A memetic algorithm for minimizing total weighted tardiness on parallel batch machines with incompatible job families and dynamic job arrival

Tsung-Che Chiang, Hsueh-Chien Cheng, Li-Chen Fu

PII: S0305-0548(10)00071-7
DOI: doi:10.1016/j.cor.2010.03.017
Reference: CAOR 2559

To appear in: *Computers & Operations Research*

Cite this article as: Tsung-Che Chiang, Hsueh-Chien Cheng and Li-Chen Fu, A memetic algorithm for minimizing total weighted tardiness on parallel batch machines with incompatible job families and dynamic job arrival, *Computers & Operations Research*, doi:10.1016/j.cor.2010.03.017

A memetic algorithm for minimizing total weighted tardiness on parallel batch machines with incompatible job families and dynamic job arrival

Tsung-Che Chiang[a], Hsueh-Chien Cheng[b], Li-Chen Fu[b,c*]

tcchiang@ieee.org, r96922066@ntu.edu.tw, lichen@ntu.edu.tw

[a]*Department of Computer Science and Information Engineering, National Taiwan Normal University, Taiwan, R.O.C.*

[b]*Department of Computer Science and Information Engineering, National Taiwan University, Taiwan, R.O.C.*

[c]*Department of Electrical Engineering, National Taiwan University, Taiwan, R.O.C.*

Abstract

This paper addresses a scheduling problem motivated by scheduling of diffusion operations in the wafer fabrication facility. In the target problem, jobs arrive at the batch machines at different time instants, and only jobs belonging to the same family can be processed together. Parallel batch machine scheduling typically consists of three types of decisions – batch forming, machine assignment, and batch sequencing. We propose a memetic algorithm with a new genome encoding scheme to search for the optimal or near-optimal batch formation and batch sequence simultaneously. Machine assignment is resolved in the proposed decoding scheme. Crossover and mutation operators suitable for the proposed encoding scheme are also devised. Through the experiment with 4860 problem instances of various characteristics including the number of machines, the number of jobs, and so on, the proposed algorithm demonstrates its advantages over a recently proposed benchmark algorithm in terms of both solution quality and computational efficiency.

Keywords: Scheduling; Batch processing machine; Memetic algorithm; Total weighted tardiness

* Corresponding author.

1. Introduction

Today microprocessors, memory chips, and other semiconductor devices are a part of our daily life, appearing from personal computers to cellular phones. Semiconductor manufacturers need to utilize their resources effectively to face the huge demand and severe competition in the market place. There is an increasing pressure on semiconductor manufacturers to improve their due date delivery performance, and effective scheduling is one of the execution-level potential solutions [1][2][3]. Wafer fabrication is the most technologically complex and capital-intensive among the four stages of semiconductor manufacturing. The wafer fab is a large-scale production system where thousands of jobs are processed by tens to hundreds of machines. It is not uncommon for one-third of fab operations to be processed on batch processing machines [1]. Batch machines are distinguished from serial ones by the multi-job capacity. In most cases, once a batch operation starts, it is non-preemptable during the long period of processing. Diffusion is a typical batch operation in the wafer fab. For the chemical nature of this operation, jobs with different recipes cannot be processed simultaneously. Jobs requiring the same recipe are regarded as a job family and have the same processing time. Since jobs of different job families cannot be batched together, scheduling of diffusion operations is a batch scheduling problem with incompatible job families. (Scheduling of burn-in operations in the wafer probe center is an example of batch scheduling problems with compatible job families.)

In this paper, we address the problem of scheduling identical parallel batch machines with incompatible job families. Dynamic job arrival is also considered, which means that jobs are not always ready at the beginning of the scheduling horizon. The jobs may arrive at the machines at different time instants. This reflects the practical condition in most multi-stage production systems. Total weighted tardiness

(TWT) is adopted to evaluate the due date delivery performance. The target problem is usually denoted by $P \mid r_{ij}, d_{ij}$, batch, incompatible family $\mid \sum w_{ij}T_{ij}$. The rest of this paper is organized as follows. Problem descriptions are given in Section 2. In Section 3, we review existing works related to the problem. Design of the proposed approach is presented in Section 4. Experiments and results are summarized in Section 5. Conclusions and future work are provided in Section 6.

2. Problem description

In the focused problem, there are m identical batch machines and n jobs. These jobs are classified into f families. Each family j contains n_j jobs, i.e., $\sum_{j=1\ldots f} n_j = n$. For each job ij of family j, it is associated with a due date d_{ij}, a weight w_{ij}, and an arrival time r_{ij}. Processing time of job ij is denoted by p_j, which depends only on its family. The maximum batch size is denoted by B, and the start time of processing of job ij is denoted by s_{ij}. A feasible solution must satisfy the following constraints:

(1) Each job ij is assigned to exactly one machine m_{ij}, $1 \le m_{ij} \le m$.

(2) Each machine can process at most B jobs at a time.

(3) Only the jobs belonging to the same family can be processed together.

(4) Processing of a job ij can start only after it arrives. In other words, given a batch B_{kj} of jobs of family j, its processing cannot start earlier than $\max\{r_{ij} \mid ij \in B_{kj}\}$.

(5) Once a machine starts processing of a batch B_{kj} of jobs of family j at time t, no job can be added to or removed from B_{kj} until the processing is completed at $t + p_j$.

Our goal is to find among the feasible solutions the one with the minimum TWT, where TWT is defined as

$$TWT = \sum_{j=1,2,\ldots f} \sum_{i=1,2,\ldots n_{fj}} w_{ij} \cdot \max(0, s_{ij} + p_j - d_{ij}) . \quad (1)$$

In practice, scheduling of parallel batch machines is usually achieved by making three types of decisions – batch forming, machine assignment, and batch sequencing. Every time when a batch machine is available to process the next batch operation, there may be more than one possible combination of jobs to form the next batch. On one hand, we intend to form a batch with more jobs inside so that the manufacturing capacity is not wasted. On the other hand, waiting for the incoming jobs to form a larger batch will delay the jobs that already arrived. Hence, we need to make a compromise between increasing utilization of batch machines and decreasing delay time of waiting jobs. Machine assignment, to find m_{ij} for job ij, aims at distributing jobs/batches suitably to machines so that workload among machines is balanced. Finally, on each machine the formed batches must be processed in a proper sequence in order to finish jobs within their due dates and to optimize the due date-related performance index (TWT in this paper).

<< Insert Fig. 1 about here >>

We use Fig. 1 as an example to illustrate scheduling of parallel batch machines with incompatible job families and dynamic job arrival. In Fig. 1 there are two machines and two job families, each containing six jobs. The maximum batch size is two. Among jobs of family 1, we form three full batches. Machine M1 is assigned to process all of them. These three batches are sequenced by their earliest startable time. We can observe that batching job 11 and 12 together postpones processing of job 11 until the arrival of job 12. Jobs of family 2 are put into four batches, and machine M2 is assigned to process all of them. Job 23 is assumed to have a tight due date so that it is processed alone right after its arrival. Job 26 is another one with tight due date, and job 25 has a loose due date. Hence, machine M2 keeps idle until the arrival of job 26 and processes the batch {24, 26} immediately. Job 25 is processed at last.

3. Literature review

Batch machine scheduling problems have been studied extensively in recent years [4][5]. At the beginning, researchers started from the simplest model – scheduling for a single batch machine. Based on greedy heuristics, Uzsoy [6] developed three efficient optimal algorithms to minimize makespan, maximum lateness, and total weighted completion time, respectively. To minimize total tardiness, Mehta and Uzsoy [7] presented a dynamic programming (DP) algorithm, which has polynomial time complexity when the number of job families and the batch machine capacity are fixed. To deal with large-scale problems, they proposed the BATC rule, extending from a well-known heuristic, apparent-tardiness-cost (ATC) rule [8]. In Azizoglu and Webster [9], a branch and bound method was applied to minimize total weighted completion time. Perez *et al*. [10] focused on minimizing TWT. They decomposed the batch scheduling problem into batch forming and batch sequencing subproblems, and then solved them by different combinations of dispatching rules, a DP algorithm, and a decomposition heuristic. Their experimental results showed that the combination of ATC and the decomposition heuristic provides good solution quality within reasonable computation time. To develop effective rules for minimizing total completion time and total tardiness automatically, a genetic programming-based learning approach was devised by Geiger and Uzsoy [11]. Their system could discover rules that are structurally similar to and competitive with the BATC rule proposed by Mehta and Uzsoy [7]. Researches were also conducted on scheduling of jobs with different sizes on a single batch machine. To minimize makespan, total completion time, and total weighted completion time, Koh *et al*. [12] proposed several rule-based heuristics like largest-job-first-fit for batch forming and weighted-shortest-processing-time (WSPT) for batch sequencing. They also proposed a random-key -based genetic algorithm (GA), which forms batches by considering jobs in increasing

order of random keys and sequences batches by rules such as WSPT. Kashan and Karimi [13] addressed the same problem as that in Koh *et al*. [12] but focused on minimizing total weighted completion time. They developed an ant colony optimization (ACO) algorithm for batch forming and batch sequencing. A pair-wise swapping heuristic was also used to adjust the batch formation to improve the solution quality.

Parallel machines are usually required in real-world production systems in order to prevent the system from being blocked by the unavailability (e.g. breakdown) of a single machine. Uzsoy [6] presented heuristics to minimize makespan and maximum lateness for identical parallel batch machines. These two heuristics were based on the longest-processing-time-first (LPT) and earliest-due-date-first (EDD) rules. Tight bounds on the solution quality of these two heuristics were also given. Balasubramanian *et al*. minimized TWT for identical parallel batch machines in [14]. They decomposed the original problem into three subproblems – batch forming, machine assignment, and batch sequencing. The GA was used to solve the machine assignment subproblem, and dispatching rules such as BATC were applied to solve the other two subproblems. They tested two versions of GA. The first version does machine assignment at first by taking the job as the unit of assignment and then applies dispatching rules to do batch forming and sequencing. The second version forms batches by rules at first, then does machine assignment by the GA, and finally does batch sequencing by rules. The experimental results showed that the second version provides better performance, which reveals that the early incorporation of domain knowledge (dispatching rules) can yield better solution quality and save computation time. The problem addressed in [14] was tackled by Raghavan and Venkataramana [15]. In their approach batches are formed by ATC-like rules, and machine assignment and batch sequencing are determined by the ACO algorithm. Koh

et al. [16] adopted similar approaches as they proposed in [12] to do parallel batch machine scheduling with unequal job sizes. Mathirajan and Sivakumar [17] aimed at minimizing TWT for parallel batch machines with unequal machine capacity and unequal job sizes. Machine assignment, batch forming, and batch sequencing were resolved by rules like large-capacity-machine-first, first-come-first-serve, and weighted-earliest-due-date-first.

To make the batch machine scheduling problem closer to the real-world situation, the dynamic job arrival should be considered. Glassey and Weng [18] and Fowler *et al*. [19] proposed rules for job dispatching on batch machines by considering the arrival of future lots. The basic idea is to evaluate the concerned performance measures (for example, total waiting time of jobs) at different time instants of starting the batch operations. They showed that the use of this information is able to reduce cycle times in the single batch machine environment. To deal with the dynamic arrival of jobs, Uzsoy [6] developed a release date update (RDU) algorithm to minimize maximum lateness for a single batch machine. The concept of RDU is to (temporarily) delay the release dates of jobs with later due dates so that the batch machine can wait for the arrival of jobs with earlier due dates. Tangudu and Kurz [20] proposed a branch and bound algorithm to minimize TWT on a single batch machine. They utilized several dominance properties to speed up their algorithm and showed that their algorithm could solve the problems with up to 32 jobs. The idea in the work by Gupta and Sivakumar [21] was similar to that in [18]. The difference is that in [21] the different starting time instants are evaluated with respect to earliness and tardiness by the compromise programming method. Kurz and Mason [22] presented a batch improvement algorithm (BIA) to minimize TWT on a single batch machine. The BIA improved the schedule iteratively by moving jobs from later batches to earlier batches without delaying the starting time of the earlier batches.

Although batch machine scheduling has been widely studied, there are not many researches on parallel batch machine scheduling with dynamic job arrival. Malve and Uzsoy [23] proposed a GA to minimize maximum lateness on identical parallel batch machines. Their GA adopts the random-key encoding scheme to search for the optimal job priority values and decodes the chromosome through the list scheduling algorithm. The RDU algorithm in [6] serves as a local optimizer of the chromosomes in each generation of the GA. Mönch *et al*. [24] extended the approach in [14] by proposing three modified BATC rules to consider the dynamic arrival of jobs. Among their rules, the rule which adds the machine idle time for waiting future jobs into the slack term in the BATC [7] index has the best TWT performance. The approach in [24] was extended further to address the problem with more than one objective by Reichelt and Mönch [25].

Besides scheduling of jobs with incompatible families, there are also researches on scheduling of jobs with compatible families. These two types of problems have very different characteristics, and readers who are also interested in that topic may refer to [26]–[28].

4. Proposed memetic algorithm

4.1 Overview

GAs, which mimic the evolutionary process in the nature, have shown many successful applications to production scheduling problems [29]–[31]. Like creatures in the nature evolve to adapt to the environment, solutions in the GA evolve to adapt to the target problem. In the GA solutions are usually encoded into a compact form to facilitate the use of reproduction operators including crossover and mutation. The encoded solution is usually referred to as an individual, and a group of individuals is referred to as a population. Starting from an initial population, some individuals are

selected as parents and then produce new individuals through crossover and mutation. Among the original and new individuals, some survive and the others die. The surviving individuals form a new population, and we call the transition from one population to another a generation. Individuals whose corresponding solutions have better objective values usually have higher probability to be selected as parents and survivors. It is expected that optimal or near-optimal individuals/solutions will be obtained by evolving the population after a number of generations.

Memetic algorithms (MAs) inherit the population-based evolutionary process from GAs and integrate the individual-based learning process into its framework. Since the individual-based learning is usually achieved by local search-based algorithms like hill climbing, MAs are also known as genetic local search algorithms. By combining the extensive search of GAs and the intensive search of local search algorithms, MAs have shown good performance in various kinds of scheduling problems, including single machine scheduling [32], flow shop scheduling [33][36], job shop scheduling [34][35], multiobjective scheduling [36][37], and batch machine scheduling [23][38].

In this paper, we develop a MA to solve the parallel batch machine scheduling problem. We propose a chromosome encoding scheme, whose feature is to consider batch formation and batch sequence simultaneously. In addition, crossover and mutation operators suitable for the new encoding scheme are devised. In Section 2, we mentioned that solving the target problem is usually achieved by making three types of decisions – batch forming, machine assignment, and batch sequencing. The proposed MA will search for good batch formation and batch sequence, and machine assignment is done during decoding the chromosomes into corresponding schedules. An overview of the proposed MA is given before we go into the details.

Step 0. Design the chromosome encoding and decoding schemes.

Step 1. Generate the initial population $Pop(1)$ with N_{POP} individuals. Evaluate them by the decoding scheme and the fitness function. Record the lowest TWT obtained by individuals in $Pop(1)$ in TWT^*.

Step2. Repeat mating selection, crossover, and mutation for $N_{POP}/2$ times.

Step 2.1. Pick up two parents through 2-tournament mating selection.

Step 2.2. Do crossover on the selected parents to produce two offspring. Apply the proposed batch formation crossover with probability $P_{F,GA}$ and the proposed batch sequence crossover with probability $(1 - P_{F,GA})$.

Step 2.3. Do mutation on the produced offspring with probability P_m. If the mutation takes place, apply the proposed batch formation mutation with probability $P_{F,GA}$ and the proposed batch sequence mutation with probability $(1 - P_{F,GA})$.

Step 2.4. Evaluate the two offspring. The best two individuals among the two parents and two offspring will replace the parents.

Step 3. If the best individual in the current population has lower TWT than TWT^*, apply local search to the best N_{LS} individuals. Then, update TWT^* with the lowest TWT obtained by the current population.

Step 4. If any of the stopping criteria is reached, stop. Otherwise, go to Step 2.

4.2 Chromosome encoding

The three types of decisions in the target problem collaborate to result in a huge solution space. Although metaheuristics including MAs are known as promising optimization approaches, it is still a great challenge to search in such a large solution space effectively. To reduce the search space, Mönch *et al*. [24] proposed to encode

only machine assignment in the chromosome, as shown in Fig. 2. They first applied their proposed BATC-based heuristics to form batches, and then used their GA to search for good machine assignment of these batches. Assuming that their proposed BATC-based heuristic forms seven batches, individual 1 in Fig. 2 represents the solution in which machine 1 is assigned to process the first three batches and machine 2 is assigned to process the last four batches. Finally, batches on each machine were sequenced by the BATC-based heuristics again. Reduction of search space is useful for the fast convergence of the GA, but meanwhile good solutions could be excluded from the search space and unreachable by the GA. In our previous work [38], we did improvement by encoding the batch sequence in the chromosome to enlarge the search space, as shown in Fig. 3. For example, individual 1 in Fig. 3 represents the solution in which we schedule the batch B1 first, then B4, B2… and finally B7. When a batch is to be scheduled, the machine with the earliest available time is responsible for processing it. Under the framework of the MA, our previously proposed approach achieved about 6% improvement percentage over Mönch *et al.*' approach [24].

<< Insert Fig. 2 and Fig. 3 about here >>

Although batch sequence was encoded in our recently proposed MA to extend the search space, batch formation was still left fixed in [38]. In this paper, we propose to encode batch formation and batch sequence simultaneously in the chromosome. A chromosome is a sequence of batches, each containing at least one job and at most B (maximum batch size) jobs. Given a schedule, we put the batches into the chromosome from left to right in increasing order of the completion time of their previous operation processed on the same machine. Batches whose previous operations have the same completion time are put in increasing order of the index of the machine on which they are processed. If a batch is the first operation on a machine, the completion time of the previous operation is zero. We use Fig. 4 to show how a

schedule is encoded into a chromosome. In the schedule, the completion times of the previous operations of batches {11, 12}, {21, 22}, {23, 24}, {13, 14}, and {15, 16} are 0, 0, 4, 8, and 12, respectively. Accordingly, we encode the schedule into individual 1. The batch {11, 12} is put on the left of {21, 22} since machine M1 has a smaller index than machine M2 does.

<< Insert Fig. 4 about here >>

Different chromosomes represent different batch sequences and/or different batch formations. For example, in Fig. 4 individual 2 has the same batch formation as individual 1 does, but the batch sequences in these two individuals are different. The batch formation and batch sequences in individuals 1 and 3 are both different. By encoding both the batch formation and batch sequence in the chromosome, we can enlarge the search space and thus include more high-quality solutions. On the other hand, to reach these high-quality solutions in such a huge space, we need to carefully design the remaining components of the MA, which will be detailed in the following subsections.

4.3 Chromosome decoding

Given a chromosome, the decoding scheme is responsible for constructing its corresponding schedule in order to calculate the concerned objective function. Since batch formation and batch sequence are already encoded in the chromosome, the remaining decision is machine assignment. In our decoding scheme, machine assignment is done by assigning the batch to the machine with the earliest available time. Ties are broken by assigning the batch to the machine with the smallest index. The decoding scheme will construct the schedule that is encoded in the chromosome. Let us use Fig. 5 as an example to explain how a chromosome is decoded.

<< Insert Fig. 5 about here >>

With the chromosome in Fig. 5, the first batch to be scheduled is {11, 12}. Both machines M1 and M2 are available at time 0, and thus we assign the batch to machine M1. Since both jobs 11 and 12 arrive at time 0, processing of this batch starts at time 0. The next batch is {21, 22}. It is assigned to machine M2 because now the available time of M1 is 4 (p_1 = 4). Waiting until both jobs 21 and 22 arrive, processing of this batch starts at time 5 and ends at time 8 (p_2 = 3). Now the available times of M1 and M2 are 4 and 8, respectively. Thus, the third batch {23, 24} is assigned to M1. Considering the arrival times of jobs inside, the third batch is processed from time 10 to 13. Scheduling of the last two batches follows the similar way, and we decode the chromosome to its corresponding schedule in Fig. 5.

The major advantage of the above decoding scheme is the simplicity. Scheduling the batches in the order of their positions in the chromosome and assigning them to the machine with the earliest available time has the time complexity $O(xm)$, where x and m denote the number of batches and the number of machines, respectively. However, this basic decoding scheme may generate schedules with long idle periods on machines and consequently with poor quality. In the schedule in Fig. 5, for example, machine M2 is idle from time 0 to 5. Since jobs 13 and 14 arrive at time 0, we can schedule them in the idle period of M2 without delaying any of the first three scheduled batches, resulting in a better schedule. Thus, we improve the basic decoding scheme by considering the idle periods on machines when scheduling the batches. In the improved scheme, the set of idle periods on all machines is identified before each batch is scheduled. Among these idle periods, only the periods into which the batch can be scheduled without violating the arrival time and processing time are kept. If there is more than one feasible period, the batch is scheduled into the period with the shortest duration. If there is no such period, the batch is scheduled on the machine with the earliest available time, as what we do in the basic decoding scheme.

Fig. 6 is provided as an example.

<< Insert Fig. 6 about here >>

The first three batches are scheduled as what we did in Fig. 5. When the batch {13, 14} is to be scheduled, there are two idle periods – {4, 10} on M1 and {0, 5} on M2. Both idle periods are feasible. According to our improved decoding scheme, we schedule the batch into the idle period {0, 5} since its duration is shorter. To schedule the last batch {15, 16}, there are also two idle periods – {4, 10} on M1 and {4, 5} on M2. Because job 16 arrives at time 7 and processing of this batch requires 4 units of time, both idle periods are infeasible. Therefore, we schedule the last batch on M2, whose available time 8 is earlier than that of M1, 13. Comparing the schedules in Fig. 5 and Fig. 6, the improved decoding scheme generates a better schedule, where all jobs are completed at earlier or the same time instants.

As we can see, the proposed decoding scheme is able to construct better schedules by changing the batch sequence originally recorded on the chromosome. To pass this result back to the MA, we adjust the batch sequence on the chromosome after applying the decoding scheme. For example, the chromosome in Fig. 6 will be modified to be {11, 12}{13, 14}{23, 24}{21, 22}{15, 16}. The only cost of the improved scheme is that its complexity is $O(x(W+m))$, where W denotes the number of idle periods. Although the value of W can be up to x, it is usually much smaller than x after we do the post-decoding adjustment to the batch sequence on the chromosome.

4.4 Fitness function, mating selection, and environmental selection

In our MA, the fitness of an individual is defined by the reciprocal of TWT obtained by the decoding scheme. Mating selection is achieved through 2-tournament, which selects two individuals randomly from the current population and picks up the one with higher fitness to be a parent. After selecting two parents, crossover will be

applied to them to produce two offspring, and mutation will be applied probabilistically to the offspring. Among the parents and the offspring, two individuals with the highest fitness will replace the parents. This environmental selection mechanism showed better performance than three existing mechanisms in our previous work [35].

4.5 Crossover and mutation

Given two parents, the task of crossover is to generate the offspring through inheriting features (gene structures) from the parents. Since different encoding schemes have different gene structures and constraints, each kind of encoding scheme has its own suitable crossover operators. For example, the machine assignment-based encoding scheme in Fig. 2 has no constraint between gene values. Thus, the simple 2-point crossover is applicable, as shown in Fig. 7. For the batch sequence-based encoding scheme in Fig. 3, in an individual each batch index must appear exactly once. Linear order crossover is one of the crossover operators that are applicable to this encoding scheme. An example in given in Fig. 8.

<< Insert Fig. 7 and 8 about here >>

Although many crossover operators were proposed in the literature, we found that the existing ones are not applicable to the proposed encoding scheme, which encodes batch formation and batch sequence simultaneously. In the following we will propose two crossover operators, one focusing on exchanging the batch formation between parents and the other focusing on exchanging the batch sequence.

<< Insert Fig. 9 about here >>

Let us describe our *batch formation crossover* operator first. Fig. 9 provides a visual explanation. The first step is to choose a job family randomly and extract the sequences of jobs belonging to this family from two parents. In Fig. 9 we assume that

job family 1 is chosen. Second, two cut points are chosen randomly in the job sequence. Third, we leave the parts outside the section enclosed by cut points unchanged and fill the enclosed section with the remaining jobs in the order of the other parent. Finally, the offspring is produced by copying one parent and re-filling the batches belonging to the selected job family according to the new job sequence and the original batch size. After doing the batch formation crossover, the offspring will inherit batch formation from both parents. For example, in Fig. 9 the first offspring inherits batch formation {11, 13} from the first parent and batch formation {15, 16} from the second parent. On the other hand, the sequence of job families of batches is passed directly. For example, in Fig. 9 the sequences of job families of batches are [f_1 f_2 f_1 f_2 f_2 f_1 f_2] in both the first parent and first offspring.

<< Insert Fig. 10 about here >>

Next, we use Fig. 10 to illustrate how the other crossover operator, *batch sequence crossover*, works. First, we extract the sequences of job families of batches from the parents. Second, two cut points are selected randomly. Third, we leave the parts outside the section enclosed by cut points unchanged and fill the enclosed section with the remaining job families in the order of the other parent. To do this, we search in one parent from the beginning for job families that appear in the part before the first cut point in the other parent. For example, in Fig. 10 the part before the first cut point x in the first parent contains one f_1 and one f_2, and we find the first f_1 at position 2 and first f_2 at position 1 in the second parent. Since the part before the first cut point in each parent is unchanged, these job families found in the other parent are ignored during the later exchange of job families. Similarly, we also search in one parent from the end for job families that appear in the part after the second cut point in the other parent. In Fig. 10, for instance, the part after the second cut point y in the first parent contains two f_2, and we find the last two f_2 at position 5 and 6 in the

second parent. In the fourth step, the remaining job families move into the section enclosed by the cut points. Finally, the offspring is produced by copying one parent and re-arranging the order of batches inside the enclosed section in the order of batches in the section of the other parent. For example, in Fig. 10 the first offspring copies the first parent and re-arranges the batches {24, 26}, {12, 15}, and {14, 16} in the order of [f_1 f_2 f_1]. Thus, the new order of these three batches becomes {12, 15}, {24, 26}, and then {14, 16}. After doing the batch sequence crossover, the offspring will inherit batch sequences from both parents and keep the batch formation in one parent.

<< Insert Fig. 11 and 12 about here >>

Comparing with crossover, which combines features of parents into the offspring, mutation is simpler since it changes gene values based on a single individual. We propose two mutation operators for batch formation and batch sequence, respectively. In the *batch formation mutation* operator, we first choose a job family randomly. Then, we randomly select two batches belonging to this family. Finally, we select one job in each selected batches randomly and exchange them. An example is given in Fig. 11. In the *batch sequence mutation* operator, two batches (not necessarily belonging to the same family) are selected randomly and then are exchanged. An example is shown in Fig. 12.

4.6 Local search procedure

With the nature of population-based search, GAs usually have better exploration ability than local-search-based approaches do. On the other hand, in some applications GAs were also reported to lack sufficient exploitation ability to find the best solution after it locates a promising region in the search space. Hence, combining the GA and the local search into the framework of MA becomes a popular approach.

In MA, the individuals join together in the evolutionary process simulated by the GA to improve themselves through selection, crossover, and so on; meanwhile, a single individual can improve himself/herself in the learning process represented by the local search. Technically speaking, the GA leads the population to one or multiple promising regions in the search space, and the local search leads each individual to be the best position in each of these regions.

As mentioned in Section 4.1, we decide whether to start the local search procedure after a new population is formed by mating selection, crossover, and mutation. In order to save the computation time, the local search procedure starts only when the best individual in the newly-formed population is better than the best individual in the previous population. In addition, only the best N_{LS} individuals in the population will do the local search. Taking each of these individuals as the starting solution, the local search procedure generates neighboring solutions and moves the current solution to the neighboring one when the neighboring one is accepted. Neighboring solutions are generated by applying the batch formation mutation operator and the batch sequence mutation operator with probability $P_{F,LS}$ and $(1 - P_{F,LS})$, respectively. Since accepting only the neighboring solution with better quality than the current one makes the local search trap in the local optimum easily, the neighboring solution is accepted if the objective value (TWT) of its derived schedule is not worse than that of the best neighbor by D_P%. The best neighbor is recorded during the local search. Once the neighboring solution is not acceptable, the search will go back to the best neighbor and then continue. The local search stops after $N_{EVAL,LS}$ neighbors are examined, and then the best neighbor will replace the individual initiating the local search in the population. The pseudo code of our local search procedure is given in Table 1.

<< Insert Table 1 about here >>

4.7 Generation of initial population and stopping criterion

A good initial population can help MA to find the optimal or near-optimal solutions efficiently, especially when the search space is large. In our approach, we use a heuristic procedure proposed by Mönch *et al*. [24] to generate the initial population.

In this procedure, each time when a machine becomes available at time t, it first collects the jobs ij that already arrived or will arrive within a time period Δt (i.e. $r_{ij} \le t + \Delta t$). Next, these jobs will be assigned a priority index by a modified version of the ATC dispatching rule and are ranked in decreasing order of the ATC indices. ($\overline{p}$ denotes the average processing time of the remaining unscheduled jobs.)

$$I_{ij,ATC}(t) = \left(\frac{w_{ij}}{p_j}\right)\exp\left(-\frac{(d_{ij} - p_j + r_{ij} - t)^+}{k\overline{p}}\right) \tag{2}$$

In order to save the computational effort, only the jobs with ranks smaller than a predefined threshold *thres* are considered in later steps. All possible combinations (batches) of these jobs are then formed and are assigned priority indices by Mönch *et al*.' proposed BATC-II rule. For each batch B_{kj}, the BATC-II index is calculated by:

$$I_{BATC-II}(t) = \sum_{ij\in B_{kj}}\left(\frac{w_{ij}}{p_j}\right)\exp\left(-\frac{(d_{ij} - p_j - t + (r_k - t)^+)^+}{k\overline{p}}\right)\min\left(\frac{|B_{kj}|}{B},1\right) \tag{3}$$

Finally, the best batch (with the largest BATC-II index) is scheduled onto the machine. Repeating the above steps, a complete schedule $s^{\text{BATC-II}}$ will be constructed. Since the performance of the ATC-based rules is sensitive to the parameter k, we follow the method proposed in [24] to determine the value of k. We test ten different values of k from 0.5 to 5 in increments of 0.5 and pick the schedule that gives the minimum TWT. This best schedule is encoded to be the first individual in our initial

population. The rest of the population is filled by duplicating the first individual and perturbing its batch formation and batch sequence using the proposed batch formation mutation and batch sequence mutation operators.

Our algorithm will stop if the best solution is not updated after $N_{NONIMPRV}$ continuous generations or when the predefined maximum number of generations (N_{GEN}) is reached.

5. Experiments and results

5.1 Benchmark approach and its parameter setting

In the experiment, we compared our approach with the approach proposed by Mönch *et al.* [24] since they have conducted a series of studies on batch machine scheduling [10][14] [24][25], and the approach in [24] was developed to solve exactly the same problem as ours. In [24], they proposed two frameworks of genetic algorithm, GA-1 and GA-2. Their difference is in the timing of batch formation. GA-1 forms batches by the dispatching rule and then assigns batches to machines, whereas GA-2 assigns jobs to machines and then forms batches by the dispatching rule. In their experiments seven variants of GA-1 and GA-2 were tested, and the variant BATC-II_GA-1_BATC-II provided the second best solution quality. The variant GA-2_DTH slightly outperformed BATC-II_GA-1_BATC-II by 2% in terms of solution quality (91.82% vs. 89.89% in Table 5 in [24]), but its required computation time was about 70 times (4038.15s vs. 58.37s in Table 9 in [24]). Regarding both solution quality and computational efficiency, we decide to take the variant BATC-II_GA-1_BATC-II as the benchmark approach in our experiment.

The differences between Mönch *et al*.' approach and our approach originate from the different encoding schemes. Mönch *et al*.' GA encoded machine assignment, whereas our MA encodes batch formation and batch sequence. In their approach,

batch formation is obtained from the schedule $s^{\text{BATC-II}}$ generated by the heuristic method as described in Section 4.7 and is fixed afterward. Their GA seeks for the optimal machine assignment of these batches. Batch sequencing is done during decoding of chromosomes, where batches are distributed to machines according to the assignment recorded on the chromosomes, and the processing sequence on each machine is determined by their proposed BATC-II dispatching rule. In the following experiment on performance comparison, the parameter setting of Mönch *et al.*' approach was identical to what they reported in Table 2 and Table 8 in [24]. We use the same values for parameters Δt and *thres* (Δt = 8 and *thres* = 10) to generate the initial population in our MA.

5.2 Test problem instances and test environment

<< Insert Table 2 about here >>

We generated the test problem instances in the same way as in [24]. Parameters and their values for instance generation are listed in Table 2. There are 3·3·3·2·3·3 = 486 categories of instances with different combinations of parameter values. Ten instances were generated for each category. Thus, totally 486·10 = 4860 problem instances were used in our experiments. The proposed MA and the benchmark approach were implemented in C++ on Microsoft Visual Studio 2005. The testing environment was a personal computer running windows XP with Intel Core 2 Duo 3.0 GHz CPU and 2 GB RAM.

5.3 Parameter tuning of the proposed MA

A MA typically has the following parameters: the population size (N_{POP}), the number of generations (N_{GEN}), the mutation rate (P_{m}), the number of individuals applying local search (N_{LS}), and the number of evaluations in local search ($N_{\text{EVAL, LS}}$).

In our MA, we introduce two additional parameters $P_{F,GA}$ and $P_{F,LS}$ to choose between two types of crossover and mutation operators in the GA and local search parts, respectively. Last, the maximum non-improving generations $N_{NONIMPRV}$ is related to the stopping criterion, and the maximum acceptable deviation percentage D_P is related to the acceptance criterion in the local search procedure. From the pilot runs, we set the values of P_m, $N_{NONIMPRV}$, and D_P by 0.1, 50, and 5, respectively. As for the other parameters, an extensive experiment was conducted to determine their suitable values.

In [24], Mönch *et al*. tuned the values of the parameters in their approach using problem instances with three job families (f = 3). In order to have a fair performance comparison, we also did parameter tuning of the proposed MA using problem instances with three job families. We randomly picked up one instance from each of the 162 (3·3·2·3·3) problem categories. Various versions of our MA with different combinations of parameter values were applied to solve each of the selected 162 problem instances. Each version of our MA solved each instance for five times. For a problem instance j, let T_j(MA, i, k) denote the TWT value obtained by version i of our MA in run k and T_j(BATC-II) denote the TWT value obtained by the BATC-II-based heuristic [24] (mentioned in Section 4.7). Similar to the analysis in [24], we evaluated version i of our MA by the average-case improvement percentage (IP) over the BATC-II-based heuristic:

$$IP(i) = 1 - (\sum_{j=1}^{162} \sum_{k=1}^{5} (T_j(MA, i, k) / T_j(BATC - II))) / (162 \cdot 5). \quad (4)$$

To avoid testing too many versions of MA, we tested the parameter values in a two-stage manner. In the first stage, we considered four typical MA parameters N_{POP}, N_{GEN}, N_{LS}, and $N_{EVAL,\,LS}$ and tested 21 versions of MA. The values of parameters $P_{F,GA}$ and $P_{F,LS}$ were both set by 0.5. The IPs of these 21 versions of MA are summarized in Table 3. Average computation time is also included.

<< Insert Table 3 about here >>

As we can see, the solution quality gets better when the population size gets larger. The only exception is the case where local search was not applied (200×500 vs. 400×250). It is consistent with our experience that a GA with too large population may wander over the search space but not able to find good solutions due to insufficient exploitation ability. The solution quality also becomes better when more computational effort is given to the local search procedure. With a fixed number of evaluations in local search, better solution quality is obtained when more individuals apply the local search procedure. Although better solution quality usually accompanies longer computation time, the incorporation of local search has the potential to improve solution quality and speed up convergence simultaneously. For example, adding the 5×100 local search procedure into the 100×1000 GA improves the solution quality by 1.5% and slightly reduces the average computation time by 0.57 second. This observation shows the advantage of MAs over pure GAs. To keep the chance of finding a version of MA that runs fast and provides high solution quality, we picked up two versions of MA, the one with the shortest average computation time and the one with the highest solution quality (marked in bold in Table 3), to the second stage of parameter tuning.

Parameters $P_{F,GA}$ and $P_{F,LS}$ were considered in the second stage. They determine the probability of using batch-formation crossover and mutation operators in the GA and local search part, respectively. Based on each of two versions of MA picked up in the first stage, we tested five values for both parameters, resulting in 25 versions. The experimental results for two groups of 25 versions are provided in Table 4 and Table 5, respectively.

<< Insert Table 4 and 5 about here >>

In Table 4, the first observation is that all versions of MA using both types of operators significantly outperform the two versions using only one type of the operators ($P_{F,GA} = P_{F,LS} = 0$ and $P_{F,GA} = P_{F,LS} = 1$). The extra improvement percentage is up to 10% (24.54% – 14.12%). This result is encouraging since it confirms that using MA to simultaneously search for the optimal batch formation and batch sequence is a good direction to enhance the solution quality. The version using only batch formation crossover/mutation operators has better performance than the version using only batch sequence crossover/mutation operators. This can be explained by the fact that batch sequence operators are not able to modify the batch formation. Thus, a wrong batch formation could delay several jobs for a long time until the arrival of the last job to start processing of the batch. On the contrary, batch formation operators are able to achieve "partial" functionality of batch sequence operators by modifying batch formation to re-arrange the processing sequence of jobs belonging to the same family. All nine versions of MA with $0 < P_{F,GA} < 1$ and $0 < P_{F,LS} < 1$ have close solution quality and computational requirement. It means that the performance of our MA is robust to the variation of values of $P_{F,GA}$ and $P_{F,LS}$ as long as both types of operators are adopted. Practitioners are relieved from comprehensive tests to determine values of these two parameters. The results in Table 5 are consistent with what we conclude from Table 4.

<< Insert Table 6 about here >>

Among all 50 (5×5×2) versions of MA in Table 4 and 5, we selected the version using $P_{F,GA} = 0.75$ and $P_{F,LS} = 0.25$ in Table 4 due to its high solution quality and short computation time. (It provides the best solution quality within four seconds.) We did the first-stage experiment again and summarize the results in Table 6. Comparing the results in Table 6 with those in Table 3, each version of MA provides slightly better solution quality and consumes slightly longer computation time than its counterpart.

Since there is only little difference between the results in Table 3 and Table 6, the second-stage experiment was not done again and the parameter tuning process ended. By making a trade-off between the solution quality and the computational requirement, we chose the fastest version to compare with the benchmark approach. In summary, the values of parameters of our MA in the experiment on performance comparison are: N_{POP} = 100, N_{GEN} = 1000, N_{LS} = 1, $N_{EVAL,\,LS}$ = 500, $P_{F,GA}$ =0.75, $P_{F,LS}$ = 0.25, P_m = 0.1, $N_{NONIMPRV}$ = 50, and D_P = 5.

5.4 Performance comparison

In the experiment on performance comparison, our MA and the benchmark approach [24] solved each of the 4860 problem instances for ten times. Besides the IP over all 4860 instances, we group instances with identical value of a certain problem parameter (such as the number of machines) and calculate IPs for various groups. To evaluate the robustness of both approaches, we calculate the IPs in the best, average, and worst cases, which are defined as follows:

$$IP^{best}(approach) = 1 - \frac{1}{|X|}\sum_{j=1}^{|X|}(\min_{k=1...10}\{T_j(approach,k)\}/T_j(BATC-II)) \quad (5)$$

$$IP^{avg}(approach) = 1 - \frac{1}{|X|}\sum_{j=1}^{|X|}(\sum_{k=1}^{10}\frac{T_j(approach,k)}{10}/T_j(BATC-II)) \quad (6)$$

$$IP^{worst}(approach) = 1 - \frac{1}{|X|}\sum_{j=1}^{|X|}(\max_{k=1...10}\{T_j(approach,k)\}/T_j(BATC-II)) \quad (7)$$

where *approach* is either our MA or the benchmark approach, X denotes the group of instances being considered, j is the index of instance in X, and k is the index of the test run. The IPs and average computation time are summarized in Table 7. Take the third column of the third row as an example, the IP values of our MA in the best, average, and worst cases are 29.3%, 26.4%, and 23.3%, respectively, when we consider the

1620 3-family problem instances. The average computation time is 3.9 seconds.

Although the benchmark approach was re-implemented by us, we have carefully checked the correctness by carrying out unit tests, tracing of the program flows, and verifying the calculations through log files. Besides, the experimental results of the original and re-implemented versions are quite close. (Recall that the problem instances in our experiment were generated in the same way as in [24].) The average IP of the re-implemented version is 10.80% for 1620 3-family problem instances, and that of the original version was 11.12% (Table 8 in [24]). The close performance also helps to verify the correctness of our re-implementation.

<< Insert Table 7 about here >>

In Table 7, we first observe that performance variations of both tested approaches with respect to problem characteristics are similar. As the number of job families (f), the number of machines (m), and the maximum batch size (B) decreases, or as the number of jobs (n) increases, the number of batches assigned to each machine becomes larger generally. In this condition, there are more decisions to make and consequently a larger space for performance improvement. When solving problem instances whose maximum batch size is eight, for example, the IPs by our MA are between 14.6% and 19.6%. By cutting the batch size to the half, the IPs by our MA become much higher, with the range between 23.1% and 29.7%. As the value of the arrival time factor (α) increases, the contention for machines gets lighter; as the value of due date factor (β) increases, the allowance time of jobs gets larger. In both of the above conditions, there is more possibility to complete the jobs within their due dates and consequently a larger space for performance improvement. For example, the IPs (24.8% ~ 31.7%) by our MA for problem instances with the arrival time factor equal to 0.75 are more than twice the IPs (10.8% ~ 15.4%) for problem instances with the arrival time factor equal to 0.25.

Comparing with IPs of the benchmark approach, IPs of our MA are generally higher by at least 10%. When solving the problem instances with a larger space for performance improvement, our MA shows greater advantage. For example, the IPs (14.6% ~ 19.6%) by our MA for problem instances with the batch size equal to eight are higher than those (5.5% ~ 8.6%) of the benchmark approach by around 10%. The difference (23.1% ~ 29.7% vs. 9.6% ~ 15.1%) increases to around 14% when the two approaches solved the problem instances with the batch size equal to four. The largest difference is observed when problem instances with loose due dates (β = 0.75) were solved. The IPs (24.2% ~ 31.5%) of our MA are higher than those (8.2% ~ 14.3%) of the benchmark approach by at least 16%. Averaging over all 4860 instances, the IP of our MA is higher than that of the benchmark approach by 12.9%, 11.9%, and 11.3% in the best, average, and worst cases, respectively. The difference between the best-case and worst-case IPs of our MA is 5.8% (24.7% – 18.9%), only slightly higher than that (11.8% – 7.6% = 4.2%) of the benchmark approach by 1.6%. It shows that our MA significantly outperforms the benchmark approach and keeps the robustness.

In Table 7 the computation time required by the benchmark approach is much shorter than that reported in [24]. The difference arises mainly from two factors, namely, different ways of implementation and different computing platforms. The way of our implementation is specific to the algorithm flow of the benchmark approach, whereas the original implementation was based on GAlib [39], which is a general library of genetic algorithms and may include some actions that are not necessary for running the benchmark approach. In addition, the CPU in our computing platform runs about twice faster than that of theirs (3.0 GHz vs. 1.7 GHz) and the size of memory of our platform is also larger (2 GB RAM vs. 256 MB RAM). Considering the computational efficiency, the computation time required by our MA is shorter than that required by the benchmark approach. One possible source for the

speedup is the difference between the decoding schemes. In the decoding scheme of the benchmark approach, the BATC-II dispatching rule must be applied iteratively to sequence batches on each machine. In our MA, the sequence of batches is already determined by the permutation of batches on the chromosome. Another source for the speedup could be the use of local search in our approach. It helps our MA to converge faster than the benchmark approach, which is a GA-alone algorithm.

6. Conclusions

In this paper, we addressed the identical parallel batch machine scheduling problem with incompatible job families and dynamic job arrival. This problem is motivated by scheduling of the diffusion operation in the wafer fabs and has a high practical value. We proposed a scheduling approach based on the framework of the MA. It uses a new genome encoding scheme that considers batch formation and batch sequence simultaneously. We also developed crossover and mutation operators suitable for the new encoding scheme. Based on the results of a comprehensive experiment, the performance improvement percentage of our approach is higher than that of the benchmark approach by at least 10% on average. Besides, the average computation time of our approach is shorter than that of the benchmark approach by more than 50%. Our approach also shows robust performance with respect to values of parameters.

Our research will continue following three directions: (1) Although the current approach encodes batch formation in the chromosome, the number of batches of each job family is still fixed. In the next version, we want to relax this constraint by allowing batches to be combined and split during the search process of the MA. (2) The problem with non-identical parallel machines will be studied. In the extended problem, machine assignment must be done more intelligently. We may solve it by

adding heuristics for machine assignment in our current MA, or we may also enhance our current genome encoding scheme by considering machine assignment. (3) An advanced local search procedure will be used in our approach. Currently, the neighborhood function uses no domain knowledge. In the future, we will utilize the information in the derived schedule (e.g. arrival times and due dates of jobs) to generate the neighboring solutions (e.g. move tardy jobs to earlier batches [22]).

Acknowledgment

This research was supported by National Taiwan Normal University under research grant No. 97091022 and by the National Science Council of Republic of China under research grant No. 97-3114-E-002-002.

References

[1] Gupta JND, Ruiz R, Fowler JW, Mason SJ. Operational planning and control of semiconductor wafer production. Production Planning & Control 2006;17(7): 639–47.

[2] Sourirajan K, Uzsoy R, Hybrid decomposition heuristics for solving larg-scale scheduling problems in semiconductor wafer fabrication. Journal of Scheduling 2007;10:41–65.

[3] Pfund ME, Balasubramanian H, Fowler JW, Mason SJ, Rose O. A multi-criteria approach for scheduling semiconductor wafer fabrication facilities. Journal of Scheduling 2008;11:29–47.

[4] Potts CN, Kovalyov MY, Scheduling with batching: a review. European Journal of Operational Research 2000;120:228–49.

[5] Mathirajan M, Sivakumar AI. A literature review, classification and simple meta-analysis on scheduling of batch processors in semiconductor. International Journal of Advanced Manufacturing Technology 2006;29:990–1001.

[6] Uzsoy R. Scheduling batch processing machines with incompatible job families. International Journal of Production Research 1995;33(10):2685–2708.

[7] Mehta SV, Uzsoy R. Minimizing total tardiness on a batch processing machine with incompatible job families. IIE Transactions 1998;30:165–178.

[8] Vepsalainen APJ, Morton TE. Priority rules for job shops with weighted tardiness costs. Management Science 1987;33(8):1035–47.

[9] Azizoglu M, Webster S. Scheduling a batch processing machine with incompatible job families. Computers & Industrial Engineering 2001;39:325–35.

[10] Perez IC, Fowler JW, Carlyle WM. Minimizing total weighted tardiness on a single batch process machine with incompatible job families. Computers & Operations Research 2005;32:327–41.

[11] Geiger CD, Uzsoy R. Learning effective dispatching rules for batch processor scheduling. International Journal of Production Research 2008; 46(6):1431–54.

[12] Koh SG, Koo PH, Kim DC, Hur WS. Scheduling a single batch processing machine with arbitrary job sizes and incompatible job families. International Journal of Producion Economics 2005;98:81–96.

[13] Kashan AH, Karimi B. Scheduling a single batch-processing machine with arbitrary job sizes and incompatible job families: An ant colony framework. Journal of the Operational Resarch Society 2008;59:1268–1280.

[14] Balasubramanian H, Mönch L, Fowler JW, Pfund ME. Genetic algorithm based scheduling of parallel batch machines with incompatible families to minimize

total weighted tardiness. International Journal of Production Research 2004;42(8):1621–38.

[15] Raghavan NRS, Venkataramana M. Scheduling parallel batch processors with incompatible job families using ant colony optimization. Proceedings of the IEEE International Conference on Automation Science and Engineering 2006:507–512.

[16] Koh SG, Koo PH, Ha JW, Lee WS. Scheduling parallel batch processing machines with arbitrary job sizes and incompatible job families. International Journal of Production Research 2004;42(19):4091–4107.

[17] Mathirajan M, Sivakumar AI. Minimizing total weighted tardiness on heterogeneous batch processing machines with incompatible job families. International Journal of Advanced Manufacturing Technology 2006;28:1038–47.

[18] Glassey CR, Weng WW. Dynamic batching heuristics for simultaneous processing. IEEE Transactions on Semiconductor Manufacturing 1991; 4(2):77–82.

[19] Fowler JW, Phillips DT, Hogg GL. Real-time control of multiproduct bulk-service semiconductor manufacturing processes. IEEE Transactions on Semiconductor Manufacturing 1992;5(2):158–63.

[20] Tangudu SK, Kurz ME. A branch and bound algorithm to minimise total weighted tardiness on a single batch processing machine with ready times and incompatible job families. Production Planning & Control 2006;17(7):728–41.

[21] Gupta AK, Sivakumar AI. Controlling delivery performance in semiconductor manufacturing using look ahead batching. International Journal of Production Research 2007;45(3);591–613.

[22] Kurz ME, Mason SJ. Minimizing total weighted tardiness on a batch-processing machine with incompatible job families and job ready times. International Journal of Production Research 2008;46(1):131–51.

[23] Malve S, Uzsoy R. A genetic algorithm for minimizing maximum lateness on parallel identical batch processing machines with dynamic job arrivals and incompatible job families. Computers & Operations Research 2007;34:3016–28.

[24] Mönch L, Balasubramanian H, Fowler JW, Pfund ME. Heuristic scheduling of jobs on parallel batch machines with incompatible job families and unequal ready times. Computers & Operations Research 2005; 32:2731–50.

[25] Reichelt D, Mönch L. Multiobjective scheduling of jobs with incompatible families on parallel batch machines. Lecture Notes in Computer Science 2006;3906:202–21.

[26] Wang CS, Uzsoy R. A genetic algorithm to minimize maximum lateness on a batch processing machine. Computers & Operations Research 2002;29:1621–40.

[27] Van Der Zee DJ. Dynamic scheduling of batch-processing machines with non-identical product size. International Journal of Production Research 2007;45(10): 2327–49.

[28] Kashan AH, Karimi B, Jenabi M. A hybrid genetic heuristic for scheduling parallel batch processing machines with arbitrary job sizes. Computers & Operations Research 2008;35:1084–98.

[29] Aytug H., Khouja M, Vergara FE. Use of genetic algorithms to solve production and operations management problems: a review. International Journal of Production Research 2003;41(17):3955–4009.

[30] Chaudhry SS, Luo W. Application of genetic algorithms in production and oprations management: a review. International Journal of Production Research 2005;43(19):4083–4101.

[31] Hart E, Ross P, Corne D. Evolutionary scheduling: a review. Genetic Programming and Evolvable Machines 2005;6:191–220.

[32] França PM, Mendes A, Moscato P. A memetic algorithm for the total tardiness single machine scheduling problem. European Journal of Operational Research 2001;132:224–42.

[33] Tseng LY, Lin YT. A hybrid genetic local search algorithm on the permutation flowshop scheduling problem. European Journal of Operational Research 2009;198:84–92.

[34] Essafi I, Mati Y, Dauzère-Pérès S. A genetic local search algorithm for minimizing total weighted tardines in the job-shop scheduling problem. Computers & Operations Research 2008;35(8):2599–2616.

[35] Chiang TC, Fu LC. A rule-centric memetic algorithm to minimize the numbe rof tardy jobs in the job shop. International Journal of Production Research 2008;46(24):6913–31.

[36] Ishibuchi H, Yoshida T, Murata T. Balance between genetic search and local search in memetic algorithms for multiobjective permutation flowshop scheduling. IEEE Transactions on Evolutionary Computation 2003;7(2): 204–23.

[37] Cheng HC, Chiang TC, Fu LC. Multiobjective job shop scheduling using memetic algorithm and shifting bottleneck procedures. Proceedings of IEEE Symposium on Computational Intelligence in Scheduling 2009; 15–21.

[38] Cheng HC, Chiang TC, Fu LC. A memetic algorithm for parallel batch machine scheduling with incompatible job families and dynamic job arrivals. Proceedings of IEEE International Conference on Systems, Man, and Cybernetics 2008; 541–46.

[39] Wall, M. GAlib: A C++ library of genetic algorithms components. http://lancet.mit.edu/ga/, 1999.

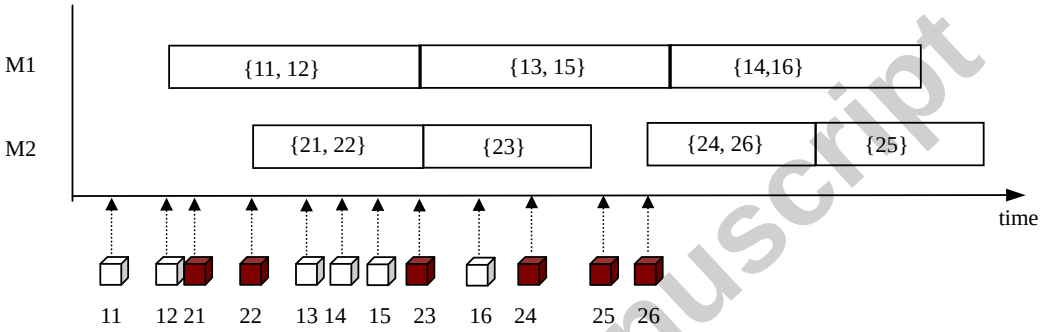


Fig. 1. An example of scheduling of parallel batch machines (m = 2, f = 2, $n_1 = n_2$ = 6)

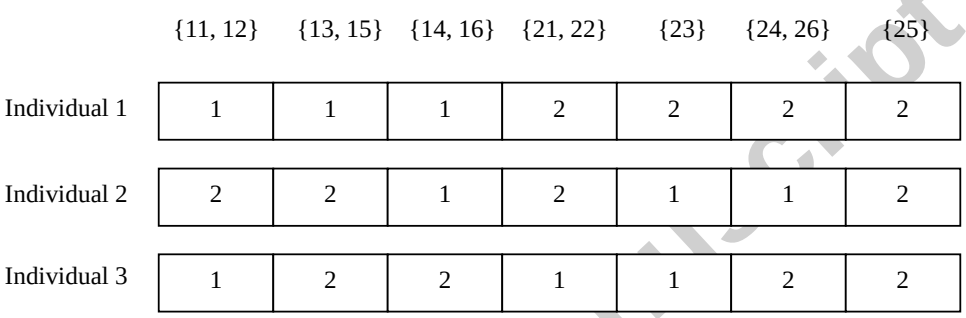


Fig. 2. Chromosome encoding considering machine assignment [24]

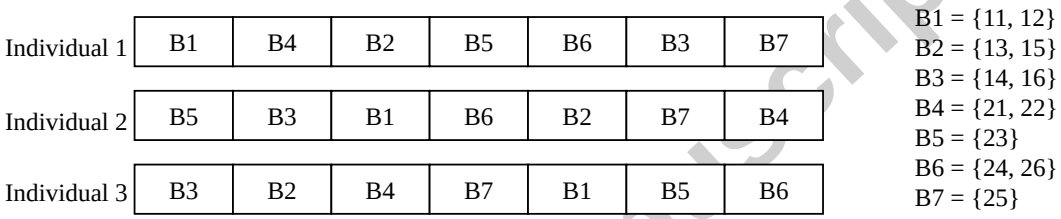


Fig. 3. Chromosome encoding considering batch sequence [38]

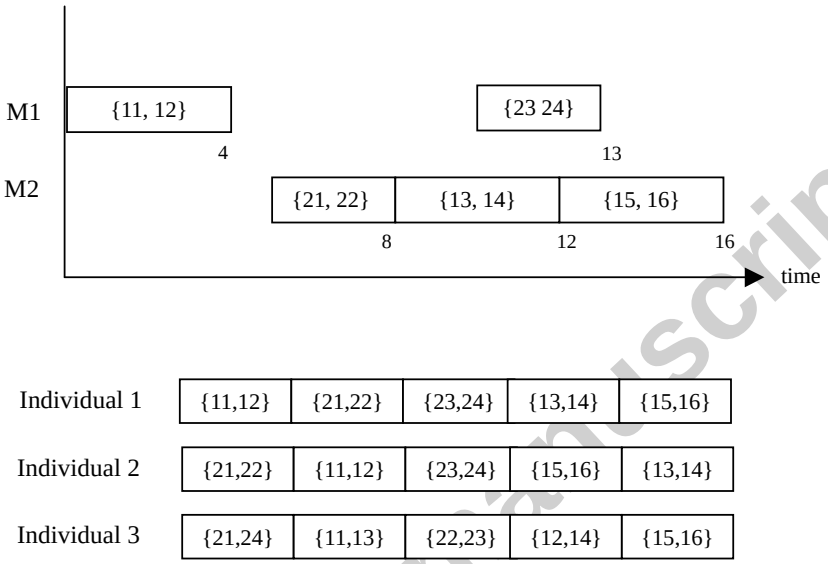
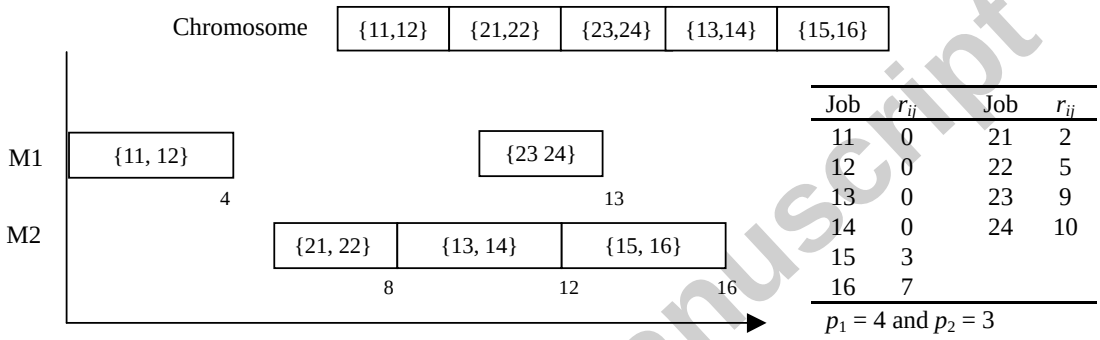


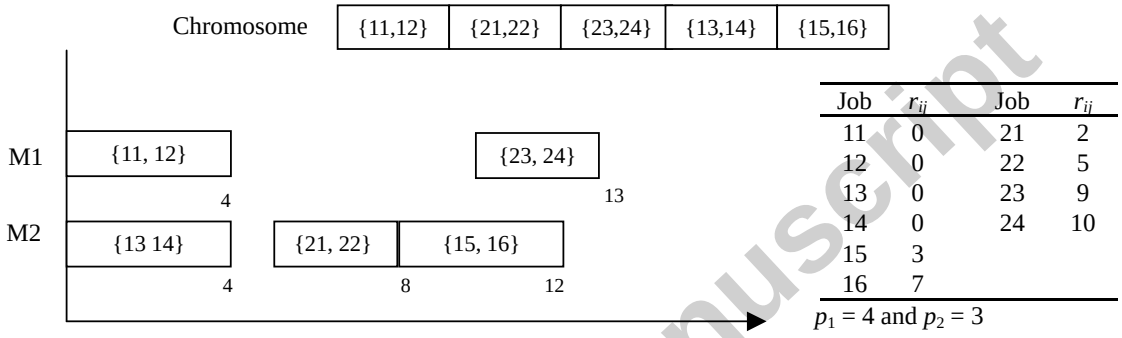
Fig. 4. Proposed chromosome encoding considering batch formation and sequence simultaneously



Job	r_{ij}	Job	r_{ij}
11	0	21	2
12	0	22	5
13	0	23	9
14	0	24	10
15	3		
16	7		

$p_1 = 4$ and $p_2 = 3$

Fig. 5. The basic decoding scheme – scheduling based on recorded batch formation and batch sequence



Job	r_{ij}	Job	r_{ij}
11	0	21	2
12	0	22	5
13	0	23	9
14	0	24	10
15	3		
16	7		

$p_1 = 4$ and $p_2 = 3$

Fig. 6. The improved decoding scheme – considering idle periods on machines

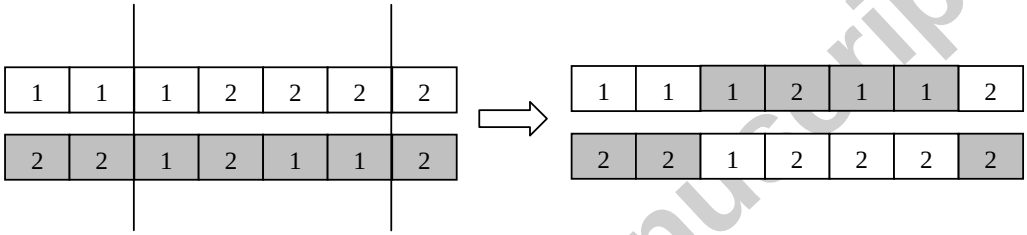


Fig. 7. Two-point crossover for chromosomes encoded by the scheme in Fig. 2

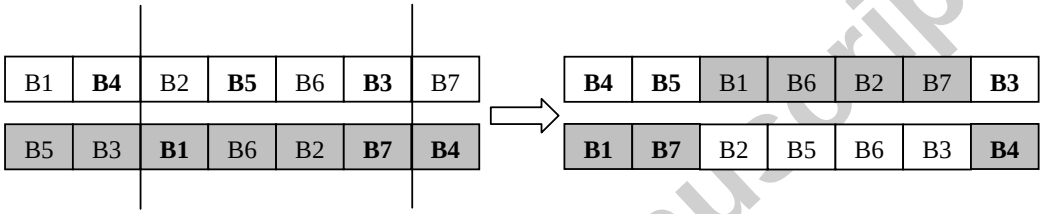


Fig. 8. Linear order crossover for chromosomes encoded by the scheme in Fig. 3

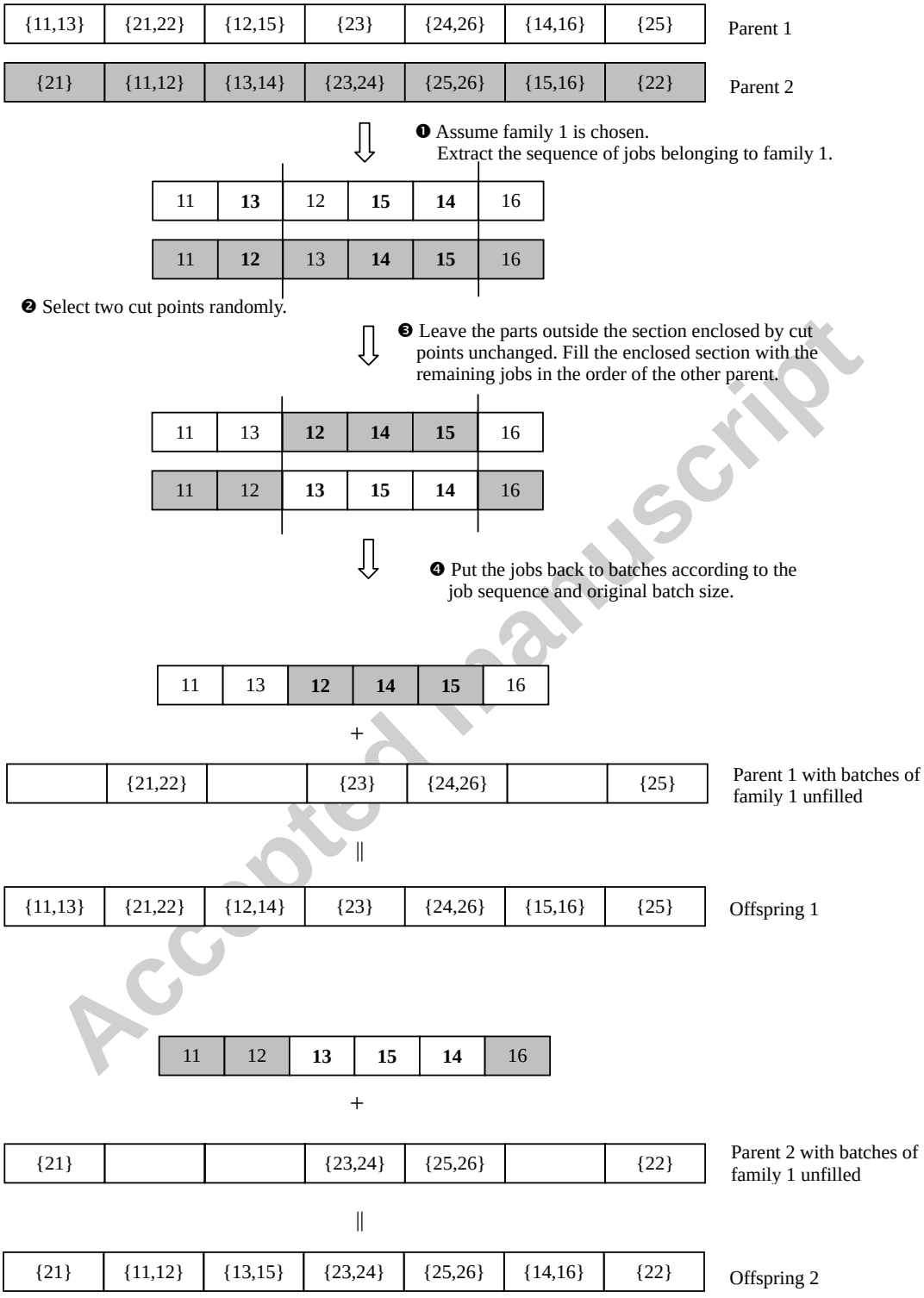


Fig. 9. Proposed batch formation crossover

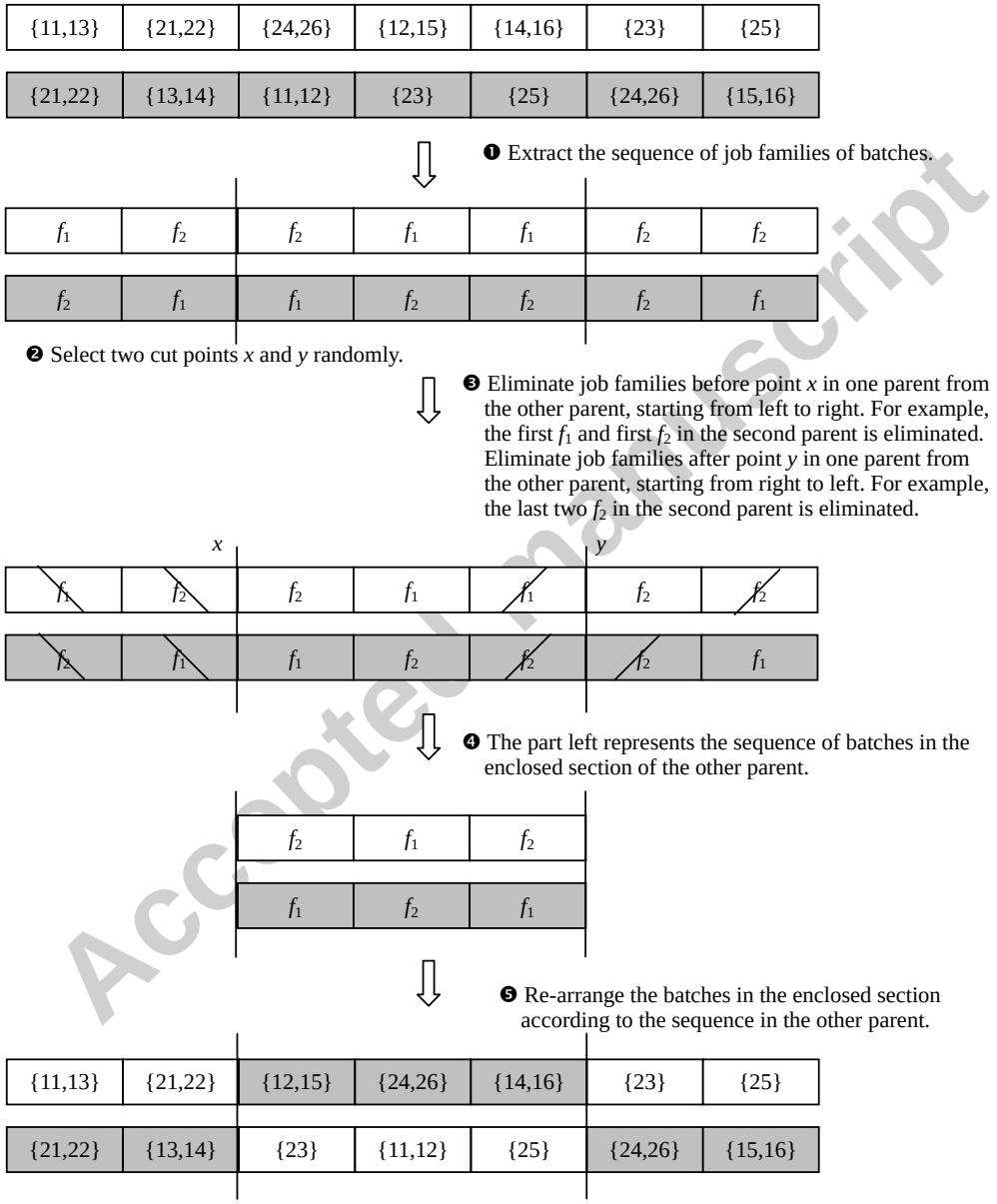


Fig. 10 Proposed batch sequence crossover

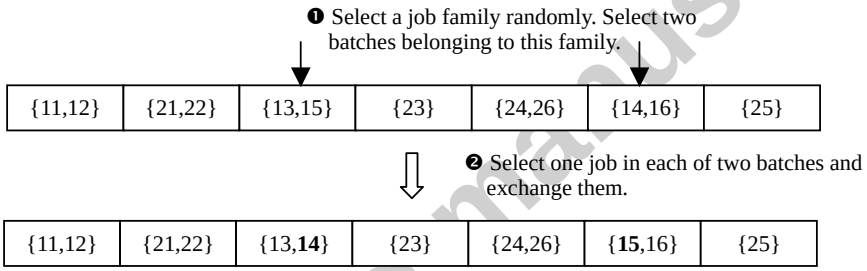


Fig. 11 Proposed batch formation mutation

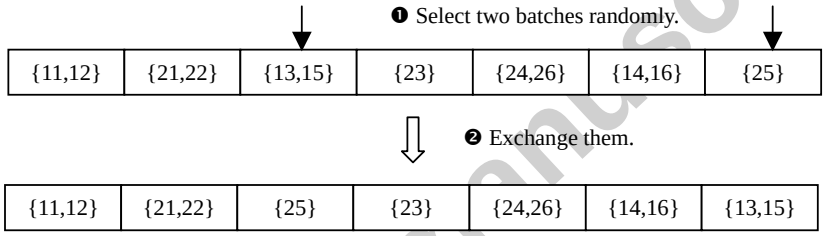


Fig. 12 Proposed batch sequence mutation

Table 1

Local search procedure

g: individual
$N_{EVAL,LS}$: the number of evaluations allocated for each invocation of local search
$P_{F,LS}$: the probability to apply the batch formation mutation operator
D_P: the maximum acceptable deviation percentage
rand(): return a random real value between 0 and 1
$BFM(g)$: return a neighboring solution using the batch formation mutation operator
$BSM(g)$: return a neighboring solution using the batch sequence mutation operator
$TWT(g)$: return total weighted tardiness of the schedule obtained by decoding g

```
LocalSearch(Chromosome g0)
Begin
    g = gbest = g0
    For t = 1 to NEVAL,LS
        If rand() < PF,LS Then
            g = BFM(g)
        Else
            g = BSM(g)

        If TWT(g) < TWT(gbest)
            gbest = g
        Else If TWT(g) > TWT(gbest)·(1+DP/100)
            g = gbest
        End if
    End for
    g0 = gbest
End
```

Table 2

Parameters for generation of problem instances

Problem parameter	Values used	Levels
Number of families (f)	3, 6, 12	3
Number of machines (m)	3, 4, 5	3
Number of jobs (n)	180, 240, 300	3
Maximum batch size (B)	4, 8	2
Family processing time (p_j)	2, 4, 10, 16, and 20 with probability 0.2, 0.2, 0.3, 0.2, and 0.1, respectively	1
Weight per job (w_{ij})	Uniform (0, 1)	1
Arrival time (r_{ij})	r_{ij} ~ Uniform (0, $\alpha \cdot \Sigma p_j/(mB)$) α = 0.25, 0.5, 0.75	3
Due date (d_{ij})	$d_{ij} - r_{ij}$ ~ Uniform (0, $\beta \cdot \Sigma p_j/(mB)$) β = 0.25, 0.5, 0.75	3

Table 3

Tuning of parameter values (N_{POP}, N_{GEN}, N_{LS}, and $N_{EVAL,\ LS}$): Average improvement percentage (%) and average computation time (s.) of proposed algorithm with different values of parameters. ($P_{F,GA}$ and $P_{F,LS}$ are both 0.5.)

	population size × maximum generation number		
	100×1000	200×500	400×250
No local search	23.14 / 4.60**	23.91 / 6.42	23.70 / 7.02
1 × 500*	**23.54 / 2.69**	24.78 / 5.22	25.47 / 7.93
5 × 100	24.64 / 4.03	25.40 / 6.58	25.88 / 8.74
10 × 50	24.86 / 4.50	25.70 / 7.23	26.05 / 9.05
1 × 1000	23.71 / 2.82	24.76 / 5.02	25.53 / 8.22
5 × 200	24.90 / 4.88	25.67 / 7.46	26.19 / 10.10
10 × 100	25.27 / 5.66	26.00 / 8.33	**26.43 / 10.70**

*Number of individuals doing local search × number of evaluations in local search

**Average improvement percentage / average computation time

Table 4
Tuning of parameter values ($P_{F,GA}$ and $P_{F,LS}$): Average improvement percentage (%) and average computation time (s.) of proposed algorithm with different values of parameters (N_{POP}, N_{GEN}, N_{LS}, and $N_{EVAL,\,LS}$ are 100, 1000, 1, and 500, respectively.)

$P_{F,LS}$ \ $P_{F,GA}$	0	0.25	0.5	0.75	1
0	14.12 / 1.06	20.01 / 4.19	23.04 / 6.14	24.14 / 6.69	24.54 / 6.81
0.25	20.35 / 1.37	22.63 / 2.35	23.55 / 2.98	**24.05 / 3.28**	24.05 / 3.44
0.5	21.51 / 1.52	23.07 / 2.25	23.54 / 2.68	23.87 / 2.89	23.82 / 2.92
0.75	22.01 / 1.76	23.19 / 2.46	23.63 / 2.86	23.64 / 2.96	23.38 / 2.82
1	22.18 / 2.18	23.41 / 3.03	23.51 / 3.63	22.70 / 3.92	18.23 / 2.90

Table 5
Tuning of parameter values ($P_{F,GA}$ and $P_{F,LS}$): Average improvement percentage (%) and average computation time (s.) of proposed algorithm with different values of parameters (N_{POP}, N_{GEN}, N_{LS}, and $N_{EVAL,\,LS}$ are 400, 250, 10, and 100, respectively.)

$P_{F,LS}$ \ $P_{F,GA}$	0	0.25	0.5	0.75	1
0	14.37 / 4.03	22.46 / 11.01	24.40 / 11.84	25.45 / 12.17	25.81 / 12.32
0.25	21.86 / 6.45	25.59 / 10.67	26.31 / 11.15	26.34 / 11.17	26.35 / 11.09
0.5	22.89 / 6.67	25.99 / 10.42	26.43 / 10.77	26.50 / 10.79	26.22 / 10.71
0.75	23.40 / 7.01	25.95 / 10.34	26.30 / 10.66	26.28 / 10.66	25.72 / 10.53
1	23.51 / 7.52	25.92 / 10.60	25.96 / 11.00	25.51 / 11.31	20.69 / 10.24

Table 6
Tuning of parameter values (N_{POP}, N_{GEN}, N_{LS}, and $N_{EVAL,\ LS}$): Average improvement percentage (%) and average computation time (s.) of proposed algorithm with different values of parameters. ($P_{F,GA}$ and $P_{F,LS}$ are 0.75 and 0.25, respectively.)

	population size × generation number		
	100×1000	200×500	400×250
No local search	23.63 / 4.80	24.50 / 6.59	24.16 / 7.21
1 × 500*	**24.05 / 3.28**	25.19 / 6.04	25.74 / 8.59
5 × 100	24.66 / 4.50	25.59 / 7.35	26.09 / 9.19
10 × 50	24.98 / 4.97	25.82 / 7.85	26.19 / 9.39
1 × 1000	24.09 / 3.54	25.08 / 6.19	25.71 / 9.32
5 × 200	25.12 / 5.41	25.82 / 8.14	26.30 / 10.71
10 × 100	25.26 / 6.18	26.10 / 8.94	26.34 / 11.15

Table 7
Performance comparison: Best-case, average-case, and worst-case average improvement percentage (%) and average computation time (s.) of benchmark and proposed algorithms.

	Benchmark [24]	Proposed MA
Number of families (f)		
3	12.8 / 10.8 / 8.1 (5.0)	29.3 / 26.4 / 23.3 (3.9)
6	12.0 / 10.1 / 7.6 (5.3)	24.6 / 21.7 / 18.7 (2.3)
12	10.7 / 9.2 / 7.1 (6.1)	20.0 / 17.5 / 14.6 (1.7)
Number of machines (m)		
3	13.2 / 11.4 / 9.0 (5.6)	27.2 / 24.2 / 21.0 (2.3)
4	11.9 / 10.1 / 7.6 (5.5)	24.5 / 21.7 / 18.7 (2.5)
5	10.4 / 8.5 / 6.1 (5.3)	22.2 / 19.6 / 16.8 (2.6)
Batch size (B)		
4	15.1 / 12.7 / 9.6 (9.0)	29.7 / 26.5 / 23.1 (3.1)
8	8.6 / 7.4 / 5.5 (1.9)	19.6 / 17.2 / 14.6 (1.8)
Number of jobs (n)		
180	8.7 / 7.1 / 5.0 (1.8)	19.7 / 17.0 / 14.2 (1.3)
240	12.0 / 10.2 / 7.6 (4.4)	24.9 / 22.1 / 19.1 (2.3)
300	14.8 / 12.7 / 10.0 (10.1)	29.3 / 26.4 / 23.2 (3.7)
Arrival time factor (α)		
0.25	5.9 / 5.0 / 3.7 (5.2)	15.4 / 13.1 / 10.8 (2.7)
0.50	12.7 / 10.8 / 8.4 (5.5)	26.8 / 24.0 / 21.0 (2.5)
0.75	16.9 / 14.2 / 10.6 (5.6)	31.7 / 28.4 / 24.8 (2.1)
Due date factor (β)		
0.25	10.0 / 9.0 / 7.5 (5.5)	18.2 / 16.2 / 14.0 (2.8)
0.50	11.2 / 9.4 / 7.0 (5.5)	24.2 / 21.4 / 18.3 (2.5)
0.75	14.3 / 11.7 / 8.2 (5.4)	31.5 / 28.0 / 24.2 (2.0)
Average	11.8 / 10.0 / 7.6 (5.5)	24.7 / 21.9 / 18.9 (2.4)