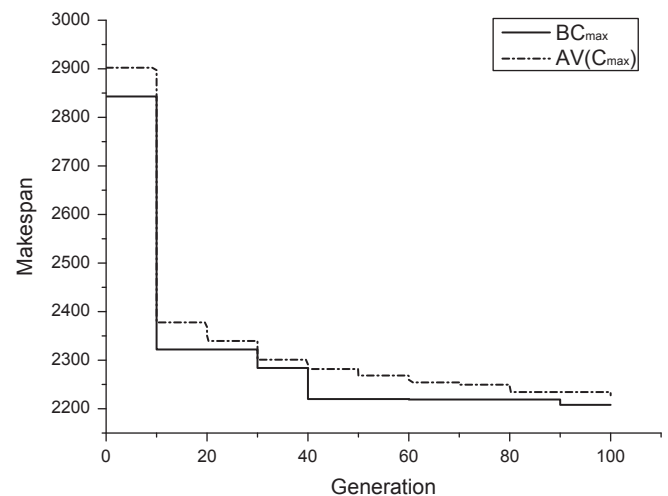
Fig. 6. Convergence curves in solving the instance 08a by the proposed HHS.

To further illustrate the property of the HHS, a large-scale FJSP instance (08a) in DPdata is used. The parameter of HMS is set as 8, others are set according to Table 4. In Fig. 6, the typical convergence curves obtained by the HHS when solving the instance 08a are depicted. From Fig. 6, it can be seen that the decreasing of the makespan is very fast during the initial 10 generations, but the speed gets much slower in the following generations. In other words, the HHS could generate high quality solutions in related short time, but the solutions are harder to be further improved with the evolution procedure in progress.

Table 8
Performance of the HHS on the instance 08a under different parameter settings.

Experiment number	$iter_{max}$	NI = 1000			NI = 3000			NI = 5000		
		BC_{max}	$AV(C_{max})$	AV(CPU)	BC_{max}	$AV(C_{max})$	AV(CPU)	BC_{max}	$AV(C_{max})$	AV(CPU)
1	150	2136	2166	47.83	2124	2142.6	176.09	2124	2132.2	279.22
2	300	2086	2090.2	479.99	2082	2087	1455.79	2082	2085	2386.80
3	450	2086	2088.2	483.28	2080	2086.4	1460.77	2080	2083.2	2434.54

In Table 8, the performance of the HHS on the instance 08a under different parameter settings is listed. Here, we only adjust the parameters $iter_{max}$ and NI, which are important for the quality of solutions. Other parameters are kept fixed. From Table 8, it can be seen that the parameter $iter_{max}$ imposes more impact on the final solution. The makespan values obtained in Experiment 2 is much better than the ones obtained in Experiment 1, but the computation time is much longer. There is little difference between the results in Experiment 2 and Experiment 3 and this is because the local iterated times 300 is big enough for almost every local search to find the local optima. Thus, to further increase the $iter_{max}$ at this time is meaningless to the solution. The enlargement of the parameter NI can also improve the quality of the solution, but the improvement is more limited compared with the increase of $iter_{max}$, and there seems to be no effect to increase NI further as it reaches a certain value. In all the three experiments, BC_{max} remains the same when NI increases from 3000 to 5000. Our computational results in Table 8 also indicate that it is quite difficult to improve the high quality solution further by the proposed HHS, no matter through increasing the parameter NI or $iter_{max}$.

### 6.3. Performance analysis of the LNS module

In this section, we will analyze the performance of the LNS module. Three large-scale FJSP instances (07a, 08a, 09a) with the increasing problem space in DPdata are used for this investigation. First, the LNS is run directly on the three instances, where the parameters $maxFail = 200, P_l = 0.33, T_{max} = 2000$ s. The detailed results are given in Table 9. The first column shows the name of the instance. In the second column, the initial solution of the LNS for each instance is listed, which is the first feasible solution found by the CP-based search. The following four columns record the minimum makespan obtained by the LNS every 500 s. The last column represents the best known solution in the literature for each instance. From Table 9, it can be seen that the initial solution for each instance is poorer with the size of the problem space increasing, the initial solutions of instances 07a, 08a, 09a are about 35.5%, 124.2%, 308.8%, respectively, from the corresponding best known ones. As for the final solutions obtained after 2000 s, the situation is similar, only the solution for the instance 07a is relatively acceptable, which is about 1.9% from the best known ones. The other two for instances 08a and 09a are quite poor, which are about 13.8% and 136.6% from the best known ones. And the poor quality of final solutions for instances 08a and 09a is partly due to their poor initial solutions, so it is necessary to provide a good initial solution for the LNS. Moreover, for every instance, the makespan decreases with the computation time increasing, but the changes of decreasing speed does not appear with certain regularity.

In the following, we will observe the decreasing speed of the makespan when solving instances with different problem spaces by the LNS. In Fig. 7, we draw the decrease of the makespan for instances 07a, 08a, 09a. To have a fair comparison, the starting solutions of decreasing for the three instances are all set as 10%

Table 9
Performance of the LNS on instances 07a, 08a, 09a.

Instance	Initial	Time (s)				BKS
		500	1000	1500	2000	
07a	3094	2370	2357	2347	2327	2283
08a	4639	2893	2515	2409	2355	2069
09a	8447	7045	6561	6058	4888	2066
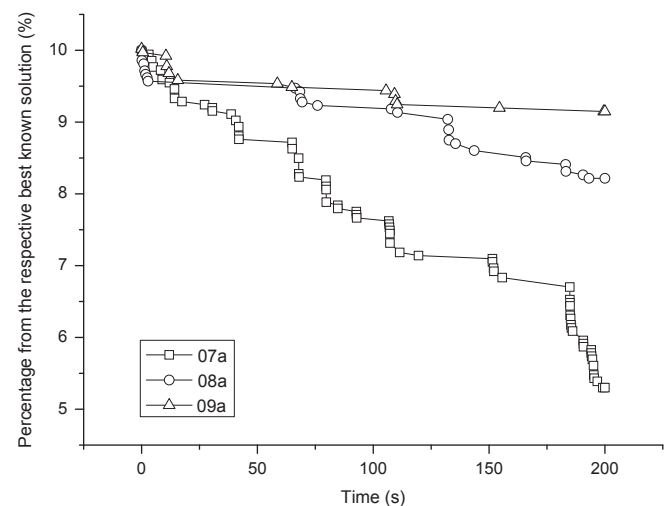
Fig. 7. Decreasing of the makespan in solving instances 07a, 08a, 09a by the LNS.

from their best known solutions. From Fig. 7, it can be seen that the overall decreasing speed of the makespan for the instance 07a is the fastest, while for the instance 09a is the slowest. It seems that the LNS generally shows a stronger ability of intensification during search when tackling the instance with smaller problem space.

6.4. Effects of integration

Based on the above performance analysis, the effects of integration are foreseeable. First, the experiments in Section 6.2 indicate that the solution is hard to improve further when the evolution procedure of the HHS reaches a certain level. Second, Section 6.3 reveals that the initial solution is one of the important factors when solving some large-scale problems and the LNS could show stronger search ability on the smaller problem space. In addition, the results in Table 9 show that the pure LNS algorithm is not an ideal optimization tool for the FJSP. So, the integration presented in Section 5 is necessary and reasonable, which overcomes the disadvantages of both algorithms.

In Fig. 8, the impact of the machine assignment information extraction is illustrated, where the instance 08a is used. The curve

with circle points depicts the decreasing of the makespan by executing the LNS directly on the initial solution provided by the HHS, while the curve with triangular points describes the case after the procedure of extraction. From Fig. 8, it can be seen that the LNS could effectively improve the high quality solution obtained by the HHS, and the clear benefit of the extraction is also displayed.
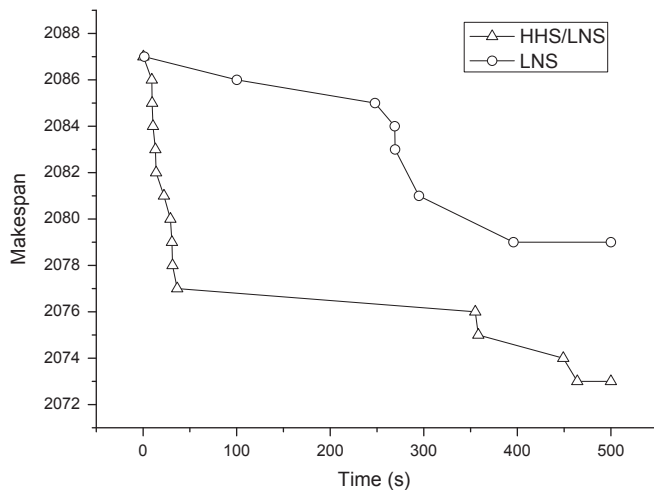

Fig. 8. The impact of the machine assignment information extraction.

Table 10
Parameters setting for the HHS/LNS on DPdata.

Parameter	Description	Value
HMS	Harmony memory size	8
HMCR	Harmony memory considering rate	0.95
PAR	Pitch adjusting rate	0.3
NI	The total number of improvisations	3000
δ	Bound factor	1.0
$iter_{max}$	The maximum iterations of local search	300
maxFail	The failure limit for each construction procedure	200
P_l	The probability of the operation in Ω belonging to Υ (P_l)	0.33
τ	The limit of selected frequency	3

6.5. Computational results on large-scale benchmark instances

In this section, we will demonstrate the effectiveness of our proposed integrated search heuristic HHS/LNS for solving large-scale FJSP problems. So, the HHS/LNS is evaluated on the instances in DPdata, which is one of the most hardest and largest benchmark data sets in the FJSP literature. The common parameters of the HHS/LNS algorithm for this set of instances are shown in Table 10. The maximum CPU time limit for the LNS in the HHS/LNS (T_{max}) is set as 500 s for the first 12 instances (01a–12a), while twice for the remaining ones.

Table 11 displays the detailed computational results for the instances considered in DPdata. Columns 1–5 correspond to the same quantities of Table 5. In columns 6–8, the original results obtained by the HHS are firstly presented. The results obtained by running the LNS directly on DPdata instances are listed in columns 9–11 to have a contrast with the results of the HHS/LNS, where the maximum CPU time limit of the LNS is extended to 2000 and 3000 s for the instances 01a–12a and instances 13a–18a, respectively. In the last four columns, four metrics mentioned in Section 6.1 for the HHS/LNS are described in detail. In Table 11, the HHS/LNS is compared with three state-of-the-art algorithms, which are TS of Mastrolilli and Gambardella [11], hGA of Gao et al. [14] and CDDS of Hmida et al. [22]. BC_{max} and $AV(C_{max})$ denote the best and average makespan obtained among the five independent runs, respectively, for the algorithms HHS/LNS, TS and hGA. The CDDS are deterministic algorithms, so we compare the $AV(C_{max})$ of HHS/LNS with the results of CDDS by using the metric dev for making a fairer comparison, just as mentioned in Section 6.2.

From Table 12, it can be seen that our results are quite competitive with state-of-the-art algorithms. The proposed HHS/LNS have a generally superior performance in terms of BC_{max} and $AV(C_{max})$ compared with TS, hGA and CDDS. As for the BC_{max}, the HHS/LNS outperforms TS in 12 out of 18 instances, outperforms hGA in 15 out of 18 instances, and outperforms CDDS-N_1, CDDS-N_2, CDDS-N_3, and CDDS-N_4 in 15, 14, 16, and 14 out of 18 instances, respectively, the HHS/LNS does not yield the best only for 6 instances, but the results are slightly worse, which are in average 0.2% from the best ones. Overall, the integrated search heuristic HHS/LNS outperforms TS and hGA by 0.09% and 0.23% on total 18 instances in terms of average dev, respectively. From the average dev for the CDDS, our HHS/LNS is better than any of four CDDS algorithms even if we compare the $AV(C_{max})$ of the HHS/LNS with

Table 11
Results on DPdata instances.

Instance	$n \times m$	d	Flex	(LB, UB)	HHS			LNS			HHS/LNS			
					BC_{max}	$AV(C_{max})$	SD	BC_{max}	$AV(C_{max})$	SD	BC_{max}	$AV(C_{max})$	SD	AV(CPU)
01a	10×5	196	1.13	(2505, 2530)	2525	2534	6.28	2554	2560	8.25	2505	2512.8	7.12	837.6
02a	10×5	196	1.69	(2228, 2244)	2242	2245.4	2.07	2233	2235.8	3.27	2230	2231.2	1.64	972.6
03a	10×5	196	2.56	(2228, 2235)	2229	2231.2	1.48	2230	2231.2	2.08	2228	2229	1.00	1164.6
04a	10×5	196	1.13	(2503, 2565)	2506	2517	8.34	2506	2522.4	25.81	2506	2506	0	849.6
05a	10×5	196	1.69	(2189, 2229)	2232	2234.6	2.41	2219	2222	3.94	2212	2215.2	2.39	931.2
06a	10×5	196	2.56	(2162, 2216)	2201	2206.8	3.77	2213	2215.6	1.82	2187	2191.8	2.77	1167.0
07a	15×8	293	1.24	(2187, 2408)	2323	2340.2	12.28	2327	2358.8	23.34	2288	2303	11.46	1547.4
08a	15×8	293	2.42	(2061, 2093)	2086	2087.6	1.52	2226	2278.6	73.08	2067	2073.8	5.16	1905.6
09a	15×8	293	4.03	(2061, 2074)	2074	2079	3.32	4004	4503	445.40	2069	2072.8	3.56	943.2
10a	15×8	293	1.24	(2178, 2362)	2341	2346.4	5.03	2357	2390.4	18.77	2297	2302.2	7.19	1590.0
11a	15×8	293	2.42	(2017, 2078)	2077	2079.8	2.77	2221	2289.2	55.92	2061	2066.6	3.28	1826.4
12a	15×8	293	4.03	(1969, 2047)	2045	2052.6	9.81	3765	4639	637.83	2027	2035.6	5.81	914.4
13a	20×10	387	1.34	(2161, 2302)	2280	2291.2	6.94	2365	2374.2	13.74	2263	2269.4	4.39	2900.3
14a	20×10	387	2.99	(2161, 2183)	2194	2195.2	1.10	3585	3743.2	162.67	2164	2167.6	2.07	3237.5
15a	20×10	387	5.02	(2161, 2171)	2220	2230.2	6.72	10,081	10,324.8	466.59	2163	2166.2	3.03	2112.3
16a	20×10	387	1.34	(2148, 2301)	2285	2289	5.10	2352	2372	15.76	2259	2266.4	5.46	2802.2
17a	20×10	387	2.99	(2088, 2169)	2160	2166.6	4.93	3568	3658	134.82	2137	2141.2	2.95	3096.4
18a	20×10	387	5.02	(2057, 2139)	2159	2170.2	6.53	10,036	10,683.8	594.23	2124	2128	3.87	2489.2

Table 12
Comparison between the proposed HHS/LNS and state-of-the-art algorithms on DPdata.

Instance	HHS/LNS		TS			hGA			CDDS								
	BC_{max}	$AV(C_{max})$	BC_{max}	$AV(C_{max})$	$dev(\%)$	BC_{max}	$AV(C_{max})$	$dev(\%)$	N_1	$dev(\%)$	N_2	$dev(\%)$	N_3	$dev(\%)$	N_4	$dev(\%)$
01a	**2505**	2512.8	2518	2528	+0.52	2518	2518	+0.52	2518	+0.21	2530	+0.68	2530	+0.68	2520	+0.29
02a	**2230**	2231.2	2231	2234	+0.04	2231	2231	+0.04	2231	−0.01	2244	+0.57	2232	+0.04	2231	−0.01
03a	**2228**	2229	2229	2229.6	+0.04	2229	2229.3	+0.04	2229	0	2235	+0.27	2230	+0.04	2233	+0.18
04a	2506	2506	**2503**	2516.2	−0.12	2515	2518	+0.36	2510	+0.16	2520	+0.56	2507	+0.04	**2503**	−0.12
05a	**2212**	2215.2	2216	2220	+0.18	2217	2218	+0.23	2220	+0.22	2219	+0.17	2216	+0.04	2217	+0.08
06a	**2187**	2191.8	2203	2206.4	+0.73	2196	2198	+0.41	2199	+0.33	2214	+1.00	2201	+0.42	2196	+0.19
07a	2288	2303	**2283**	2297.6	−0.22	2307	2309.8	+0.82	2299	−0.17	**2283**	−0.88	2293	−0.44	2307	+0.17
08a	**2067**	2073.8	2069	2071.4	+0.10	2073	2076	+0.29	2069	−0.23	2069	−0.23	2069	−0.23	2069	−0.23
09a	2069	2072.8	**2066**	2067.4	−0.10	**2066**	2067	−0.10	2069	−0.18	**2066**	−0.33	**2066**	−0.33	**2066**	−0.33
10a	2297	2302.2	**2291**	2305.6	−0.74	2315	2315.2	+0.78	2301	−0.05	**2291**	−0.49	2307	+0.21	2311	+0.38
11a	**2061**	2066.6	2063	2065.6	+0.10	2071	2072	+0.48	2078	+0.55	2069	+0.12	2078	+0.55	2063	−0.17
12a	**2027**	2035.6	2034	2038	+0.34	2030	2030.6	+0.15	2034	−0.08	2031	−0.23	2040	+0.22	2031	−0.23
13a	2263	2269.4	2260	2266.2	−0.13	**2257**	2260	−0.27	**2257**	−0.55	2265	−0.19	2260	−0.42	2259	−0.46
14a	**2164**	2167.6	2167	2168	+0.14	2167	2167.6	+0.14	2167	−0.03	2189	+0.98	2183	+0.71	2176	+0.39
15a	**2163**	2166.2	2167	2167.2	+0.18	2165	2165.4	+0.09	2167	+0.04	2165	−0.06	2178	+0.54	2171	+0.22
16a	2259	2266.4	**2255**	2258.8	−0.18	2256	2258	−0.13	2259	−0.33	2256	−0.46	2260	−0.28	2256	−0.46
17a	**2137**	2141.2	2141	2144	+0.19	2140	2142	+0.14	2143	+0.08	2140	−0.06	2156	+0.69	2143	+0.08
18a	**2124**	2128	2137	2140.2	+0.61	2127	2130.7	+0.14	2137	+0.42	2127	−0.05	2131	+0.14	2131	+0.14
Average improvement					+0.09			+0.23		+0.02		+0.08		+0.15		+0.01

Table 13
The makespan of new best-known solutions identified by the proposed HHS/LNS on DPdata.

Instance	(LB, UB)	Prev. best known	New best known
01a	(2505, 2530)	2518	2505
02a	(2228, 2244)	2231	2230
03a	(2228, 2235)	2229	2228
05a	(2189, 2229)	2216	2212
06a	(2162, 2216)	2196	2187
08a	(2061, 2093)	2069	2067
11a	(2017, 2078)	2063	2061
12a	(1969, 2047)	2030	2027
14a	(2161, 2183)	2167	2164
15a	(2161, 2171)	2165	2163
17a	(2088, 2169)	2140	2137
18a	(2057, 2139)	2127	2124

Table 14
Comparison of total computational time (in seconds) required by HHS/LNS, TS, hGA, and CDDS on DPdata.

Algorithm	HHS/LNS	TS	hGA	CDDS
CI-CPU	6279	2467	6206	2890

the results of CDDS. It is also very encouraging that the HHS/LNS obtains a remarkable 12 new best known solutions to instances 01a, 02a, 03a, 05a, 06a, 08a, 11a, 12a, 14a, 15a, 17a and 18a in DPdata. The ability of the HHS/LNS to establish new best known solutions with reasonable computing effort strongly proves its effectiveness. In Table 13, we record both the previous and our newly obtained best known solutions to instances in DPdata. Among them, the best lower bound for the instances 01a and 03a is equal to 2505 and 2228, respectively [10], so the instances 01a and 03a are solved optimally by our integrated method. An optimal solution attained by our algorithm for the instance 03a is illustrated in the Appendix.

From Table 11, the average computation time of the HHS/LNS on DPdata is much longer than those of HHS on BRdata. It is not surprising because the instances from DPdata are harder and larger and certainly deserve more search effort. Considering very recent and related literature, such as Oddi et al. set CPU time limit of 3200 s on an AMD Phenom II X4 Quad 3.5 GHz to solve a few large FJSP instances [24], while Beck and Feng execute their algorithm on a cluster with 2 GHz Dual Core AMD Opteron 270 nodes, each with 2 GB of RAM for 3600 s to cope with some hard JSP problems [5]. Thus, the efficiency of our proposed HHS/LNS on solving large-scale FJSP instances is quite acceptable and reasonable. As for a little bigger SD values, it may be explained that our HHS/LNS is composed of two algorithm modules and could be

affected by more nondeterministic factors, another possible reason is attributed to the huge values of operation processing time for the instances in DPdata. From the results obtained by the pure LNS, it can be concluded that the LNS is not comparable with the HHS/LNS, especially for instances with larger problem space. For example, the LNS obtains the same BC_{max} with the HHS/LNS on the instance 04a, but it yields quite bad solutions on the instance 12a.

In Table 14, we make the direct comparison of computational effort required by HHS/LNS, TS, hGA, and CDDS. Line CI-CPU gives the sum of the average computer-independent CPU time on instances in DPdata for each algorithm. All these values have been processed using the normalization coefficients of Dongarra [37], given different performances of different CPUs. In addition, our HHS and LNS are coded in Java and COMET language, respectively, while the other three algorithms are all coded in C/C++. It is well known that the efficiency of Java is much lower than that of C/C++, and the COMET language is in general 3–5 times slower than the comparable C/C++ code [38]. So, we further divide the amount of computational time HHS/LNS takes by a factor of 4 for making a fairer comparison. It should be noted that the comparison between CPU time is mean to be indicative, because we do not have access to other information that influences the computation time, such as the operating systems, software engineering decisions, and coding skills of the programmer. From Table 14, it can be found that the computational effort of HHS/LNS is comparable with that of hGA, but is much more than that of TS and CDDS. Nevertheless, the computational time of all four algorithms on the large-scale instances are all within a factor of three.

From Tables 6 and 7, the HHS could not obtain the best results on several instances in BRdata. Thus, it seems interesting to run the HHS/LNS on BRdata to further refine the solutions obtained by the HHS, although the instances in BRdata are generally not so large-scale as those in DPdata. The best makespan values obtained

by the HHS, LNS, and HHS/LNS on BRdata are listed in Table 15, where the maximum CPU time limit of the pure LNS is set as 1000 s. Unfortunately, as can be seen from Table 15, it is not so encouraging as expected. The HHS/LNS improve the solutions to only two instances (MK06 and MK10) based on HHS, and does not achieve any new best known solution. The reason may be that the solutions yielded by the HHS alone have reached or are quite close to the actual optimum ones of BRdata instances. It is worth noting that the true optimal solutions for the BRdata instances are not known except for the instances MK03 and MK09, whose *LB* values can be verified by the algorithms. Moreover, the performance of pure LNS on BRdata is also comparable with that of HHS and HHS/LNS. Recalling the results obtained by the HHS/LNS on DPdata, it appears to be more promising to use HHS/LNS for solving problems which are relatively large and hard, because there may exist larger gaps between the true optimal solutions and the solutions obtained by the state-of-the-art algorithms for these problems.

Table 16 summarizes the computational results over all the concerned benchmark instances in terms of mean relative error (MRE). The first column reports the data set, the second column reports the number of instances for each data set, the third column reports the average number of alternative machines per operation, the next seven columns report the MRE of the best solution obtained by the HHS, pure LNS, HHS/LNS, TS of Mastrolilli and Gambardella [11], hGA of Gao et al. [14], CDDS of Hmida et al. [22] and M^{2}h of Bozejko et al. [12]. The maximum CPU time limit of the pure LNS is set as 2000 s for BCdata and HUdata instances. For each instance, the relative error (RE) is defined as $RE = [(MK - LB)/LB] \times 100\%$, in which *MK* is the best makespan obtained by the referred algorithm and *LB* is the best-known lower bound. Through the experiment, we find that our HHS alone can solve many instances in HUdata optimally. So, for the integrated method HHS/LNS, if an instance can solved optimally by the HHS, then the LNS would not be executed. From Table 16, it can be seen that our HHS/LNS outperforms all the other algorithms on DPdata, BCdata, Hurink Edata, and Hurink Rdata. But the HHS/LNS is dominated by both the hGA and CDDS on BRdata, and by both the TS and hGA on Hurink Vdata. It should be noted that the LNS

alone could achieve good performance on BCdata and Hurink Edata, although it is still worse than the HHS/LNS. Recalling that the work of Pacino and Van Hentenryck [23] claims that their proposed large neighborhood search using random selection of the relaxations is very effective on Hurink Edata, our results are basically consistent with theirs. It seems that the LNS is more adaptive to solve the instances with a lower degree of flexibility (BCdata and Hurink Edata). However, the good performance of the LNS generally could not be kept when dealing with the large-scale instances with a higher degree of flexibility. The extremely poor results of the LNS on Hurink Vdata are just illustrative of this point. Moreover, the results of the LNS also confirm one of our motivations for this study, which is that the intensification ability of the LNS degrades heavily along with the increase of problem space.

7. Conclusion and future work

In this paper, two algorithm modules, denoted as HHS and LNS, have been developed for the FJSP with makespan criterion. The HHS bears features of evolution-based approaches with the memetic paradigm, while the LNS is a typical constraint-based approach. Considering the advantage and deficiency of the two algorithms, an integrated search heuristic HHS/LNS is established on the base of them. The HHS/LNS is in fact a two-stage algorithm, which starts by executing the HHS, and then the LNS is adopted to further improve the solution obtained by the HHS. To intensify the search of LNS, some good machine assignment information is extracted from the elite solutions in the HM before entering the LNS module, which limits the LNS to a more promising problem space. Empirical results demonstrate that the proposed HHS/LNS shows the competitive performance with state-of-the-art algorithms on large-scale FJSP problems, new upper bounds have been found for 12 out of 18 instances considered in DPdata, while the remaining are about 0.2% on average from the best known solutions. In addition, the evaluation of the HHS/LNS on the other benchmarks also proves its effectiveness.

Indeed, we prefer to regard our integrated method as a kind of framework rather than a single algorithm. The solution procedures used in this paper can be generally divided into three steps: First, an evolutionary algorithm is run on the problem until the solution is hard to improve; second, extract some information from the elite solutions obtained by the evolutionary algorithm, and reduce the problem space according to the extracted information; third, a search method with a strong intensification ability is executed on the reduced problem to further improve the solution obtained by the evolutionary algorithm. In our opinion, the HHS/LNS is only an instance of the framework, and the other suitable alternatives can replace HHS or LNS to form new algorithms. The HHS is used in this paper mainly because of its simple structure and high efficiency.

In future research, we will focus on improving the efficiency and stability of our proposed HHS/LNS. How to better balance the search of HHS and LNS should also be considered, which concerns

Table 15
The comparison of best makespan obtained by the HHS, LNS, and HHS/LNS on BRdata.

Instance	HHS	LNS	HHS/LNS
MK01	40	40	40
MK02	26	26	26
MK03	204	204	204
MK04	60	60	60
MK05	172	173	172
MK06	59	60	58
MK07	139	140	139
MK08	523	523	523
MK09	307	307	307
MK10	202	206	198

Table 16
Mean relative error (MRE) over best-known lower bound.

Data set	Num.	Alt.	HHS (%)	LNS (%)	HHS/LNS(%)	TS (%)	hGA (%)	CDDS (%)	M^{2}h (%)
BRdata	10	2.59	15.52	16.20	14.98	15.14	14.92	14.98	N/A
DPdata	18	2.49	2.90	63.03	1.89	2.01	2.12	1.94	N/A
BCdata	21	1.18	22.86	22.73	22.43	22.53	22.61	22.54	22.53
Hurink Edata	43	1.15	2.34	2.32	2.11	2.17	2.13	2.32	N/A
Hurink Rdata	43	2	1.41	1.44	1.18	1.24	1.19	1.34	N/A
Hurink Vdata	43	4.31	0.19	30.27	0.11	0.095	0.082	0.12	N/A

Note: N/A means that the corresponding data is not available.

the maximization of the strength of both algorithms. In addition, it would be interesting to investigate the performance of the integrated method by replacing HHS or LNS with the other better alternatives. Lastly, a possible research direction is to introduce the LNS into evolutionary algorithms instead of traditional local search to form the memetic framework. Compared with the traditional local search, the LNS can involve more variables and is more efficient in handling constraints, so it is expected to achieve good performance when embedded into evolutionary algorithms.

Acknowledgment

Appendix A

For a detailed scheduling of all the new best known solutions listed in Table 13, refer to http://learn.tsinghua.edu.cn:8080/2010210742/NBKS.rar.

Here, an optimal solution obtained by the HHS/LNS for the instance 03a in DPdata with the makespan of 2228 is illustrated in the following. The starting and completion time of the operations assigned to each machine are given as follows:

M_1: $(O_{8,1}$: 0–55$)(O_{8,2}$: 55–108$)(O_{3,2}$: 108–188$)(O_{10,3}$: 188–246$)$ $(O_{10,4}$: 246–303$)$ $(O_{10,5}$: 303–333$)(O_{10,6}$: 333–393$)(O_{8,4}$: 393–448$)$ $(O_{1,4}$: 448–464$)(O_{7,4}$: 464–497$)(O_{5,3}$: 497–588$)(O_{2,4}$: 588–633$)(O_{9,7}$: 633–675$)(O_{3,7}$: 675–699$)(O_{3,8}$: 699–788$)(O_{8,7}$: 788–861$)(O_{8,8}$: 861–891$)(O_{2,7}$: 891–922$)(O_{7,8}$: 922–971$)(O_{1,8}$: 971–1000$)(O_{7,9}$: 1000–1034$)(O_{8,10}$: 1034–1084$)(O_{10,11}$: 1084–1126$)(O_{6,9}$: 1126–1226$)$ $(O_{2,10}$: 1226–1246$)(O_{5,11}$: 1246–1340$)(O_{7,12}$: 1340–1399$)(O_{9,13}$: 1399–1442$)(O_{2,13}$: 1442–1520$)(O_{5,13}$: 1520–1619$)(O_{3,18}$: 1619–1680$)$ $(O_{5,15}$: 1680–1708$)(O_{6,12}$: 1708–1724$)(O_{6,13}$: 1724–1810$)(O_{7,15}$: 1810–1889$)$ $(O_{7,16}$: 1889–1899$)$ $(O_{2,18}$: 1899–1937$)$ $(O_{5,18}$: 1937–2006$)$ $(O_{1,14}$: 2006–2068$)(O_{8,23}$: 2068–2136$)(O_{10,19}$: 2136–2186$)(O_{5,21}$: 2186–2228$)$.

M_2: $(O_{6,1}$: 0–100$)(O_{2,1}$: 100–141$)(O_{10,2}$: 141–182$)(O_{6,2}$: 182–250$)$ $(O_{9,4}$: 250–286$)(O_{2,3}$: 286–349$)(O_{3,4}$: 349–388$)(O_{3,5}$: 388–486$)(O_{9,6}$: 486–568$)(O_{6,6}$: 568–594$)(O_{5,4}$: 594–637$)(O_{5,5}$: 637–669$)(O_{4,7}$: 669–738$)(O_{5,6}$: 738–792$)(O_{2,6}$: 792–865$)(O_{1,6}$: 865–889$)(O_{1,7}$: 889–952$)$ $(O_{6,7}$: 952–981$)(O_{9,9}$: 981–1032$)(O_{1,9}$: 1032–1115$)(O_{2,9}$: 1115–1126$)$ $(O_{1,10}$: 1126–1149$)(O_{3,13}$: 1149–1184$)(O_{10,12}$: 1184–1217$)(O_{4,12}$: 1217–1278$)(O_{6,10}$: 1278–1365$)$ $(O_{3,15}$: 1365–1398$)(O_{3,16}$: 1398–1457$)$ $(O_{8,15}$: 1457–1523$)(O_{2,14}$: 1523–1560$)(O_{2,15}$: 1560–1628$)(O_{9,16}$: 1628–1646$)(O_{9,17}$: 1646–1709$)(O_{9,18}$: 1709–1808$)(O_{5,17}$: 1808–1905$)$ $(O_{6,14}$: 1905–1941$)(O_{10,18}$: 1941–1964$)(O_{8,21}$: 1964–2023$)(O_{2,19}$: 2023–2072$)(O_{9,22}$: 2072–2159$)(O_{2,20}$: 2159–2228$)$.

M_3: $(O_{10,1}$: 0–81$)(O_{9,2}$: 81–139$)(O_{9,3}$: 139–204$)(O_{3,3}$: 204–287$)$ $(O_{1,1}$: 287–329$)(O_{8,3}$: 329–348$)(O_{1,2}$: 348–410$)(O_{1,3}$: 410–432$)(O_{10,7}$: 432–484$)(O_{8,5}$: 484–572$)(O_{1,5}$: 572–668$)$ $(O_{7,6}$: 668–742$)(O_{9,8}$: 742–824$)(O_{5,7}$: 824–916$)(O_{8,9}$: 916–964$)(O_{3,10}$: 964–1008$)(O_{6,8}$: 1008–1028$)(O_{3,11}$: 1028–1089$)$ $(O_{3,12}$: 1089–1142$)(O_{9,11}$: 1142–1177$)(O_{9,12}$: 1177–1255$)(O_{8,12}$: 1255–1330$)$ $(O_{2,11}$: 1330–1420$)(O_{2,12}$: 1420–1441$)(O_{1,11}$: 1441–1507$)(O_{9,14}$: 1507–1584$)(O_{9,15}$: 1584–1626$)$ $(O_{7,14}$: 1622–1699$)(O_{4,15}$: 1699–1730$)(O_{10,16}$: 1730–1794$)(O_{3,19}$: 1794–1850$)(O_{8,20}$: 1850–1944$)(O_{6,15}$: 1944–2017$)(O_{5,19}$: 2017–2078$)$ $(O_{4,17}$: 2078–2118$)(O_{4,18}$: 2118–2181$)(O_{9,23}$: 2181–2227$)$.

M_4: $(O_{3,1}$: 0–73$)(O_{7,1}$: 73–114$)(O_{5,1}$: 114–209$)(O_{4,2}$: 209–279$)$ $(O_{6,3}$: 279–362$)(O_{7,3}$: 362–385$)(O_{9,5}$: 385–430$)(O_{6,4}$: 430–499$)$ $(O_{7,5}$: 499–591$)(O_{4,6}$: 591–651$)(O_{2,5}$: 651–742$)(O_{4,8}$: 742–836$)(O_{4,9}$: 836–864$)(O_{3,9}$: 864–937$)(O_{5,8}$: 937–977$)(O_{10,10}$: 977–1057$)(O_{9,10}$: 1057–1132$)(O_{7,10}$: 1132–1179$)(O_{4,11}$: 1179–1196$)(O_{5,10}$: 1196–1234$)$

$(O_{10,13}$: 1234–1274$)(O_{7,11}$: 1274–1336$)(O_{8,13}$: 1336–1372$)(O_{8,14}$: 1372–1443$)(O_{4,13}$: 1443–1477$)(O_{3,17}$: 1477–1545$)(O_{4,14}$: 1545–1641$)$ $(O_{10,15}$: 1645–1713$)$ $(O_{5,16}$: 1713–1772$)(O_{8,19}$: 1772–1814$)(O_{9,19}$: 1814–1830$)(O_{10,17}$: 1830–1907$)(O_{4,16}$: 1907–1989$)(O_{3,22}$: 1989–2024$)(O_{8,22}$: 2024–2060$)(O_{3,23}$: 2060–2133$)(O_{1,15}$: 2133–2227$)$.

M_5: $(O_{9,1}$: 0–65$)(O_{4,1}$: 65–143$)(O_{2,2}$: 143–233$)(O_{7,2}$: 233–333$)$ $(O_{5,2}$: 333–401$)(O_{4,3}$: 401–466$)(O_{4,4}$: 466–510$)(O_{6,5}$: 510–527$)$ $(O_{4,5}$: 527–583$)(O_{3,6}$: 583–654$)(O_{8,6}$: 654–700$)(O_{10,8}$: 700–792$)$ $(O_{7,7}$: 792–882$)(O_{10,9}$: 882–914$)(O_{4,10}$: 914–977$)(O_{2,8}$: 977–1074$)$ $(O_{5,9}$: 1074–1158$)(O_{8,11}$: 1158–1195$)(O_{3,14}$: 1195–1290$)(O_{10,14}$: 1290–1370$)(O_{6,11}$: 1370–1437$)(O_{5,12}$: 1437–1500$)(O_{7,13}$: 1500–1598$)$ $(O_{1,12}$: 1598–1627$)(O_{5,14}$: 1627–1672$)(O_{1,13}$: 1672–1683$)(O_{8,16}$: 1683–1722$)(O_{8,17}$: 1722–1739$)(O_{8,18}$: 1739–1769$)(O_{2,16}$: 1769–1817$)$ $(O_{2,17}$: 1817–1872$)(O_{3,20}$: 1872–1900$)(O_{9,20}$: 1900–1934$)(O_{3,21}$: 1934–1952$)(O_{9,21}$: 1952–2051$)(O_{7,17}$: 2051–2083$)(O_{5,20}$: 2083–2181$)(O_{8,24}$: 2181–2202$)(O_{8,25}$: 2202–2227$)$.

References

[1] Colorni A, Dorigo M, Maniezzo V, Trubian M. Ant system for job-shop scheduling. Belgian Journal of Operations Research, Statistics and Computer Science 1994;34(1):39–53.

[2] Cheng C, Smith S. Applying constraint satisfaction techniques to job shop scheduling. Annals of Operations Research 1997;70:327–57.

[3] Nowicki E, Smutnicki C. An advanced tabu search algorithm for the job shop problem. Journal of Scheduling 2005;8(2):145–59.

[4] Huang K, Liao C. Ant colony optimization combined with taboo search for the job shop scheduling problem. Computers & Operations Research 2008;35(4):1030–46.

[5] Beck J, Feng T, Watson J. Combining constraint programming and local search for job-shop scheduling. INFORMS Journal on Computing 2011;23(1):1–14.

[6] Qing-dao-er-ji R, Wang Y. A new hybrid genetic algorithm for job shop scheduling problem. Computers & Operations Research 2012;39(10):2291–9.

[7] Garey MR, Johnson DS, Sethi R. The complexity of flowshop and jobshop scheduling. Mathematics of Operations Research 1976;1:117–29.

[8] Brandimarte P. Routing and scheduling in a flexible job shop by tabu search. Annals of Operations Research 1993;41(3):157–83.

[9] Hurink J, Jurisch B, Thole M. Tabu search for the job-shop scheduling problem with multi-purpose machines. OR Spectrum 1994;15(4):205–15.

[10] Dauzère-Pérès S, Paulli J. An integrated approach for modeling and solving the general multiprocessor job-shop scheduling problem using tabu search. Annals of Operations Research 1997;70(0):281–306.

[11] Mastrolilli M, Gambardella L. Effective neighbourhood functions for the flexible job shop problem. Journal of Scheduling 2000;3(1):3–20.

[12] Bozejko W, Uchronski M, Wodecki M. Parallel Meta²heuristics for the Flexible Job Shop Problem. In: Proceedings of ICAISC 2010, Part II, Lecture notes in computer science, vol. 6114, p. 395–402.

[13] Pezzella F, Morganti G, Ciaschetti G. A genetic algorithm for the flexible job-shop scheduling problem. Computers & Operations Research 2008;35(10):3202–12.

[14] Gao J, Sun L, Gen M. A hybrid genetic and variable neighborhood descent algorithm for flexible job shop scheduling problems. Computers & Operations Research 2008;35(9):2892–907.

[15] Bagheri A, Zandieh M, Mahdavi I, Yazdani M. An artificial immune algorithm for the flexible job-shop scheduling problem. Future Generation Computer Systems 2010;26(4):533–41.

[16] Xing L, Chen Y, Wang P, Zhao Q, Xiong J. A knowledge-based ant colony optimization for flexible job shop scheduling problems. Applied Soft Computing 2010;10(3):888–96.

[17] Wang L, Zhou G, Xu Y, Wang S, Liu M. An effective artificial bee colony algorithm for the flexible job-shop scheduling problem. International Journal of Advanced Manufacturing Technology 2012; 60(1-4):303–15.

[18] Moscato P. On evolution, search, optimization, genetic algorithms and martial arts: towards memetic algorithms. C3P Report 826, California Institute of Technology; 1989.

[19] Harvey WD. Nonsystematic backtracking search. Ph.D. thesis, CIRL, University of Oregon; 1995.

[20] Shaw P. Using constraint programming and local search methods to solve vehicle routing problems. In: CP-98, Fourth international conference on principles and practice of constraint programming, Lecture notes in computer science, vol. 1520; 1998. p. 417–31.

[21] Cesta A, Oddi A, Smith S. Iterative flattening: a scalable method for solving multi-capacity scheduling problems. In: Proceedings of the national conference on artificial intelligence, 2000. p. 742–7.

[22] Ben Hmida A, Haouari M, Huguet M, Lopez P. Discrepancy search for the flexible job shop scheduling problem. Computers & Operations Research 2010;37(12):2192–201.

[23] Pacino D, Van Hentenryck P. Large neighborhood search and adaptive randomized decompositions for flexible jobshop scheduling. In: Proceedings

of the 22th international joint conference on artificial intelligence, Barcelona, 2011.

[24] Oddi A, Rasconi R, Cesta A, Smith S. Iterative flattening search for the flexible job shop scheduling problem. In: Proceedings of the 22th international joint conference on artificial intelligence, Barcelona, 2011.

[25] Geem Z, Kim J, Loganathan G. A new heuristic optimization algorithm: harmony search. Simulation 2001;76(2):60–8.

[26] Pan Q, Wang L, Gao L. A chaotic harmony search algorithm for the flow shop scheduling problem with limited buffers. Applied Soft Computing 2011;11 (8):5270–80.

[27] Lee K, Geem Z. A new meta-heuristic algorithm for continuous engineering optimization: harmony search theory and practice. Computer Methods in Applied Mechanics and Engineering 2005;194:3902–33.

[28] Tamaki H. A paralleled genetic algorithm based on a neighborhood model and its application to the jobshop scheduling. Parallel Problem Solving from Nature 1992;2:573–82.

[29] Cheng R, Gen M, Tsujimura Y. A tutorial survey of job-shop scheduling problems using genetic algorithms—I. Representation. Computers & Industrial Engineering 1996;30(4):983–97.

[30] Wang L, Pan Q, Fatih Tasgetiren M. Minimizing the total flow time in a flow shop with blocking by using hybrid harmony search algorithms. Expert Systems with Applications 2010;37(12):7929–36.

[31] Carchrae T, Beck J. Principles for the design of large neighborhood search. Journal of Mathematical Modelling and Algorithms 2009;8(3):245–70.

[32] Michel L, Van Hentenryck P. A constraint-based architecture for local search. In: Conference on object-oriented programming systems, languages, and applications, Seattle, November 2002. p. 101–10.

[33] Comet Tutorial. URL: ⟨http://www.lsi.upc.edu/~larrosa/comet.pdf⟩; 2010.

[34] Godard D, Laborie P, Nuijten W. Randomized large neighborhood search for cumulative scheduling. In: Proceedings of the 15th international conference on automated planning & scheduling, 2005. p. 81–9.

[35] Barnes JW, Chambers JB. Flexible Job Shop Scheduling by tabu search. Graduate program in operations research and industrial engineering. The University of Texas at Austin, Technical Report Series, ORP96-09; 1996.

[36] Mastrolilli M, Gambardella L. Effective neighbourhood functions for the flexible job shop problem: Appendix, Technical Report, IDSIA-Istituto Dalle Intelligenza Artificiale. Electronic version available at: ⟨http://www.idsia.ch/~monaldo/fjsp.html⟩; 2000.

[37] Dongarra J. Performance of various computers using standard linear equations software. Computer Science Department, University of Tennessee, Knoxville, Tennessee; 2009.

[38] Jain S, Van Hentenryck P. Large neighborhood search for dial-a-ride problems. In: CP-2011, 17th international conference on principles and practice of constraint programming, Lecture notes in computer science, vol. 6876; 2011. p. 400–13.