J Intell Manuf (2018) 29:19–34
https://doi.org/10.1007/s10845-015-1083-z

CrossMark

A genetic algorithm embedded with a concise chromosome representation for distributed and flexible job-shop scheduling problems

Po-Hsiang Lu[1] · Muh-Cherng Wu[1] · Hao Tan[1] · Yong-Han Peng[1] · Chen-Fu Chen[1]

Received: 19 November 2014 / Accepted: 17 April 2015 / Published online: 5 May 2015
© Springer Science+Business Media New York 2015

Abstract This paper proposes a genetic algorithm GA_JS for solving distributed and flexible job-shop scheduling (DFJS) problems. A DFJS problem involves three scheduling decisions: (1) job-to-cell assignment, (2) operation-sequencing, and (3) operation-to-machine assignment. Therefore, solving a DFJS problem is essentially a *3-dimensional* solution space search problem; each *dimension* represents a type of decision. The GA_JS algorithm is developed by proposing a *new* and *concise* chromosome representation $\boldsymbol{S_{JOB}}$, which models a 3-dimensional scheduling solution by a *1-dimensional scheme* (i.e., a sequence of all jobs to be scheduled). That is, the chromosome space is 1-dimensional (1D) and the solution space is 3-dimensional (3D). In GA_JS, we develop a 1D-to-3D *decoding* method to convert a 1D chromosome into a 3D solution. In addition, given a 3D solution, we use a *refinement* method to improve the scheduling performance and subsequently use a 3D-to-1D *encoding* method to convert the refined 3D solution into a 1D chromosome. The 1D-to-3D decoding method is designed to obtain a "good" 3D solution which tends to be *load-balanced*. In contrast, the refinement and 3D-to-1D encoding methods of a 3D solution provides a novel way (rather than by genetic operators) to generate new chromosomes, which are herein called *shadow chromosomes*. Numerical experiments indicate that GA_JS outperforms the *IGA* developed by De Giovanni and Pezzella (Eur J Oper Res 200:395–408, 2010), which is the up-to-date best-performing genetic algorithm in solving DFJS problems.

✉ Muh-Cherng Wu
mcwu@mail.nctu.edu.tw

[1] Department of Industrial Engineering and Management, National Chiao Tung University, Hsin-Chu City 30010, Taiwan

Keywords Genetic algorithm · Distributed flexible job-shop · Chromosome representation · Chromosome space · Solution space · Shadow chromosomes

Introduction

This research examines the *distributed* and *flexible job-shop* scheduling (DFJS) problems. The DFJS problem addresses a manufacturing system comprising several sub-systems (also called manufacturing cells); each cell is a flexible job-shop. Each job shall be processed in one cell (i.e., cross-cell production is prohibited). DFJS examples can be a multi-factory network in which factories are geographically distributed, and can be a multi-cell plant where several manufacturing cells are located in the same plant. To reduce overall completion time, the assignment of jobs to cells is very important because it shall affect *cell loading* profiles.

As stated, each cell in a DFJS system is a *flexible job-shop*, which denotes that an operation of a job can be processed by more than one machine. Therefore, assigning an operation to a different machine yields a different process route. The operation-to-machine assignment decision is very important in scheduling because it shall affect *machine loading* profiles.

In a flexible job-shop, after the operation-to-machine assignment decision has been made, each machine may have several operations to be processed. Operations assigned to the same machine must be sequenced in using the machine capacity. Therefore, operation-sequencing decision has an effect on job completion time and is very important in scheduling.

In summary, a DFJS problem involves three scheduling decisions: (1) job-to-cell assignment, (2) operation-sequencing, and (3) operation-to-machine assignment. Two degenerated DFJS problems are studied in literature. One is

called flexible job-shop scheduling (FJS) problem, in which only one manufacturing cell exists in the DFJS system. The other is called distributed job-shop scheduling (DJS) problem, in which each job has a *fixed route*. Therefore, the job-to-cell assignment decision is not concerned in the FJS problem while the operation-to-machine assignment decision is not concerned in the DJS problem.

DJS problems are more complicated than classical job-shop scheduling (JS) problems which are NP-hard (Garey et al. 1976). Many prior studies developed genetic algorithms to solve DJS problems. For example, Jia et al. (2003) developed a genetic algorithm, which is further enhanced by Jia et al. (2007). Chan et al. (2005) developed an adaptive genetic algorithm with dominated genes. Some other prior studies attempted to solve dynamic DJS problems which include unexpected events. For example, Zhang et al. (2008) developed a multi-agent genetic algorithm and Chou and Cheng (2010) developed an agent-based method.

FJS problems are strongly NP-hard (Garey et al. 1976). Some studies developed mathematical programming approach (Bruker and Schlie 1990; Jinyan et al. 1995; Kim and Egbelu 1999; Choi and Choi 2002). Some other studies (e.g., Hmida et al. 2010) developed tree search method. And many other studies developed meta-heuristic algorithms in two approaches. One approach is making the two scheduling decisions in a hierarchical manner. That is, the operation-to-machine assignment decision is firstly made; then the operation-sequencing decision is subsequently made by meta-heuristic algorithms (Brandimarte 1993; Tung et al. 1999; Kacem et al. 2002a, b; Bożejko et al. 2010). The other approach is making the two decisions simultaneously by meta-heuristic algorithms, which include tabu search algorithms (Hurink et al. 1994; Dauzère-Pérès and Paulli 1997; Mastrolilli and Gambardella 2000; Jia and Hu 2014), genetic algorithm (Ho and Tay 2004; Pezzella et al. 2008; Tay and Wibowo 2004; Zhang et al. 2011), simulated annealing (Baykasoğlu 2002), and *hybrid* meta-heuristic algorithms (Xia and Wu 2005; Gao et al. 2008; González et al. 2013; Xing et al. 2010; Yuan and Xu 2013; Gutiérrez and García-Magariño 2011).

In solving DFJS problems, two studies (Chan et al. 2006; De Giovanni and Pezzella 2010) developed genetic algorithms (GAs); and Ziaee (2014) developed a heuristic algorithm. A mathematical formulation of DFJS problems can be referred to Chan et al. (2006). The GA developed by De Giovanni and Pezzella (2010) is called *IGA*, which is the up-to-date best-performing algorithm for solving DFJS problems and is taken as the benchmark of this research. To facilitate comparison, the chromosome representations used in Chan et al. (2006) is called $\boldsymbol{S_C}$ and that used in De Giovanni and Pezzella (2010) is called $\boldsymbol{S_G}$ hereafter.

This paper proposes a genetic algorithm GA_JS for solving DFJS problems. The GA_JS algorithm is developed by proposing a *new* and *concise* chromosome representation $\boldsymbol{S_{JOB}}$, which models a *3-dimensional* DFJS scheduling solution by a *1-dimensional scheme* (i.e., a sequence of all jobs to be scheduled). That is, the chromosome space is 1-dimensional (1D) and the solution space is 3-dimensional (3D).

In GA_JS, we develop a 1D-to-3D *decoding* method to convert a 1D chromosome into a 3D solution. In addition, given a 3D solution, we use a refinement method to improve the scheduling performance and subsequently use a 3D-to-1D *encoding* method to convert the refined 3D solution into a 1D chromosome. The 1D-to-3D decoding method is designed to obtain a "good" 3D solution which tends to be *load-balanced*. In contrast, the refinement and 3D-to-1D encoding methods of a 3D solution provides a novel way (rather than by genetic operators) to generate new chromosomes, which are herein called *shadow chromosomes*. Numerical experiments indicate that GA_JS outperforms the *IGA* developed by De Giovanni and Pezzella (2010).

The remaining of this paper is organized as follows. "Comparison of chromosome representations" section compares the proposed chromosome representation $\boldsymbol{S_{JOB}}$ against the two prior ones $\boldsymbol{S_C}$ and $\boldsymbol{S_G}$. "GA_JS algorithmic framework" section describes the algorithmic framework of GA_JS. "Decoding of $\boldsymbol{S_{JOB}}$ chromosomes" section presents the 1D-to-3D decoding method for converting a 1D $\boldsymbol{S_{JOB}}$ chromosome into a 3D scheduling solution. "Refinement and encoding of 3D solutions" section describes the refinement and 3D-to-1D encoding methods for obtaining a shadow chromosome from a 3D solution. "GA_JS algorithm" section summarizes the proposed algorithm GA_JS. "Numerical experiments" section reports experiments of comparing GA_JS against *IGA*. Concluding remarks are in last section.

Comparison of chromosome representations

This section compares the proposed chromosome representation $\boldsymbol{S_{JOB}}$ against two prior ones $\boldsymbol{S_C}$ and $\boldsymbol{S_G}$, in which $\boldsymbol{S_C}$ is proposed by Chan et al. (2006) and $\boldsymbol{S_G}$ is proposed by De Giovanni and Pezzella (2010). The three chromosome representations are explained by referring to a DFJS problem shown in Table 1(a), in which there are 3 jobs and 2 cells and each job has 3 operations. Then, the three chromosome representations are examined in terms of eight encoding principles (or properties) proposed in prior research (Gen et al. 2008; Gen and Cheng 1997; Raidl and Julstrom 2003; Lin and Gen 2006).

$\boldsymbol{S_C}$ chromosome representation

As shown in Fig. 1a, an $\boldsymbol{S_C}$ chromosome (represented by a sequence of genes) denotes a sequence of all operations. In

the example DFJS problem, there are 9 operations in total; thus an $\boldsymbol{S_C}$ chromosome involves 9 genes. Each gene models an operation by a 5-tuple vector. See the figure, the first gene is (2, 2, 3, 1, 1) in which the 3rd and 4th elements are used to identify the operation O_{31} (the 1st operation of Job J_3). Moreover in the vector (2, 2, 3, 1, 1), the 1st element denotes the job-to-cell assignment decision, and the 2nd denotes the operation-to-machine assignment decision. That is, job J_3 is assigned to cell U^2, and operation O_{31} is assigned to machine M^{22}. Finally, the 5th element in the vector is a binary variable, in which 1 denotes that the gene is a *dominated* gene and 0 denotes that the gene is a *non-dominated* one. While applying genetic operators to generate new chromosomes, only dominated genes can be changed and non-dominated genes shall keep unchanged.

In summary, $\boldsymbol{S_C}$ chromosome representation explicitly models the three DFJS scheduling decisions: (1) job-to-cell assignment, (2) operation-sequencing, and (3) operation-to-machine assignment. As a result, the use of $\boldsymbol{S_C}$ chromosomes results in a 3D *chromosome space* while developing a GA.

Table 1 A DFJS example (a) Job-Cell-Op table, (b) Job-Cell table

	U^1			U^2		
	d_i^1	M^{11}	M^{12}	d_i^2	M^{21}	M^{22}
(a)						
J_1						
O_{11}	1	6	2	2	–	2
O_{12}		1	4		4	5
O_{13}		2	–		3	4
J_2						
O_{21}	1	2	3	2	6	7
O_{22}		1	5		2	–
O_{23}		5	3		4	5
J_3						
O_{31}	1	–	1	2	–	2
O_{32}		4	5		–	2
O_{33}		3	–		1	3

	U^1	U^2
(b)		
J_1	8.5	10
J_2	9.5	13
J_3	8.5	6

$\boldsymbol{S_G}$ chromosome representation

As shown in Fig. 1b, an $\boldsymbol{S_G}$ chromosome also denotes a sequence of all operations (i.e., each gene models an operation). Therefore, an $\boldsymbol{S_G}$ chromosome also involves 9 genes in the example DFJS problem. Each gene models an operation by a 2-tuple vector. See the figure, the first gene is (2,3) in which the 1st element denotes a cell (U^2), and the 2nd element denotes a job (J_3). This implies that job J_3 is assigned to cell U^2. Moreover, the first appearance of job J_3 implies that this gene denotes operation O_{31} (i.e., the 1st operation of job J_3). Operation O_{31} and the others can be identified accordingly.

Out of the three DFJS scheduling decisions, $\boldsymbol{S_G}$ chromosome representation models only two ones: (1) job-to-cell

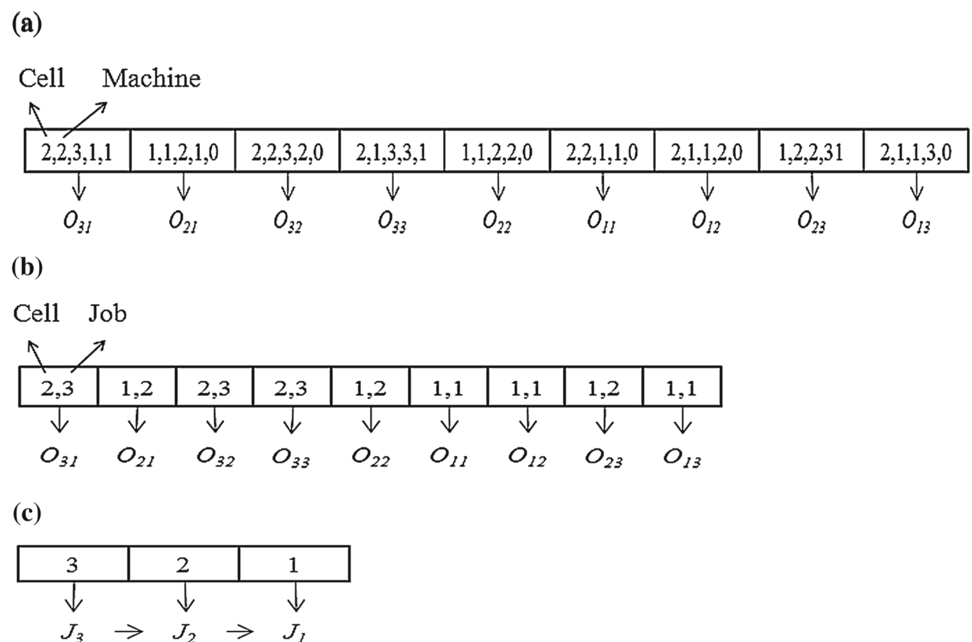

Fig. 1 **a** $\boldsymbol{S_C}$ chromosome representation, **b** $\boldsymbol{S_G}$ chromosome representation, and **c** $\boldsymbol{S_{JOB}}$ chromosome representation

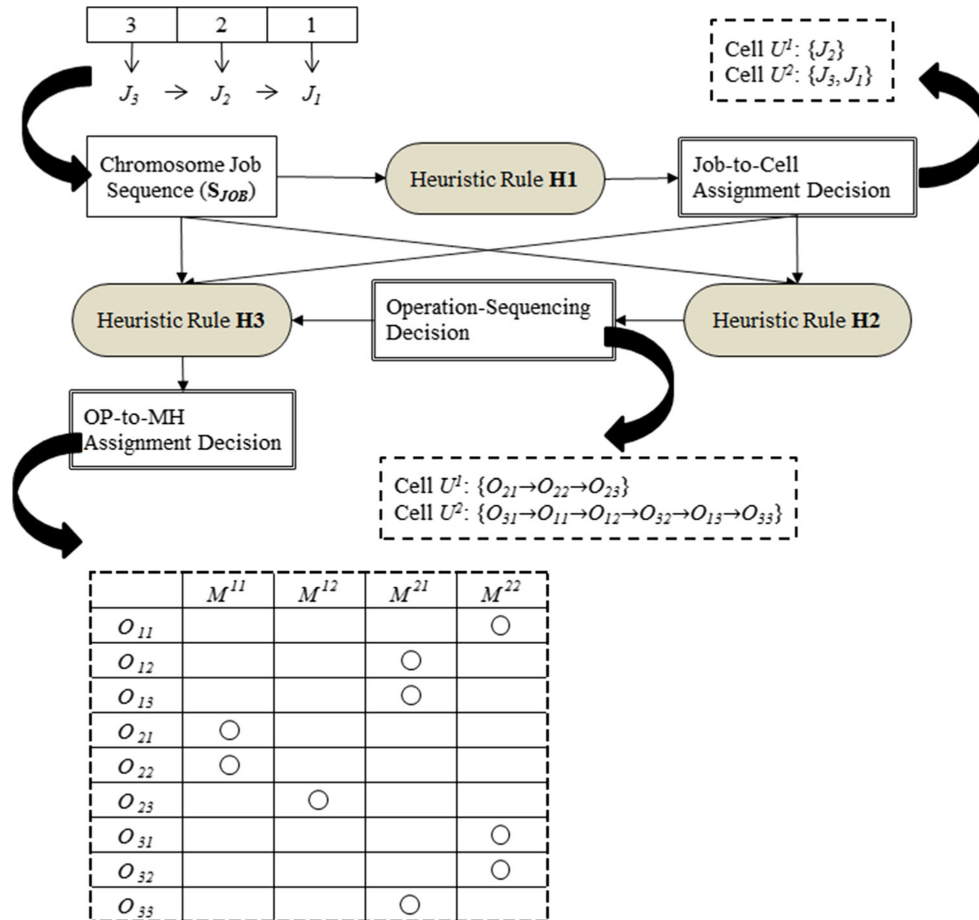

Fig. 2 The decoding method for converting a S_{JOB} chromosome into a 3D scheduling solution

assignment and (2) operation-sequencing. And the operation-to-machine assignment decision is obtained by a heuristic rule (De Giovanni and Pezzella 2010). As a result, the use of S_G chromosomes results in a 2D *chromosome space* while developing a GA.

S_{JOB} chromosome representation

As shown in Fig. 1c, an S_{JOB} chromosome denotes a sequence of all jobs (i.e., each gene models a job). Now an S_{JOB} chromosome involves only 3 genes in the example DFJS problem. See the figure, the chromosome (3,2,1) denotes that a job sequence $J_3 \rightarrow J_2 \rightarrow J_1$. By developing a decoding method which involves three heuristic rules (Fig. 2), we can convert an S_{JOB} chromosome into a DFJS scheduling solution. In the proposed GA_JS algorithm, the use of S_{JOB} chromosome representation results in a 1D *chromosome space*.

In summary, solving a DFJS problem is a 3D *solution space* search problem. In developing a GA, the use of S_C results in a 3D *chromosome space*; the use of S_G results in a 2D *chromosome space*; and the use of S_{JOB} results in a 1D *chromosome space*.

Chromosome properties examinations

According to prior research (Gen et al. 2008; Gen and Cheng 1997; Raidl and Julstrom 2003; Lin and Gen 2006), eight principles (or properties) have been proposed to evaluate an encoding scheme. In the following, the three chromosome representations (S_C, S_G, and S_{JOB}) are examined in terms of the eight properties, in which m denotes the total number of all job operations and n denotes the total number of jobs; therefore $m > n$ substantially because a job has many operations.

Property 1 (*Space*): *Chromosomes should not require extravagant amounts of memory.* The S_C chromosome represents a solution by $5m$ integers (m genes and each gene involves 5 integers). The S_G chromosome represents a solution by $2m$ integers (m genes and each gene involves 2 integers). The S_{JOB} chromosome represents a solution by n integers (n genes and each gene involves 1 integer). In terms

of memory requirement, the proposed $\boldsymbol{S_{JOB}}$ chromosome is the most concise.

Property 2 (*Time*): *The time complexity of executing evaluation, recombination and mutation on chromosomes should be small.* In terms of chromosome evaluation, S_C requires less computation time than $\boldsymbol{S_G}$; and $\boldsymbol{S_G}$ requires less time than $\boldsymbol{S_{JOB}}$. This is due to that S_C requires no decoding, while $\boldsymbol{S_{JOB}}$ requires three decoding rules and $\boldsymbol{S_G}$ requires only one decoding rule. Yet, in terms of recombination and mutation on chromosomes, $\boldsymbol{S_{JOB}}$ requires the least in computation time because its gene length is the shortest.

Property 3 (*Feasibility*): *A chromosome corresponds to a feasible solution.* The $\boldsymbol{S_C}$ chromosome may yield an infeasible solution; for example if operation O_{32} precedes O_{31}. The $\boldsymbol{S_G}$ chromosome may also yield an infeasible solution; for example, if one job (J_3) is assigned to two different cells (U^1 and U^2); that is genes (1, 3) and (2, 3) appear simultaneously. In contrast, the proposed $\boldsymbol{S_{JOB}}$ chromosome always yields feasible solutions.

Property 4 (*Legality*): *Any permutation of a chromosome corresponds to a solution.* In $\boldsymbol{S_C}$ chromosomes, a permutation may results in an infeasible solution; while in $\boldsymbol{S_G}$ and $\boldsymbol{S_{JOB}}$ chromosomes, any permutation always results in a feasible solution.

Property 5 (*Completeness*): *Any solution has a corresponding chromosome.* In $\boldsymbol{S_C}$ chromosomes, any solution indeed has a *corresponding chromosome*. In $\boldsymbol{S_G}$ and $\boldsymbol{S_{JOB}}$ chromosomes, not all solution has a corresponding chromosome due to the use of decoding rules. For example, decoding the $\boldsymbol{S_G}$ chromosome in Fig. 1b by a heuristic rule shall yield only one operation-to-machine assignment decision; yet there are many other alternatives.

Property 6 (*Uniqueness*): *The mapping from chromosomes to solution may belong one of the following three cases: 1-to-1 mapping, n-to-1 mapping, and 1-to-n mapping*. We consider the solution space as the set of all the decision portfolios, each portfolio represents an alternative of the three scheduling decisions. Then, $\boldsymbol{S_C}$ is 1-to-1, $\boldsymbol{S_G}$ and $\boldsymbol{S_{JOB}}$ are both *n*-to-1.

Property 7 (*Heritability*): *Offspring of simple crossover (i.e., one-cut point crossover) should correspond to solutions which combine the basic feature of their parents*. The three chromosomes $\boldsymbol{S_C}$, $\boldsymbol{S_G}$, and $\boldsymbol{S_{JOB}}$ have different degrees of heritability. $\boldsymbol{S_G}$ has the highest degree of heritability by completely keeping job-to-cell assignment and partially keeping operation-sequencing, $\boldsymbol{S_{JOB}}$ ranks 2 by partially keeping job sequence, and $\boldsymbol{S_C}$ ranks the last because a simple crossover will very likely yield an infeasible solution.

Property 8 (*Locality*): *A mutated chromosome should usually represent a solution similar to that of its parent*. $\boldsymbol{S_C}$ has the highest degree of locality due to substantially keeping operation-to-machine assignment and operation-sequencing; $\boldsymbol{S_G}$ ranks 2 due to substantially keeping operation sequence, and $\boldsymbol{S_{JOB}}$ ranks the last due to substantially keeping job sequence.

GA_JS algorithmic framework

The proposed GA_JS algorithmic framework is shown in Fig. 3. As stated, the chromosome space is 1D and the solution space is 3D. To justify the "goodness" of a 1D chromosome, we have to develop a 1D-to-3D decoding method for converting a 1D chromosome into a 3D scheduling solution. Details of the decoding method shall be presented in "Decoding of $\boldsymbol{S_{JOB}}$ chromosomes" section.

The GA_JS attempts to find out a best-ever solution by an evolutionary search process. That is, GA_JS firstly generates a finite set of chromosomes (called *chromosome population*) and iteratively updates the chromosome population in order to find a best-ever 3D scheduling solution. The

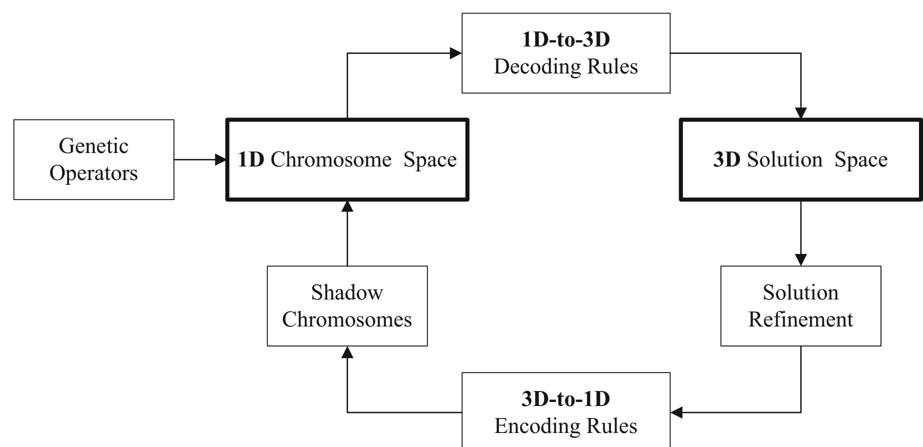

Fig. 3 The algorithmic framework of GA_JS

GA_JS is distinguished in proposing a new way of generating new chromosomes (called *shadow chromosomes*).

See Fig. 3, the chromosome population in GA_JS is updated by two ways: (1) genetic operators and (2) shadow chromosomes. The basic idea of genetic operators is a 1D-to-1D chromosome generation; that is, a new 1D chromosome is generated from one or two existing 1D chromosome. The genetic operators in the GA_JS include *crossover* and *mutation* operators, which are essentially adapted from prior GA studies.

In contrast, the basic idea of shadow chromosomes is a 3D-to-1D chromosome generation. That is, given a 3D scheduling solution, we firstly use a refinement method to improve the scheduling performance; the outcome is called the *refined* 3D solution. Secondly, we use a 3D-to-1D encoding method to generate a new chromosome from the refined 3D solution. Such a newly generated chromosome is called a shadow chromosome. Notice that shadow chromosomes are generated by a novel way rather than genetic operators. Details of the refinement and 3D-to-1D encoding methods shall be presented in "Refinement and encoding of 3D solutions" section; and the GA_JS procedure is summarized in "GA_JS algorithm" section.

Decoding of $\boldsymbol{S_{JOB}}$ chromosomes

In the proposed GA_JS, the use of $\boldsymbol{S_{JOB}}$ chromosome results in a 1D chromosome space; yet a DFJS problem is concerned with a 3D solution space. We develop a *decoding* method to convert an $\boldsymbol{S_{JOB}}$ chromosome into a DFJS scheduling solution. The 1D-to-3D decoding method involves three heuristic rules (**H1**, **H2,** and **H3**) as shown in Fig. 2. Each heuristic rule is respectively explained below by referring to the example DFJS problem in Table 1(a) and the example $\boldsymbol{S_{JOB}}$ chromosome ($J_3 \rightarrow J_2 \rightarrow J_1$) in Fig. 1c. Notation used for explaining GA_JS and the three heuristic rules and are listed below.

Notation

- J_i: job i, $i = 1, \ldots, n$
- O_{ij}: jth operation of job i, $j = 1, \ldots, n_i$
- U^l: cell l, $l = 1, \ldots, q$
- M^{lk}: kth machine in cell l, $k = 1, \ldots, H_l$
- d_i^l: transportation time required to move job i in and out cell l
- p_{ij}^{lk}: processing time of operation O_{ij} on machine M^{lk}
- $\boldsymbol{J}$: set of all jobs
- $\boldsymbol{U}$: set of all cells
- $\boldsymbol{M}$: set of all machines

Decoding rule H1: job-to-cell assignment

See Fig. 2, given a $\boldsymbol{S_{JOB}}$ chromosome, heuristic rule **H1** is developed to determine the job-to-cell assignment decision. The idea of rule **H1** is to *balance the workload of each cell*. The **H1** procedure involves two steps with its pseudo codes listed below followed by an example.

Procedure H1

Step 1: Convert *Job-Cell-Op* table into *Job-Cell* table (from Table 1(a) to Table 1(b))

purpose: estimate average processing time (p_i^l) for each job (J_i) in each cell (U^l)

input: *Job-Cell-Op* table ($p_{ij}^{lk}, 1 \le i \le n, 1 \le l \le q, j = 1, \ldots, n_i, k = 1, \ldots, H_l$)

output: *Job-Cell* table ($p_i^l, 1 \le i \le n, 1 \le l \le q$)

begin

    for $i = 1$ **to** n **do**

        for $l = 1$ **to** q **do**

            compute $p_i^l = \sum_j \overline{p_{ijl}}$, where $\overline{p_{ijl}} = \frac{\sum_k p_{ij}^{lk}}{H_l}$;

        end

    end

    output *Job-Cell* table;

end

Step 2: Assign each job to a cell based on *Job-Cell* table (Table 1(b))

input: *Job-Cell* table, $\boldsymbol{S_{JOB}}$ chromosome—a sequence of jobs ($J_{\sigma(1)} \rightarrow \cdots \rightarrow J_{\sigma(n)}$)

output: job-to-cell assignment (S_1, S_2, ..S_q, where S_l is a set of assigned jobs of cell l)

begin

    $L^l \leftarrow 0$ $(1 \le l \le q)$; /*initial loading L^l of each cell l is 0*/

    for $i = 1$ **to** n **do** /*for each job*/

        for $l = 1$ **to** q **do** /*for each cell*/

            compute $C_{J_{\sigma(i)}}^l = L^l + p_{J_{\sigma(i)}}^l + d_{J_{\sigma(i)}}^l$; /*$C_{J_{\sigma(i)}}^l$ is makespan of $J_{\sigma(i)}$ in cell l*/

        end

        $l^* = Arg(\min_{1 \le l \le q} C_{J_{\sigma(i)}}^l)$; /*cell l^* has minimum makespan for $J_{\sigma(i)}$*/

        place $J_{\sigma(i)}$ in set S_{l^*}; /*assign job $J_{\sigma(i)}$ to cell l^**/

        $L^{l^*} = L^{l^*} + p_{J_{\sigma(i)}}^{l^*}$; /*update the load of cell l^**/

    end

    output job-to-cell assignment (S_1, S_2, ..S_q);

end

We now use an example to explain the **H1** procedure. Step 1 is designed to estimate the average processing time p_i^l for each job (J_i) in each cell (U^l) to obtain a *Job-Cell* table (Table 1(b)). Each element (p_i^l) in Table 1(b) is obtained by the formula: $p_i^l = \sum_j \overline{p_{ijl}}$, where $\overline{p_{ijl}} = \frac{\sum_k p_{ij}^{lk}}{H_l}$ denotes the average processing time of operation O_{ij} in cell U^l. See Table 1(a), $\overline{p_{111}} = (6+2)/2 = 4$ indicates that the average processing time of operation O_{11} in cell U^1 is 4. Accordingly, we can obtain $\overline{p_{121}} = 2.5$ and $\overline{p_{131}} = 2$. As a result, $p_1^1 =$

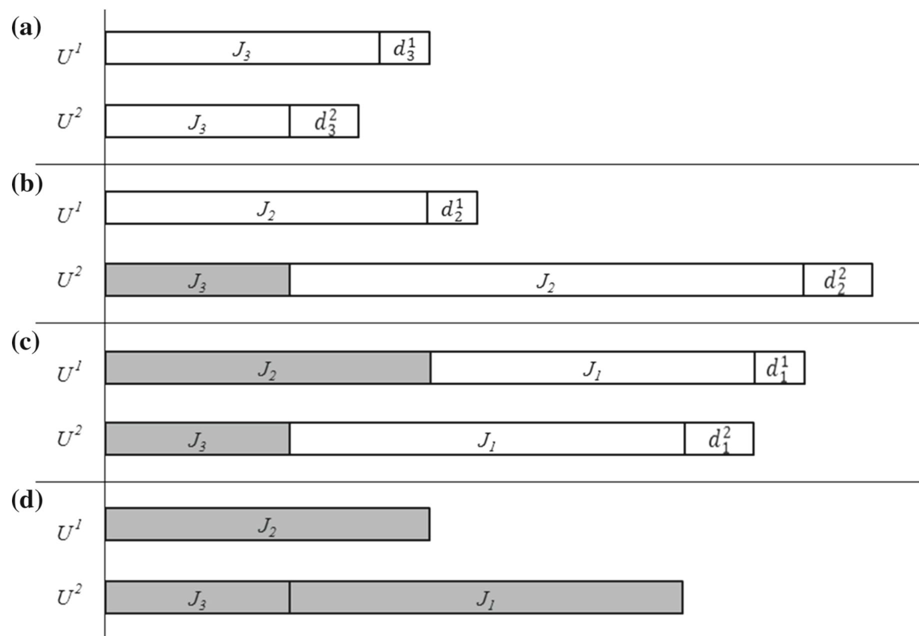

Fig. 4 An example of implementing heuristic rule **H1**

$\sum_j \overline{p_{1j1}} = (\overline{p_{111}} + \overline{p_{121}} + \overline{p_{131}}) = (4 + 2.5 + 2) = 8.5$. Each element $\left(p_i^l\right)$ in Table 1(b) can be accordingly obtained.

Step 2 is designed to assign each job to a particular cell. Firstly, $\boldsymbol{S_{JOB}}$ is used to determine the sequence of assigning jobs; for example, the three jobs in Fig. 4 shall be assigned to cells by following the sequence $J_3 \rightarrow J_2 \rightarrow J_1$. Then, the *Job-Cell* table (Table 1(b)) is used to assign each job to a cell by a *try-and-select* approach. That is, a job is assigned to each possible cell, and we shall select the cell whose "expected makespan" is the minimum one. The "expected makespan" is the sum of the cell workloads and the job transportation time. For example, in Fig. 4a, job J_3 is tried to be assigned to cells U^1 and U^2 which shows that the expected makespan of U^1 is longer than that of U^2; as a result, J_3 is assigned to cell U^2. Accordingly, the job assignment result can be obtained in Fig. 4. Notice that minimizing "expected makespan" in making job-to-cell assignment decision tends to balance the workload of each cell.

See Fig. 2, decoding the example $\boldsymbol{S_{JOB}}$ chromosome ($J_3 \rightarrow J_2 \rightarrow J_1$) by applying heuristic rule **H1** results in a job-to-cell assignment decision, in which $\{J_2\}$ is assigned to cell U^1 and $\{J_3 \rightarrow J_1\}$ is assigned to cell U^2.

Decoding rule H2: operation-sequencing

See Fig. 2, heuristic rule **H2** is used to determine the operation-sequencing decision for each cell. The idea of rule **H2** is to give higher priority to the job with longer remaining processing time with its pseudo codes listed below followed by an example.

Procedure H2

for $l = 1$ **to** q **do**

Denote the jobs assigned to cell l as the ordered set $\{J_{\sigma(1)} \rightarrow \cdots \rightarrow J_{\sigma(h)}\}$

Step 1: Sequencing $O_{\sigma(i),1}$ in cell l.

input: job sequence $\{J_{\sigma(1)} \rightarrow \cdots \rightarrow J_{\sigma(h)}\}$ in cell l

output: $S_1 = \{O_{\sigma(1),1} \rightarrow \cdots \rightarrow O_{\sigma(h),1}\}$; /*follow the given job sequence*/

Step 2: Sequencing remaining operations of each job in cell l.

input: *Job-Cell-Op* table, $S_1 = \{O_{\sigma(1),1} \rightarrow \cdots \rightarrow O_{\sigma(h),1}\}$

output: S_2 = {a sequence of all remaining operations}

begin

$r(i) \leftarrow 2 \;\; (1 \le i \le h)$; /*initialize 1st remaining operation of each job*/

for $i = 1$ **to** h **do** /*for each job in cell l*/

$\overline{p_{\sigma(i),j,l}} = \frac{\sum_k p_{\sigma(i),j}^{lk}}{H_l} \; (r(i) \le j \le n_{\sigma(i)})$; /*APT of each remaining operation*/

$p_{\sigma(i)}^l = \sum_{r(i) \le j \le n_{\sigma(i)}} \overline{p_{\sigma(i),j,l}}$; /*APT of all remaining operations of job $\sigma(i)$*/

$i^* = \text{Arg}(\text{Max}_{1 \le i \le h}\, p_{\sigma(i)}^l)$; /*choose job i^* with longest APT*/

place $O_{\sigma(i^*),r(i^*)}$ in the ordered set S_2; /*pick 1st remaining operation of job i^**/

$r(i^*) \leftarrow r(i^*) + 1$; /*update the 1st remaining operation of job i^**/

end

output S_2 = {a sequence of all remaining operations};

end

output S = {a sequence of all operations} by consolidating S_1 and S_2;

end

We now explain heuristic rule **H2** by referring to Table 2, in which job J_1 and J_3 both have been assigned to cell U^2 by applying rule **H1**. In Step 1, we sequence the first operations (O_{31} and O_{11}) of all jobs in the cell (U^2) by following the job

Table 2 An example of implementing heuristic rule **H2**

(a)

	1st operation	Remaining operations	
J_3	O_{31}	O_{32}	O_{33}
J_1	O_{11}	O_{12}	O_{13}

Operation sequence: $O_{31} \to O_{11}$

(b)

	1st operation	Remaining operations		Remaining load
J_3	~~O_{31}~~	O_{32}	O_{33}	4.0
J_1	~~O_{11}~~	O_{12}	O_{13}	8.0

Operation sequence: $O_{31} \to O_{11} \to O_{12}$

(c)

	1st operation	Remaining operations		Remaining load
J_3	~~O_{31}~~	O_{32}	O_{33}	4.0
J_1	~~O_{11}~~	~~O_{12}~~	O_{13}	3.5

Operation sequence: $O_{31} \to O_{11} \to O_{12} \to O_{32}$

(d)

	1st operation	Remaining operations		Remaining load
J_3	~~O_{31}~~	~~O_{32}~~	O_{33}	2.0
J_1	~~O_{11}~~	~~O_{12}~~	O_{13}	3.5

Operation sequence: $O_{31} \to O_{11} \to O_{12} \to O_{32} \to O_{13}$

(e)

	1st operation	Remaining operations		Remaining load
J_3	~~O_{31}~~	~~O_{32}~~	O_{33}	2.0
J_1	~~O_{11}~~	~~O_{12}~~	~~O_{13}~~	0.0

Operation sequence: $O_{31} \to O_{11} \to O_{12} \to O_{32} \to O_{13} \to O_{33}$

sequence $J_3 \to J_1$ in Fig. 1c. See Table 2(a), the resulting operation sequence is $O_{31} \to O_{11}$.

In Step 2, we attempt to sequence the remaining operations. See Table 2(b), of all remaining operations, now only the *leading* operations (O_{32} and O_{12}) of each job can be selected as the next one of the present operation sequence ($O_{31} \to O_{11}$). This implies that we need to select one job in the cell; once a job is selected, its leading operation is selected. For all jobs in the cell, we select the one whose *remaining load* is the longest. See Table 2(b), the *remaining load* of J_3 is 4.0 and that of J_1 is 8.0. Therefore, J_1 (its leading operation O_{12}) shall be selected; and the operation sequence becomes $O_{31} \to O_{11} \to O_{12}$. Herein, the *remaining load* of a job denotes the sum of the average processing time (APT) of all its remaining operations. For example, for job J_3 in Table 1(a), the APT of O_{32} is $p_{32}^{22} = 2$ and that of O_{33} is $(p_{33}^{21} + p_{33}^{22})/2 = (1+3)/2 = 2$; as a result, the *remaining load* of job J_3 is $2+2=4$. Accordingly, we can obtain that the *remaining load* of job J_1 is 8.0. See Table 2(e), repeatedly following the above step, we can obtain the ultimate operation sequence $O_{31} \to O_{11} \to O_{12} \to O_{32} \to O_{13} \to O_{33}$.

The idea of rule **H2** is intended to give higher dispatching priorities to those jobs which have longer remaining processing times. The reason is that jobs with longer remaining processing times tend to be completed later. To reduce total completion time, operations of these jobs are thus given higher priorities while allocating machine capacity to operations.

See Fig. 2, decoding the example $\boldsymbol{S_{JOB}}$ chromosome ($J_3 \to J_2 \to J_1$) by successively applying heuristic rules **H1 and H2** results in the following outcomes. Jobs $\{J_3 \to J_1\}$ is assigned to cell U^2, and the operation-sequencing decision in cell U^2 is $O_{31} \to O_{11} \to O_{12} \to O_{32} \to O_{13} \to O_{33}$.

Decoding rule H3: operation-to-machine assignment

Heuristic rule **H3** is used to make the operation-to-machine assignment decision for each cell. The idea of rule **H3** is to balance the workload of each machine by adopting a *try-and-select* approach. That is, we try to assign an operation (O_{ij}) to each possible machine in the cell; and the machine which has the lowest *cumulative load* is selected. The *cumulative load* of a machine is the total processing times of all assigned operations and the presently tried operation O_{ij}. Based on the idea, the pseudo code of **H3** is presented below followed by an example.

Procedure H3

input: *Job-Cell-Op* table, job-to-cell assignment, operation sequence in each cell l

output: operation-to-machine assignment in each cell l

notation:

Assume that h jobs are assigned to cell l; as a result, it involves s operations in total.

Denote the s operations as an ordered set $S = \{O_{\alpha(1),\beta(1)} \to \cdots \to O_{\alpha(s),\beta(s)}\}$, where $O_{\alpha(d),\beta(d)}$ is the d-th element in the ordered set, which is an operation that can be interpreted by referring that $O_{1,2}$ denotes the 2nd operation of job J_l.

begin

for $l = 1$ **to** q **do**

  $L_l^k \leftarrow 0\ (1 \le k \le H_l)$; /*initial loading of each machine is 0*/

  for each operation $O_{\alpha(d),\beta(d)}$ in the ordered set S, $1 \le d \le s$

    $i \leftarrow \alpha(d)$, $j \leftarrow \beta(d)$; /*change notation*/

    for $k = 1$ **to** H_l **do** /*for each machine in cell l*/

      compute $T_l^k = L_l^k + p_{ij}^{lk}$; /*try loading the operation to each machine*/

    end

    $k^* = Arg(\min_k T_l^k)$; /*pick the machine with lightest cumulative loading*/

    place operation $O_{\alpha(d),\beta(d)}$ in set $S_{k^*}^l$; /*assign operation to machine k^**/

    $L_l^{k^*} \leftarrow T_l^{k^*}$; /*update machine loading*/

  end

end

output operation-to-machine assignment, set S_k^l $(1 \le l \le q, 1 \le k \le H_l)$;

end

We now explain procedure **H3** by the example DFJS problem shown in Table 1(a). By applying rules **H1** and **H2**, job J_1 and J_3 now have been assigned to cell U^2; and the operation sequence in cell U^2 is $O_{31} \to O_{11} \to O_{12} \to O_{32} \to O_{13} \to O_{33}$. Detail steps of applying rule **H3** to assign the six operations in cell U^2 are shown in Table 3.

Table 3 An example of implementing heuristic rule **H3**

	M^{21}		M^{22}		Job assignment
	p_{ij}^{21}	C_Load	p_{ij}^{22}	C_Load	
Initial		0		0	
O_{31}					
Try and Select	–	–	2	[2]	M^{22}
Update C_Load		0		2	
O_{11}					
Try and Select	–	–	2	[4]	M^{22}
Update C_Load		0		4	
O_{12}					
Try and Select	4	[4]	5	9	M^{21}
Update C_Load		4		4	
O_{32}					
Try and Select	–	–	2	[6]	M^{22}
Update C_Load		4		6	
O_{13}					
Try and Select	3	[7]	4	10	M^{21}
Update C_Load		7		6	
O_{33}					
Try and Select	1	[8]	3	9	M^{21}
Update C_Load		8		6	

In Table 3, the operation sequence ($O_{31} \rightarrow O_{11} \rightarrow O_{12} \rightarrow O_{32} \rightarrow O_{13} \rightarrow O_{33}$) forms the first column. The 1st row indicates that the cumulative loads (*C_Load*) for the two machines are initially both 0. The 2nd row indicates that operation O_{31} is assigned to machine M^{22} because machine M^{21} cannot process O_{31}. The 3rd row updates the *C_Load* of each machine after assigning O_{31} to M^{22}. The 4th row indicates that operation O_{11} is assigned to machine M^{22} because machine M^{21} cannot process O_{11}. The 5th row updates the *C_Load* of each machine after assigning O_{11} to M^{22}. The 6th row indicates that operation O_{12} is assigned to machine M^{12} due to having lower *C_Load*. Repeatedly applying the above procedure, each row in Table 3 can be obtained. Notice that in the procedure a random selection method is used for resolving the tie-breaking issue. The resulting operation-to-machine assignment decision is shown in the last column of the table.

See Fig. 2, by successively applying heuristic rules **H1, H2, and H3** to decode the example $\boldsymbol{S_{JOB}}$ chromosome ($J_3 \rightarrow J_2 \rightarrow J_1$). Its three DFJS scheduling decisions (job-to-cell assignment, operation-to-machine assignment, and operation-sequencing) can be revealed. Based on three DFJS scheduling decisions, we can generate its Gantt chart (i.e., the exact scheduling solution) as shown in Fig. 5a.

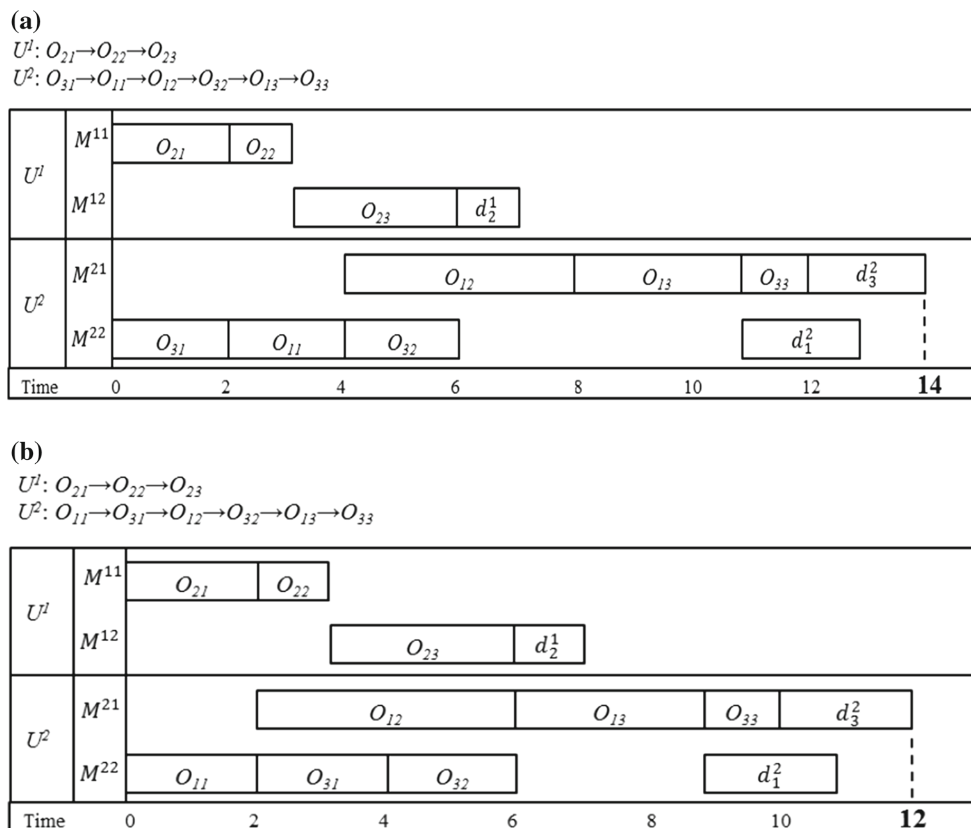

Fig. 5 Gantt chart of $\boldsymbol{S_{JOB}}$ chromosome **a** before applying refinement method **b** after applying refinement method

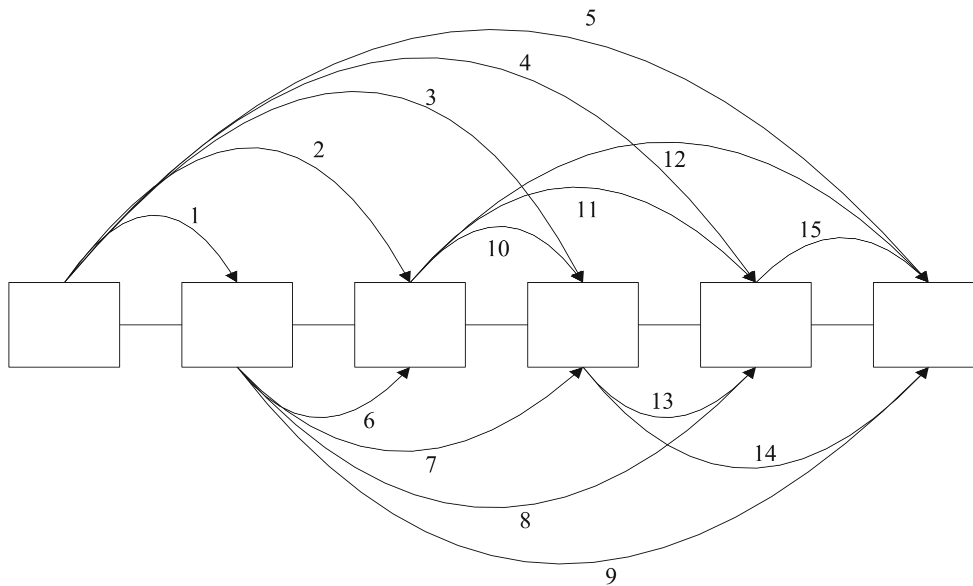

Fig. 6 Define the sequence of operation swapping in the refinement method

The ideas underlying the above three heuristic rules are summarized below. Rule **H1** attempts to balance the workload of each cell. Rule **H3** attempts to balance the workload of each machine. Rule **H2** attempts to give higher priority to the operations which tend to have a higher impact on the overall completion time.

Refinement and encoding of 3D solutions

This section presents the cell refinement method and 3D-to-1D encoding method shown in Fig. 3. Given a 3D scheduling solution as input, the cell refinement method is used to generate a *refined* 3D solution for obtaining better scheduling performance. In turn, the 3D refined solution is encoded to obtain a new chromosome (i.e., shadow chromosome). The pseudo code for each of the two methods is presented below and followed by examples.

Cell refinement method

Adapted from De Giovanni and Pezzella (2010), the cell refinement method is designed to improve the scheduling performance of the *critical cell* by changing the *operation-sequencing* as well as the *operation-to-machine assignment* in the cell. Notice that the *critical cell* is the cell with maximum makespan; for example, cell U^2 is the critical cell in Fig. 5a.

Procedure Cell Refinement

input: a scheduling solution *S*, Gantt chart of *S*

output: a refined scheduling solution *R*

begin

compute makespan (C^l) for each cell *l* for solution *S*;

step 1: $l^* = Arg\,(\max_{1\le l\le q} C^l)$; /*determine critical cell l^**/

$C^{l^*} = \max_{1\le l\le q} C^l$; /*the longest makespan of the scheduling solution*/

denote the operation sequence of cell l^* as $\Psi = \{O^{l^*}_{\alpha(1),\beta(1)} \rightarrow \cdots \rightarrow O^{l^*}_{\alpha(s),\beta(s)}\}$;

$\Omega \leftarrow \Psi$; /*create a temporary set for storing an operation sequence in cell l^**/

for *i* = 1 **to** *s* **do**

for *j* = i+1 **to** *s* **do**

update Ω by swapping $O^{l^*}_{\alpha(i),\beta(i)}$ and $O^{l^*}_{\alpha(j),\beta(j)}$ in Ψ;

apply decoding rule **H3** to Ω and obtain a scheduling solution *R*;

evaluate the makespan of *R*, denoted by *MK(R)*;

If ($MK(R) < C^{l^*}$) **then**

$\Psi \leftarrow \Omega$;

$C^{l^*} \leftarrow MK(R)$;

go to step 1;

If ($MK(R) = C^{l^*}$) **then**

$\Psi \leftarrow \Omega$;

end

end

output the scheduling solution *R*;

end

Notice that the Gantt chart of cell U^2 is implicitly determined by the operation sequence within the cell. See Fig. 2, the application of rule **H2** yields the operation sequence within cell $U^2(O_{31} \rightarrow O_{11} \rightarrow O_{12} \rightarrow O_{32} \rightarrow O_{13} \rightarrow O_{33})$; in turn, the application of rule **H3** yields the operation-to-machine assignment decision. These two decisions result in the Gantt chart of Fig. 5a. That is, changing the *operation sequence within the cell* shall change the Gantt chart (i.e., scheduling performance).

The refinement method is designed to *exhaustively swap* any two operations on the operation sequence ($O_{31} \rightarrow O_{11} \rightarrow O_{12} \rightarrow O_{32} \rightarrow O_{13} \rightarrow O_{33}$) within critical cell U^2 in order to obtain better scheduling performance. To avoid infeasible swapping, we model the operation sequence by replacing each operation by its associated job. Accordingly, operation sequence $O_{31} \rightarrow O_{11} \rightarrow O_{12} \rightarrow O_{32} \rightarrow O_{13} \rightarrow O_{33}$ is represented by $J_3 \rightarrow J_1 \rightarrow J_1 \rightarrow J_3 \rightarrow J_1 \rightarrow J_3$. Notice that each job appears several times in the sequence, in which the nth appearance of a job denotes its nth operation. Now, assume the first two operations are swapped; this yields a new sequence $J_1 \rightarrow J_3 \rightarrow J_1 \rightarrow J_3 \rightarrow J_1 \rightarrow J_3$ which can be interpreted as $O_{11} \rightarrow O_{31} \rightarrow O_{12} \rightarrow O_{32} \rightarrow O_{13} \rightarrow O_{33}$.

With sequence representation $O_{31} \rightarrow O_{11} \rightarrow O_{12} \rightarrow O_{32} \rightarrow O_{13} \rightarrow O_{33}$, the swapping of O_{31} and O_{32} is *infeasible* and shall be prohibited. Yet, with sequence representation $J_3 \rightarrow J_1 \rightarrow J_1 \rightarrow J_3 \rightarrow J_1 \rightarrow J_3$, the swapping of O_{31} and O_{32} shall be interpreted as the swapping of J_3 and J_3, which is an *unchanged swapping* (i.e., *not meaningful swapping*) and needs not to be carried out.

The operation swapping is carried out in an *exhaustive* and *dynamic-updating* manner. Consider the operation sequence $J_3 \rightarrow J_1 \rightarrow J_1 \rightarrow J_3 \rightarrow J_1 \rightarrow J_3$ which is a 6-element array (i.e., the value of each element denotes an operation). A swap denotes a pair of two elements. An exhaustive swapping theoretically involves C(6,2) = 15 swaps in total. See Fig. 6, each of these 15 swaps is indexed in a predefined sequence and carried out accordingly. In performing these swaps, if a swap is not meaningful, we just skip it. While a swap is meaningful, we compute the resulting makespan. If the makespan improves, the swap is regarded as "dominant"; and the operation sequence must be *updated*; then the remaining swaps are carried out on the *updated* operation sequence.

Example input and output of the refinement methods are illustrated below. See Fig. 5a, the input is an operation sequence $J_3 \rightarrow J_1 \rightarrow J_1 \rightarrow J_3 \rightarrow J_1 \rightarrow J_3$ whose resulting makespan is 14. See Fig. 5b, after exhaustively carrying out the 15 swaps, we find that the output of the refinement method yields an operation sequence $J_1 \rightarrow J_3 \rightarrow J_1 \rightarrow J_3 \rightarrow J_1 \rightarrow J_3$ whose resulting makespan is 12.

Noticeably, after carrying out the refinement method, the makespan of the critical cell may be reduced. As a result, another cell may turn out to be the new critical cell. Then, the refinement method must be accordingly performed on the new critical cell. This refinement procedure is iteratively carried out until the *ultimate* critical cell is determined and its minimum makespan is obtained.

Encoding method for generating shadow chromosomes

The 3D-to-1D encoding method attempts to convert a 3D scheduling solution obtained by the aforementioned refinement method into a 1D chromosome (called shadow chromosome). The pseudo code of the 3D-to-1D encoding method is listed below and followed by an example.

Procedure 3D-to-1D encoding

input: a scheduling solution S, its critical cell l^* and operation sequence in each cell l

output: a job sequence $\Omega = \{\pi(i)|n\}$, where $\pi(i)$ denotes i-th job in the sequence

notation:

Assume that cell l involves h jobs and as a result involves s operations in total.

Denote the s operations in cell l as an ordered set $S^l = \{O^l_{\alpha(1),\beta(1)} \rightarrow \cdots \rightarrow O^l_{\alpha(s),\beta(s)}\}$, where $O^l_{\alpha(d),\beta(d)}$ is the d-th element in the ordered set, which is an operation that can be interpreted by referring that $O^2_{3,1}$ denotes the 1st operation of job J_3 in cell U^2.

begin

$\Psi \leftarrow S^{l^*}$; /*place operation sequence of critical cell in Ψ*/

while $(l \neq l^*)$ **do**

$\Psi \leftarrow \Psi \cup S^l$; /*successively place op-sequence of each non-critical cell in Ψ*/

end

Denote the obtained set as $\Psi = \{O^{l^*}_{\alpha(1),\beta(1)} \rightarrow \cdots \rightarrow O^q_{\alpha(z),\beta(z)}\}$;

$i \leftarrow 1$;

$\pi(i) \leftarrow \alpha(1)$; /*determine the 1st job in the output job sequence*/

$\Omega = \{\pi(i)|i\}$; /*create a set for storing output job sequence*/

for $d = 2$ **to** z **do**

if $(\alpha(d) \notin \Omega)$ **then**

$i \leftarrow i + 1$;

$\pi(i) \leftarrow \alpha(d)$; /*update Ω set*/

end

output $\Omega = \{\pi(i)|n\}$; /*a sequence of all jobs*/

end

We now use an example to explain the 3D-to-1D encoding method by considering the 3D scheduling solution in Fig. 5b, which is the output of the refinement method. Its operation sequence within cell U^2 is $O_{11} \rightarrow O_{31} \rightarrow O_{12} \rightarrow O_{32} \rightarrow O_{13} \rightarrow O_{33}$; and its operation sequence within cell U^1 is $O_{21} \rightarrow O_{22} \rightarrow O_{23}$.

Firstly, we form an "*aggregated*" operation sequence by placing the operation sequence of the critical cell (U^2) in the first block and *successively* place the operation sequences of the other cells (U^1) in the remaining blocks. This yields an aggregated operation sequence $\boxed{O_{11} \rightarrow O_{31} \rightarrow O_{12} \rightarrow O_{32} \rightarrow O_{13} \rightarrow O_{33}} \rightarrow \boxed{O_{21} \rightarrow O_{22} \rightarrow O_{23}}$. Secondly, we keep only the first operation of each job and yield a concise sequence $\boxed{O_{11} \rightarrow O_{31}} \rightarrow \boxed{O_{21}}$; in turn each operation is replaced by its associated job. As a result, this yields a job sequence $J_1 \rightarrow J_3 \rightarrow J_2$ which is called a *shadow chromosome*.

The reason why we generate shadow chromosomes in such a way is explained below. Remind that the shadow chromosome $J_1 \rightarrow J_3 \rightarrow J_2$ indirectly derives from the chromosome $J_3 \rightarrow J_2 \rightarrow J_1$ in Fig. 1c by the following steps. Firstly, the chromosome $J_3 \rightarrow J_2 \rightarrow J_1$ is *decoded* and yields a 3D scheduling solution in Fig. 2, whose job-to-cell assignment decisions are $U^1 \leftarrow \{J_2\}$ and $U^2 \leftarrow \{J_3, J_1\}$. Secondly, given the job-to-cell assignment decision, the *refinement* method attempts to improve the scheduling performance by changing the operation-sequencing and the operation-to-machine assignment decisions. Namely, the refinement method is designed to obtain a *near-optimum* schedule for the job-to-cell assignment decision.

Now to further improve the scheduling performance, we need to change the *job-to-cell assignment decision*. And the way of generating a shadow chromosome ($J_1 \rightarrow J_3 \rightarrow J_2$) is for providing a new job-to-cell assignment decision. Applying heuristic rule **H1** to decode the shadow chromosome ($J_1 \rightarrow J_3 \rightarrow J_2$), we tend to place $\{J_1, J_3\}$ which are originally in the critical cell U^2 to different cells. That is, following the job sequence $J_1 \rightarrow J_3 \rightarrow J_2$, job J_1 tends to be the first allocated job of one cell; and job J_3 tends to be the first allocated job of another cell. This implies that we attempt to "break" the critical cell and generate a new job-to-cell assignment decision; as a result, we may come out a new critical cell with better scheduling performance.

Herein we explain why such a new chromosome is named as a *shadow* chromosome. The term "shadow" is adopted from the "shadow cabinet" in British political system. As known, a shadow cabinet intends to criticize the policies of the government and offer *alternative* policies. Likewise, the shadow chromosome is designed to offer an *alternative* job-to-cell allocation policy by "breaking" the critical cell.

GA_JS algorithm

In this section, we present the procedure of the proposed algorithm GA_JS, in which N, ρ_c, ρ_m, k_b, ϕ_f, and ϕ_{max} are given parameters and a job in a chromosome is called a *gene*.

Procedure *GA_JS*

Create an initial population $P = \{N$ chromosomes$\}$ by randomly sequencing jobs

Evaluate each chromosome in P by decoding methods **H1**, **H2**, and **H3**

Repeat (outer loop)

  Repeat (inner loop)

    Select two parent chromosomes from P randomly

    Apply *crossover* with probability ρ_c to the two chromosomes and update P

    Apply *mutation* with probability ρ_m to the two chromosomes and update P

  Until inner loop iterates N times

  For each of the best k_b chromosomes (say, h) in P

    Apply *Refinement routine to* chromosomes h

    Apply 3D-to-1D Encoding to h and generate *shadow chromosome s*

    Replace chromosomes h by s in population P

  Endfor

Until either one of the following two termination conditions appears;

- The up-to-date best solution has not changed for ϕ_f outer loop iterations
- The total number of outer loop iterations equals ϕ_{max}

Output the up-to-date best solution

The crossover operator, designed to generate two *new* chromosomes from two *existing* chromosomes (i.e. parent chromosomes), is a *one-point crossover* (Gen and Cheng 1997). As shown in Fig. 7a, it randomly divides the two parent chromosomes (A, B) into two substrings (A_1, A_2, B_1, B_2) and two new chromosomes are generated by two steps. Consider parent chromosome A as an example. In step 1, the

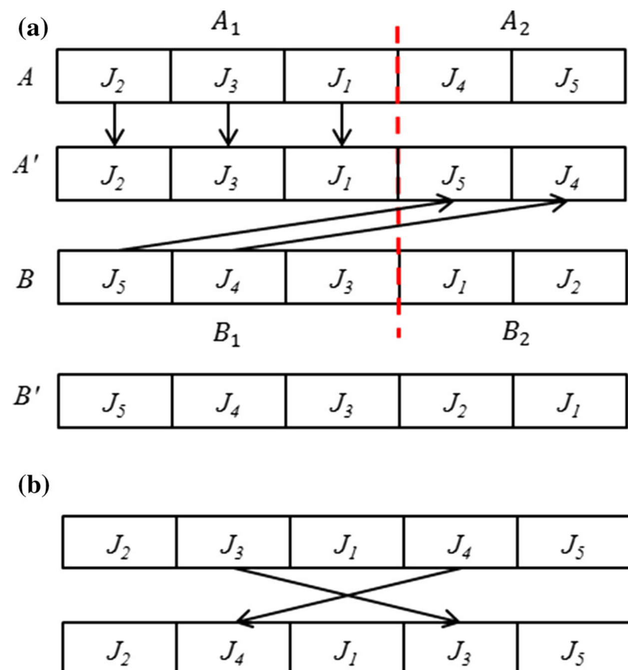

Fig. 7 **a** Crossover operator, **b** mutation operator

first substring (A_1) is maintained ($J_2 \rightarrow J_3 \rightarrow J_1$). In step 2, the *gene value sequence* ($J_4 \rightarrow J_5$) in the second substring (A_2) is modified into a new sequence ($J_5 \rightarrow J_4$), which is obtained by following the gene value sequence ($J_5 \rightarrow J_4 \rightarrow J_3 \rightarrow J_1 \rightarrow J_2$) of the other parent chromosome (B). As a result, a new chromosome A' (i.e., $J_2 \rightarrow J_3 \rightarrow J_1 \rightarrow J_5 \rightarrow J_4$) is generated. Accordingly, the other new chromosome B' can be generated by the crossover operation.

The mutation operator is designed to generate one new chromosome from one existing chromosome (i.e. parent chromosome). It randomly chooses two genes and exchanges their gene values. As shown in Fig. 7b, two genes are randomly selected from the parent chromosome ($J_2 \rightarrow \boxed{J_3} \rightarrow J_1 \rightarrow \boxed{J_4} \rightarrow J_5$) and their gene values are exchanged to generate a new chromosome ($J_2 \rightarrow \boxed{J_4} \rightarrow J_1 \rightarrow \boxed{J_3} \rightarrow J_5$).

Numerical experiments

This section compares the empirical performance of the proposed algorithm GA_JS against *IGA* which is proposed by De Giovanni and Pezzella (2010). Notice that the completion time of all jobs (called makepan) in the DFJS problems is taken as the performance measure.

Experiment design

To make a *compatible* comparison, we repeat the 2-cell, 3-cell and 4-cell experiments reported in De Giovanni and Pezzella (2010). Each of these three experiments includes 23 DFJS instances; 15 replicates are carried out for each DFJS instance in the GA_JS. Genetic parameters of GA_JS are set the same as that of *IGA* in each experiment (Table 4).

Algorithm GA_JS is implemented in C++ and run on a personal computer equipped with a 3.0 GHz AMD Athlon(tm) II*4640 processor and 4GB RAM. In contrast, IGA is implemented in C++ and run on a personal computer equipped with 2.0 GHz Intel Core2 processor and 2 GB RAM.

Table 4 Genetic parameters

Type	DFJS 2-cell	DFJS 3(4)-cell
N	50	50
ρ_c	0.9	0.9
ρ_m	0.9	0.9
k_b	3	3
ϕ_f	225	188
ϕ_{max}	300	250

The three experiment results are shown in Table 5. Each column in the table is explained below. See Table 5, the first three columns respectively give (1) the name of instance, (2) the number of jobs, and (3) the number of operations per job. The 4th column *LB* reports a lower bound proposed by De Giovanni and Pezzella (2010). The formula for determining the *LB* is as shown below.

$$LB = \max_{i \in \boldsymbol{J}} \left\{ \min_{l \in \boldsymbol{U}} \left\{ \sum_{j=1}^{n_i} \min_{k \in \boldsymbol{M}} \left\{ p_{ij}^{lk} \right\} + d_i^l \right\} \right\}$$

The 5th column *MK* denotes the best makespan of all replicates in an instance. The 6th column *Av.* denotes the average makespan of all replicates in an instance. The 7th column $T(s)$ denotes the average computation time in seconds. The 8th column $Gap\% = \frac{MK-LB}{LB}$ reports the gap between *MK* and *LB*, and the remaining columns in the table are defined accordingly.

Performance comparison

We first compare GA_JS against *IGA* in terms of *Gap%*. See Table 5, GA_JS outperforms *IGA* in each of the three experiments. In 2-cell experiment, the *Gap%* of of GA_JS is 10.1 % better than that of *IGA* (12.4 %). In 3-cell experiment, the *Gap%* of GA_JS is 0.9 % better than that of *IGA* (2.0 %). In 4-cell experiment, the *Gap%* of GA_JS is 0.0 % better than that of *IGA* (0.2 %).

We then compare GA_JS against *IGA* in terms of average solution quality (*Av.*). To statistically justify the performance difference, we use Wilcoxon signed rank test. In 2-cell experiment, GA_JS outperforms *IGA* with p-value $= 0.002 < 0.05$. In 3-cell experiment, GA_JS outperforms *IGA* with p-value $= 0.018 < 0.05$. In 4-cell experiment, the difference between GA_JS and *IGA* is not statistically significant with p-value $= 0.091 > 0.05$.

We further compare GA_JS against *IGA* in terms of computation time. Such a comparison is for reference only because these two algorithms as stated above are run on different computers. See Table 5, GA_JS requires less computation time than *IGA* in each of the three experiments. In 2-cell experiment, the average computation time of GA_JS is 38.4 s, faster than that of *IGA* (79.6 s). In 3-cell experiment, the average computation time of GA_JS is 15.1 s, faster than that of *IGA* (27.9 s). In 4-cell experiment, the average computation time of GA_JS is 9.9 s, faster than that of *IGA* (16.2 s).

Analysis of experiment results

According to experiment results, GA_JS appears to outperform *IGA*. Yet, their performance differences become

Table 5 Performance comparison of GA_JS against IGA in 2, 3, and 4-cell scenarios

Inst.	Jobs	Ops.	LB	GA_JS(2-cell)				IGA(2-cell)				GA_JS(3-cell)				IGA(3-cell)				GA_JS(4-cell)				IGA(4-cell)			
				MK	*Av.*	$T(s)$	*Gap%*	*MK*	*Av.*	$T(s)$	*Gap%*	*MK*	*Av.*	$T(s)$	*Gap%*	*MK*	*Av.*	$T(s)$	*Gap%*	*MK*	*Av.*	$T(s)$	*Gap%*	*MK*	*Av.*	$T(s)$	*Gap%*
la01	10	5	413	413	413.0	7.3	0.0	413	413.0	12.0	0.0	413	413.0	3.7	0.0	413	413.0	4.6	0.0	413	413.0	3.8	0.0	413	413.0	1.8	0.0
la02	10	5	394	394	394.0	6.5	0.0	394	394.0	11.2	0.0	394	394.0	3.8	0.0	394	394.0	3.6	0.0	394	394.0	3.7	0.0	394	394.0	1.8	0.0
la03	10	5	349	349	349.0	7.2	0.0	349	349.0	10.8	0.0	349	349.0	3.8	0.0	349	349.0	3.8	0.0	349	349.0	3.7	0.0	349	349.0	2.2	0.0
la04	10	5	369	369	369.0	6.6	0.0	369	369.0	11.4	0.0	369	369.0	3.8	0.0	369	369.0	3.8	0.0	369	369.0	3.7	0.0	369	369.0	2.0	0.0
la05	10	5	380	380	380.0	6.6	0.0	380	380.0	8.0	0.0	380	380.0	3.7	0.0	380	380.0	2.6	0.0	380	380.0	3.6	0.0	380	380.0	1.0	0.0
la06	15	5	413	424	431.1	24.8	2.7	445	449.6	45.8	7.7	413	413.0	8.6	0.0	413	413.0	17.4	0.0	413	413.0	6.6	0.0	413	413.0	9.0	0.0
la07	15	5	376	398	406.1	26.5	5.9	412	419.2	50.2	9.6	376	376.0	9.9	0.0	376	376.0	18.2	0.0	376	376.0	7.3	0.0	376	376.0	9.6	0.0
la08	15	5	369	406	418.2	26.2	10.0	420	427.8	53.8	13.8	369	369.0	9.1	0.0	369	369.0	19.6	0.0	369	369.0	6.7	0.0	369	369.0	12.6	0.0
la09	15	5	382	447	459.8	27.8	17.0	469	474.6	45.2	22.8	382	382.0	10.2	0.0	382	387.4	17.8	0.0	382	382.0	7.3	0.0	382	382.0	11.6	0.0
la10	10	5	443	443	443.4	24.4	0.0	445	448.6	45.0	0.5	443	443.0	9.2	0.0	443	443.0	17.0	0.0	443	443.0	6.6	0.0	443	443.0	7.8	0.0
la11	20	5	413	548	556.4	63.8	32.7	570	571.6	126.0	38.0	413	418.0	27.3	0.0	425	436.8	50.6	2.9	413	413.0	13.7	0.0	413	413.0	29.6	0.0
la12	20	5	408	480	491.7	62.5	17.6	504	508.0	116.0	23.5	408	408.0	19.3	0.0	408	408.0	44.6	0.0	408	408.0	13.4	0.0	408	408.0	26.6	0.0
la13	20	5	382	533	538.0	67.2	39.5	542	552.2	125.4	41.9	398	409.5	26.7	4.2	419	430.2	45.8	9.7	382	382.0	15.9	0.0	382	386.0	27.6	0.0
la14	20	5	443	542	555.5	62.9	22.3	570	576.0	122.2	28.7	443	443.0	20.3	0.0	443	448.8	48.8	0.0	443	443.0	12.9	0.0	443	443.0	29.8	0.0
la15	20	5	378	562	566.9	65.7	48.7	584	588.8	119.6	54.5	420	426.8	28.0	11.1	451	456.0	42.2	19.3	378	381.6	18.5	0.0	397	402.0	28.8	5.0
la16	10	10	717	717	717.0	53.3	0.0	717	717.0	140.2	0.0	717	717.0	21.8	0.0	717	717.0	36.0	0.0	717	717.0	14.6	0.0	717	717.0	20.2	0.0
la17	10	10	646	646	646.0	53.9	0.0	646	646.0	112.6	0.0	646	646.0	21.2	0.0	646	646.0	31.6	0.0	646	646.0	13.4	0.0	646	646.0	16.4	0.0
la18	10	10	663	663	663.0	54.6	0.0	663	663.0	132.4	0.0	663	663.0	21.9	0.0	663	663.0	36.8	0.0	663	663.0	14.8	0.0	663	663.0	24.4	0.0
la19	10	10	617	617	617.5	59.6	0.0	617	617.2	147.2	0.0	617	617.0	21.3	0.0	617	617.0	62.4	0.0	617	617.0	13.3	0.0	617	617.0	33.0	0.0
la20	10	10	756	756	756.0	53.8	0.0	756	756.0	99.8	0.0	756	756.0	21.1	0.0	756	756.0	34.2	0.0	756	756.0	13.9	0.0	756	756.0	18.0	0.0
mt06	6	6	47	47	47.0	3.5	0.0	47	47.0	2.0	0.0	47	47.0	2.4	0.0	47	47.0	1.0	0.0	47	47.0	2.0	0.0	47	47.0	0.2	0.0
mt10	10	10	655	655	655.0	54.6	0.0	655	655.0	173.0	0.0	655	655.0	22.4	0.0	655	655.0	50.0	0.0	655	655.0	14.4	0.0	655	655.0	31.2	0.0
mt20	20	10	387	529	547.3	64.6	36.7	560	566.0	121.2	44.7	408	418.5	26.8	5.4	439	442.6	48.2	13.4	387	387.0	14.5	0.0	387	388.4	27.0	0.0
Av.						38.4	10.1			79.6	12.4			15.1	0.9			27.9	2.0			9.9	0.0			16.2	0.2

smaller when we increase the number of cells. In 2-cell experiment, the difference of *Gap%* is 2.3 % = 12.4 %–10.1 %. In 3-cell experiment, the difference of *Gap%* is 1.1 % = 2.0 %–0.9 %. In 4-cell experiment, the difference of *Gap%* is 0.2 % = 0.2 %–0.0 %.

The reason why the performance differences between GA_JS and *IGA* monotonically decrease against the number of cells is explained below. As stated, we have 23 DFJS instances in the experiments. In each DFJS instance, the number of jobs and the number of operations are always kept the same, even in different cell environment. This implies that the cell loading becomes lower while we increase the number of cells. That is, in 4-cell environment, the capacity supply may become much higher than the capacity demand. As a result, GA_JS can find out the lower bound (*LB*) solution in 23 instances; and *IGA* can find out *LB* solutions in 22 instances. Therefore, the reported comparison in 3-cell/4-cell is concerned with *light-loading* cases. To make a comparison in *high-loading* cases for 3-cell/4-cell environments, we need to increase the number of jobs and operations. However, the high-loading experiment results of *IGA* for 3-cell/4-cell environments are not reported in literature and cannot be compared.

Concluding remarks

This paper proposes a genetic algorithm GA_JS for solving distributed and flexible job-shop scheduling (DFJS) problems. A DFJS problem involves three scheduling decisions: (1) job-to-cell assignment, (2) operation-sequencing, and (3) operation-to-machine assignment. Therefore, solving a DFJS problem is essentially a 3D *solution space* search problem, in which each dimension represents a type of decision.

The *GA_JS* algorithm is developed by proposing a *new* and *concise* chromosome representation $\boldsymbol{S_{JOB}}$, which models a 3D scheduling solution by a *1-dimensional scheme* (i.e., a sequence of all jobs to be scheduled). That is, the chromosome space is 1D and the solution space is 3D. In GA_JS, we develop a 1D-to-3D *decoding* method to convert a 1D chromosome into a 3D solution. In addition, given a 3D solution, we use a *refinement* method to improve the scheduling performance and subsequently develop a 3D-to-1D *encoding* method to convert the refined 3D solution into a 1D chromosome.

The 1D-to-3D decoding method is designed to obtain a "good" 3D solution which tends to be *load-balanced* in terms of job-to-cell assignment and operation-to-machine assignment decisions. The refinement method is designed to find a *near-optimum* schedule for a given job-to-cell assignment decision. In contrast, the 3D-to-1D encoding method is deigned to *change the job-to-cell assignment* decision by "breaking" the critical cell for generating a new chromosome (called *shadow chromosomes*).

Numerical experiments reveal that GA_JS outperforms *IGA* (the up-to-date best-performing genetic algorithm in solving DFJS problems) in 2-cell and 3-cell environments. However, GA_JS and *IGA* appear to perform equally well in 4-cell environment. This is due to that the 4-cell experiments reported in prior studies are concerned with light-loading cases.

In future research, we attempt to develop a brand-new chromosome representation or extend the $\boldsymbol{S_{JOB}}$ representation to solve a DFJS problem which includes the decision of when to carry out preventive maintenance (PM). Such a scheduling problem is in essence a 4-dimensional solution space search problem. How to model the PM decision as well as the three DFJS scheduling decisions makes room for further study.

Acknowledgements This research is financially supported by National Science Council (Taiwan), under a research contract NSC-100-2221-E-009-059-MY3.

References

Baykasoğlu, A. (2002). Linguistic-based meta-heuristic optimization model for flexible job shop scheduling. *International Journal of Production Research*, *40*(17), 4523–4543.

Bożejko, W., Uchroński, M., & Wodecki, M. (2010). Parallel hybrid metaheuristics for the flexible job shop problem. *Computers & Industrial Engineering*, *59*, 323–333.

Brandimarte, P. (1993). Routing and scheduling in a flexible job shop by tabu search. *Annals of Operations Research*, *41*, 157–183.

Bruker, P., & Schlie, R. (1990). Job-shop scheduling with multi-purpose machines. *Computing*, *45*, 369–375.

Chan, F. T. S., Chung, S. H., & Chan, P. L. Y. (2005). An adaptive genetic algorithm with dominated genes for distributed scheduling problems. *Expert Systems with Applications*, *29*, 364–371.

Chan, F. T. S., Chung, S. H., & Chan, P. L. Y. (2006). Application of genetic algorithms with dominant genes in a distributed scheduling problem in flexible manufacturing systems. *International Journal of Production Research*, *44*(3), 523–543.

Choi, I. C., & Choi, D. S. (2002). A local search algorithm for jobshop scheduling problems with alternative operations and sequence-dependent setups. *Computers & Industrial Engineering*, *42*, 43–58.

Chou, Y. C., & Cheng, H. H. (2010). An autonomic mobile agent-based system for distributed job shop scheduling. *IEEE/ASME international conference on mechatronics and embedded systems and applications* (pp. 113–118).

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

De Giovanni, L., & Pezzella, F. (2010). An improved genetic algorithm for the distributed and flexible job-shop scheduling problem. *European Journal of Operational Research*, *200*, 395–408.

Gao, J., Sun, L., & Gen, M. (2008). A hybrid genetic and variable neighborhood descent algorithm for flexible job shop scheduling problems. *Computers & Operations Research*, *35*, 2892–2907.

Garey, M. R., Johnson, D. S., & Sethi, R. (1976). The complexity of flowshop and jobshop scheduling. *Mathematics of Operations Research*, *1*(2), 117–129.

Gen, M., & Cheng, R. (1997). *Genetic algorithms and engineering design*. New York: Wiley.

Gen, M., Cheng, R., & Lin, L. (2008). *Network models and optimization: Multiobjective genetic algorithm approach*. London: Springer.

González, M. A., Vela, C. R., & Varela, R. (2013). An efficient memetic algorithm for the flexible job shop with setup times. In: *Proceedings of the 23th international conference on automated planning and scheduling* (pp. 91–99).

Gutiérrez, C., & García-Magariño, I. (2011). Modular design of a hybrid genetic algorithm for a flexible job-shop scheduling problem. *Knowledge-Based Systems*, *24*, 102–112.

Hmida, A. B., Haouari, M., Huguet, M. J., & Lopez, P. (2010). Discrepancy search for the flexible job shop scheduling problem. *Computers & Operations Research*, *37*, 2192–2201.

Ho, N. B., & Tay, J. C. (2004). GENACE: An efficient cultural algorithm for solving the flexible job-shop problem. *Proceedings of the IEEE congress on evolutionary computation* (pp. 1759–1766).

Hurink, J., Jurisch, B., & Thole, M. (1994). Tabu search for the job-shop scheduling problem with multi-purpose machines. *OR Spektrum*, *15*, 205–215.

Jia, H. Z., Nee, A. Y. C., Fuh, J. Y. H., & Zhang, Y. F. (2003). A modified genetic algorithm for distributed scheduling problems. *Journal of Intelligent Manufacturing*, *14*, 351–362.

Jia, H. Z., Nee, A. Y. C., Fuh, J. Y. H., & Zhang, Y. F. (2007). Integration of genetic algorithm and Gantt chart for job shop scheduling in distributed manufacturing systems. *Computer & Industrial Engineering*, *53*, 313–320.

Jia, S., & Hu, Z. H. (2014). Path-relinking tabu search for the multi-objective flexible job shop scheduling problem. *Computers & Operations Research*, *47*, 11–26.

Jinyan, M., Chai, S. Y., & Youyi, W. (1995). FMS jobshop scheduling using lagrangian relaxation method. *Proceedings of IEEE international conference on robotics and automation* (pp. 490–495).

Kacem, I., Hammadi, S., & Borne, P. (2002a). Approach by localization and multiobjective evolutionary optimization for flexible job-shop scheduling problems. *IEEE Transactions on Systems, Man and Cybernetics, Part C: Applications and Reviews*, *32*(1), 1–13.

Kacem, I., Hammadi, S., & Borne, P. (2002b). Pareto-optimality approach for flexible job-shop scheduling problems: Hybridization of evolutionary algorithms and fuzzy logic. *Mathematics and Computers in Simulation*, *60*, 245–276.

Kim, K.-H., & Egbelu, P. J. (1999). Scheduling in a production environment with multiple process plans per job. *International Journal of Production Research*, *37*(12), 2725–2753.

Lin, L., & Gen, M. (2006). Node-based genetic algorithm for communication spanning tree problem. *IEICE Transactions on Communications*, *E89–B*(4), 1091–1098.

Mastrolilli, M., & Gambardella, L. M. (2000). Effective neighbourhood functions for the flexible job shop problem. *Journal of Scheduling*, *3*, 3–20.

Pezzella, F., Morganti, G., & Ciaschetti, G. (2008). A genetic algorithm for the flexible job-shop scheduling problem. *Computers & Operations Research*, *35*, 3202–3212.

Raidl, G. R., & Julstrom, B. A. (2003). Edge sets: an effective evolutionary coding of spanning trees. *IEEE Transactions on Evolutionary Computation*, *7*(3), 225–239.

Tay, J. C., & Wibowo, D. (2004). An effective chromosome representation for evolving flexible job shop schedules. *Genetic and Evolutionary Computation*, *3103*, 210–221.

Tung, L. F., Lin, L., & Nagi, R. (1999). Multi-objective scheduling for the hierarchical control of flexible manufacturing systems. *The International Journal of Flexible Manufacturing Systems*, *11*, 379–409.

Xia, W., & Wu, Z. (2005). An effective hybrid optimization approach for multi-objective flexible job-shop scheduling problems. *Computers & Industrial Engineering*, *48*, 409–425.

Xing, L. N., Chen, Y. W., Wang, P., Zhao, Q. S., & Xiong, J. (2010). A knowledge-based ant colony optimization for flexible job shop scheduling problems. *Applied Soft Computing*, *10*, 888–896.

Yuan, Y., & Xu, H. (2013). An integrated search heuristic for large-scale flexible job shop scheduling problems. *Computers & Operations Research*, *40*, 2864–2877.

Zhang, G., Gao, L., & Shi, Y. (2011). An effective genetic algorithm for the flexible job-shop scheduling problem. *Expert Systems with Applications*, *38*, 3563–3573.

Zhang, Y. X., Li, L., Wang, H., Zhao, Y. Y., Guo, X., & Meng, C. H. (2008). Approach to the distributed job shop scheduling based on multi-agent. In *Proceedings of the IEEE international conference on automation and logistics* (pp. 2031–2034).

Ziaee, M. (2014). A heuristic algorithm for the distributed and flexible job-shop scheduling problem. *J Supercomput*, *67*, 69–83.