Journal of Scheduling **8:** 323–351, 2005.

A GENETIC ALGORITHM FOR HYBRID FLOW-SHOP SCHEDULING WITH MULTIPROCESSOR TASKS

CEYDA OĞUZ[1] AND M. FIKRET ERCAN[2]

[1] *Department of Logistics, The Hong Kong Polytechnic University, Kowloon, Hong Kong SAR*
[2] *School of Electrical and Electronics Engineering, Singapore Polytechnic, 500 Dover Rd, S139651 Singapore*

ABSTRACT

The hybrid flow-shop scheduling problem with multiprocessor tasks finds its applications in real-time machine-vision systems among others. Motivated by this application and the computational complexity of the problem, we propose a genetic algorithm in this paper. We first describe the implementation details, which include a new crossover operator. We then perform a preliminary test to set the best values of the control parameters, namely the population size, crossover rate and mutation rate. Next, given these values, we carry out an extensive computational experiment to evaluate the performance of four versions of the proposed genetic algorithm in terms of the percentage deviation of the solution from the lower bound value. The results of the experiments demonstrate that the genetic algorithm performs the best when the new crossover operator is used along with the insertion mutation. This genetic algorithm also outperforms the tabu search algorithm proposed in the literature for the same problem.

KEY WORDS: multiprocessor task scheduling, hybrid flow-shop, genetic algorithm

1. INTRODUCTION

Multiprocessor task scheduling problem is a generalization of the classical machine scheduling problem by allowing tasks to be processed on more than one processor at a time. Although multiprocessor task scheduling problems were motivated mainly by computer systems, such as fault-tolerant systems (Krawczyk and Kubale, 1985), applications of multiprocessor tasks can also be found in work-force assignment for manufacturing activities (Chen and Lee, 1999) and in berth allocation of container terminals (Lee and Cai, 1999). Another application of multiprocessor task scheduling is in real-time machine-vision systems. These systems utilize multiple layers of multiprocessor computing platforms where images have to pass through from one layer to another. Images are processed by several feature-extracting algorithms on parallel identical processors at each layer (Ercan and Fung, 1999). These systems can be analyzed from a scheduling perspective because: (1) their multi-layer structure with parallel identical processors resembles the multi-stage hybrid flow-shop environment and (2) each incoming image can be treated as a multiprocessor task as they can be processed on more than one processor simultaneously.

Motivated by the real-time machine-vision systems as described above, in this paper, we consider multiprocessor task scheduling problem to minimize makespan in hybrid flow-shop environments. This problem can be stated formally as follows: consider a set $J = \{1, 2, \ldots, n\}$ of n jobs to be processed in a k-stage flow-shop, where each stage i has m_i identical parallel processors

Correspondence to: Ceyda Oğuz, E-mail: lgtceyda@polyu.edu.hk, coguz@ku.edu.tr

(machines), $i = 1, 2, \ldots, k$. It is convenient to view a job as a sequence of k tasks—one task for each stage, where the processing of any task can commence only after the completion of the preceding task. Each task, within a job, requires one or several processors simultaneously at the corresponding stage. Let $size_{ij}$ and p_{ij} be the number of processors required and the time needed, respectively, to process job j at stage i; $i = 1, 2, \ldots, k$ and $j \in J$. In other words, task i of job j has to be processed on $size_{ij}$ of m_i identical parallel procesors of stage i for p_{ij} time units; $i = 1, 2, \ldots, k$ and $j \in J$. We have the following assumptions for the problem as well:

1. All processors and all jobs are available from time $t = 0$;
2. Processors used at each stage cannot process tasks corresponding to any other stages;
3. Each processor can process not more than one job at a time;
4. Preemption of jobs is not allowed.

Since all processors at each stage are identical and preemptions are not allowed, to define a schedule it suffices to specify the completion time for all tasks comprising each job. Let $C_{ij}(s)$ be the completion time of the ith task of job j in schedule s; $i = 1, 2, \ldots, k$ and $j \in J$. It is necessary to find a schedule minimizing the criterion $C_{\max}(s) = \max_{j \in J}\{C_{kj}(s)\}$. The problem of minimizing the criterion $C_{\max}(s)$ is referred to as the makespan minimization problem. Using well-known three-field notation (see, for example, Błażewicz et al., 2001), this problem can be denoted by $Fk(Pm_1, \ldots, Pm_k)|size_{ij}|C_{\max}$.

If we have $size_{ij} = 1$ for all $i = 1, 2, \ldots, k$ and $j \in J$, $Fk(Pm_1, \ldots, Pm_k)|size_{ij}|C_{\max}$ problem will be reduced to the classical hybrid flow-shop makespan minimization problem, in which a job can be processed on at the most one processor at a time. This problem is NP-hard even for $k = 2$ (Gupta, 1988). On the other hand, if we have $k = 1$, then we obtain the single-stage multiprocessor task scheduling problem to minimize makespan, which is also shown to be NP-hard (Lloyd, 1981).

The review on hybrid flow-shop scheduling problems by Vignier, Billaut, and Proust (1999) reveals that most studies consider the criterion of minimizing the makespan and use the branch-and-bound (see, for example, Portmann et al., 1998; Moursli and Pochet, 2000) or heuristic algorithms (see, for example, Haouari and M'Hallah, 1997; Riane, Artiba, and Elmaghraby, 1998) to solve the problem.

Extensive surveys on the multiprocessor task scheduling problems on the other hand were presented by Drozdowski (1996, 1997) and Lee, Lei, and Pinedo (1997). In particular, computational complexity results of different multiprocessor task scheduling problems were presented by Błażewicz, Drabowski, and Węglarz (1986), Błażewicz et al. (1990), Brucker et al. (2000) and Du and Leung (1989), and multiprocessor task scheduling problems in different shop environments were analyzed by Brucker and Krämer (1995, 1996).

Since $Fk(Pm_1, \ldots, Pm_k)|size_{ij}|C_{\max}$ problem is NP-hard, only small size problem instances can be solved exactly and this justifies the use of heuristic algorithms to obtain good approximate solutions to large size problem instances. Hence, in this paper, we focus on developing an efficient approximate algorithm to find a near optimal solution to $Fk(Pm_1, \ldots, Pm_k)|size_{ij}|C_{\max}$ problem. We also concentrate on the fact that the real-life problems, like in machine-vision systems, require a quick but a good solution. It is thus of interest to develop a fast approximation algorithm that could be employed in machine-vision systems, which can be modeled as a $Fk(Pm_1, \ldots, Pm_k)|size_{ij}|C_{\max}$ problem.

In the literature, there exists constructive heuristic algorithms (Oğuz and Ercan, 1997; Oğuz et al., 2003) and a tabu search algorithm (Oğuz et al., 2004) proposed for this problem. Considering

the success of the genetic algorithms developed for scheduling problems, we choose to use this search method to provide a good approximate solution to $Fk(Pm_1, \ldots, Pm_k)|size_{ij}|C_{max}$ problem. Genetic algorithms have been applied to different scheduling problems, see for example Caraffa et al. (2001) for blocking flow-shop environment; Chen, Vempati, and Aljaber (1995), Iyer and Saxena (2004) and Reeves (1995) for permutation flow-shop environment; Della Croce, Tadei, and Volta (1995) and Dorndorf and Pesch (1995) for job-shop environment. Portmann et al. (1998) proposed a genetic algorithm to update the upper bound in their branch-and-bound algorithm for a hybrid flow-shop scheduling problem. To the best of our knowledge, there are no genetic algorithms applied to scheduling problems with multiprocessor tasks in the literature.

The rest of the paper is organized as follows. Section 2 first describes the special features of the genetic algorithm and then presents the specific implementation details of our genetic algorithm. In particular we introduce a new crossover operator and consider one of the successful crossovers from the literature. Section 3 illustrates our computational experiments employed to test the performance of the genetic algorithm. After describing the data generation and the best parameter settings for the genetic algorithm proposed, we subsequently report the computational results in comparison with the performance of the tabu search algorithm of Oğuz et al. (2004). Section 4 concludes the paper.

2. GENETIC ALGORITHM AND ITS IMPLEMENTATION

Genetic algorithms, introduced by Holland (1975), are iterative stochastic algorithms in which natural evolution is used to model the search method. Genetic algorithms use a collection of solutions (population) which evolves through genetic operators (selection, crossover, mutation and replacement) to obtain better solutions. The process starts with an initial population of a certain size, which is usually randomly generated. In this population, solutions are encoded as a string (most of the time binary or integer) and are usually referred to as individuals or chromosomes. Each individual is composed of genes that characterize the solution. Each individual is evaluated by its fitness, which is determined by the associated value of the objective function. A new population is generated by applying the genetic operators to the individuals of the current population and this is considered as a generation. More specifically, during a generation, the individuals are evaluated and then the most fit individuals in the population (parents) are selected for the generation of new solutions (offspring). This generation takes place according to the genetic operators, such as mutation (introducing variations into the individuals) and crossover (taking the best features of each parent and mixing the remaining features). Hence, the new individuals are somewhat different than their parents. In the new generation, the fitness of the offspring is evaluated in a similar fashion as their parents, and the worst fitted individuals die to maintain the desired population. The birth and death processes define the population size but it usually remains constant from one generation to the next. This process is repeated until a stopping criterion is satisfied. The output of this simulated evolution process is the best individual in the final population, which can be a highly evolved solution to the problem.

Our proposed genetic algorithm adopts the general structure described above. In the following, we will explain the details of our implementation of the genetic algorithm for the $Fk(Pm_1, \ldots, Pm_k)|size_{ij}|C_{\max}$ problem.

2.1. Initial population

The initial population consists of *pop_size* randomly generated individuals, where *pop_size* is the population size to be kept constant through the generations. *pop_size* is one of the control parameters and its value was determined in our preliminary computational experiments, which is explained in Section 3.3.

2.2. Encoding of solutions (individuals)

A logical encoding of a solution to the $Fk(Pm_1, \ldots, Pm_k)|size_{ij}|C_{\max}$ problem is to consider the sequence of jobs at each stage considering the fact that different permutation of jobs may occur at different stages. In this representation, an individual will have k strings, each being a permutation of $1, 2, \ldots, n$, corresponding to the job list at different stages. However, one can easily observe that we can only search freely and randomly at the first stage for different permutations of jobs where all jobs are available at time zero. For each subsequent stage, permutations that deviate from the permutation of the preceding stage may result in a schedule with substantial waiting times. This effect will be more pronounced when the number of jobs is much larger than the number of processors available at each stage. Furthermore, this representation causes difficulties in defining the genetic operators.

Hence we chose to consider the sequence of jobs only at stage 1 in the encoding of a solution as an individual and then to decode each individual to a full schedule by using a generalization of List Scheduling algorithm to incorporate the jobs at other stages. In this encoding, a string of n integers, which is a permutation of $1, 2, \ldots, n$, corresponding to the job list at stage 1, is used to represent an individual. Each integer in the permutation, representing a job, is called a gene in genetic algorithm terminology.

2.3. Decoding of individuals

An individual is decoded to a schedule by employing a generalization of the List Scheduling algorithm (Algorithm LS). Algorithm LS constructs the schedule by iteratively assigning the jobs to the processors according to the list at the first stage in such a way that job completion times will be minimal. Then, a new list is created on the basis of the jobs' completion times from the first stage (non-decreasing). At the second stage, the jobs are iteratively assigned to the processors according to the new list and so on. It should be noted that Algorithm LS preserves the sequence of jobs in the list while assigning the jobs to the processors at a stage. This implies that a job will not be scheduled before the job that precedes it in the list even though there are enough processors and time available. However, since there are multiple processors at a stage, two jobs may start at the same time if there are enough processors available even though they are not at the same position in the list. Furthermore, in constructing the list for stage i, we consider only the completion times of the jobs at stage $i-1$ and the list for stage $i-1$ may become irrelevant; $i = 2, 3, \ldots, k$. That is, the list for stage i will be arranged in non-decreasing order of completion time of jobs at stage $i-1$ and if there are more than one job that complete at a certain time at stage $i-1$, then we consider the order of these jobs in the list for stage $i-1$ to construct the list for stage i; $i = 2, 3, \ldots, k$.

Let π_i denote the list for stage i and $\pi_i(h)$ denote the job at position h in list π_i; $i = 1, 2, \ldots, k$ and $h \in J$. A formal description of Algorithm LS is as follows:

Algorithm LS

Input: π_1, the sequence obtained from genetic algorithm (individual).

for $h = 1, 2, \ldots, n$ **do**
    Schedule job $\pi_1(h)$ to the first available $size_{1,\pi_1(h)}$ processors at stage 1 without violating the order of jobs in π_1.
end for
for $i = 2, 3, \ldots, k$ **do**
    Form the new list π_i by ordering the jobs according to the non-decreasing values of $C_{i-1,\pi_{i-1}(h)}$, $h \in J$.
    if $C_{i-1,\pi_{i-1}(h)} = C_{i-1,\pi_{i-1}(l)}$ and $h < l$ in π_{i-1} for any $\pi_{i-1}(h)$ and $\pi_{i-1}(l)$, $h, l \in J$ **then** order jobs in π_i assuming $C_{i-1,\pi_{i-1}(h)} < C_{i-1,\pi_{i-1}(l)}$.
    end if
    if $C_{i-1,\pi_{i-1}(h)} = C_{i-1,\pi_{i-1}(l)}$ and $h > l$ in π_{i-1} for any $\pi_{i-1}(h)$ and $\pi_{i-1}(l)$, $h, l \in J$ **then** order jobs in π_i assuming $C_{i-1,\pi_{i-1}(l)} < C_{i-1,\pi_{i-1}(h)}$.
    end if
    for $h = 1, 2, \ldots, n$ **do**
        Schedule job $\pi_i(h)$ to the first available $size_{i,\pi_i(h)}$ processors at stage i without violating the order of jobs in π_i.
    end for
end for

As it can be seen from the above description, Algorithm LS will create a semi-active schedule for each sequence obtained from the genetic algorithm because jobs are scheduled as soon as possible by preserving the job order defined by the list. Furthermore, Algorithm LS will build a non-permutation schedule, that is the sequence of jobs at each stage may be different. In the following, we present an example to illustrate how Algorithm LS decodes a given permutation and to emphasize the issues discussed above.

Example 1. Consider nine jobs to be scheduled in a two-stage hybrid flow-shop, with five processors at both stages. The processing times and the processor requirements of the jobs are given in Table 1.

Table 1. Data for Example 1

j	1	2	3	4	5	6	7	8	9
p_{1j}	4	5	5	4	3	2	1	1	2
$size_{1j}$	1	3	3	3	3	1	2	2	2
p_{2j}	2	6	2	1	1	4	1	2	1
$size_{2j}$	4	5	2	5	3	2	1	2	3

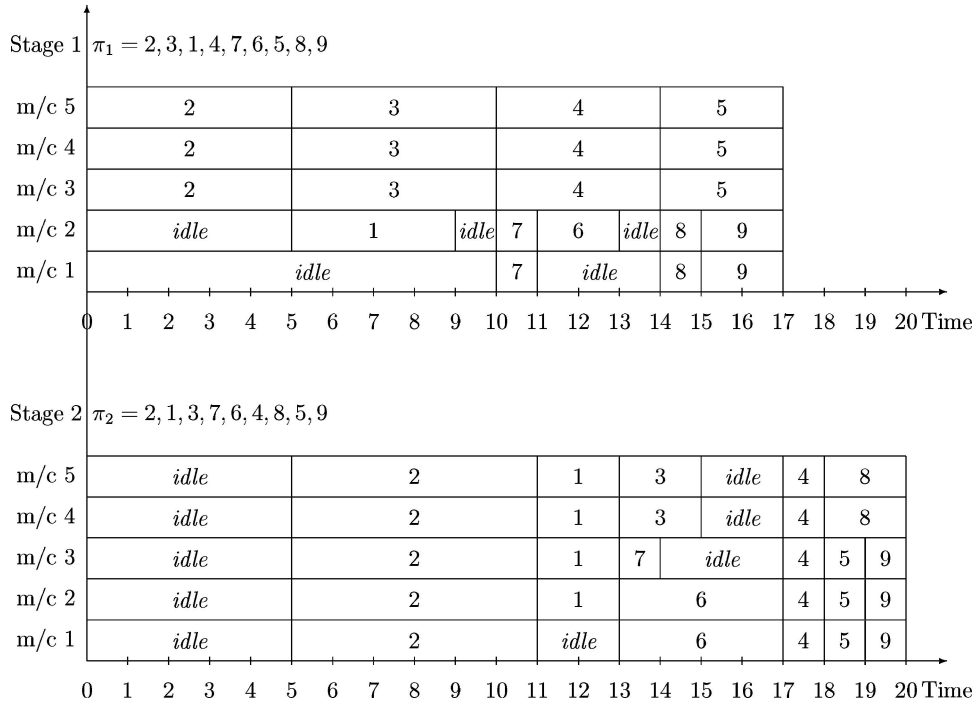

Figure 1. The schedule of the permutation 2, 3, 1, 4, 7, 6, 5, 8, 9 after being decoded by the List Scheduling algorithm. The idle periods on the machines are denoted by "*idle*".

Let us assume that genetic algorithm produced the permutation 2, 3, 1, 4, 7, 6, 5, 8, 9 as the best solution. Algorithm LS will decode this sequence into a schedule for $F2(P5, P5)|size_{ij}|C_{\max}$ problem as given in Figure 1.

At stage 1 the list $\pi_1 = 2, 3, 1, 4, 7, 6, 5, 8, 9$, and we schedule the jobs by iteratively assigning them to the processors according to π_1 and their processor requirements starting at time 0. After scheduling jobs 2 and 3, the next job to be scheduled is job 1. Even though the number of available processors at time 0 is enough for the processor requirement of job 1, we do not schedule it starting at time 0 because this will violate the precedence relation between jobs 3 and 1 in list π_1. Hence we schedule job 1 starting at time 5 since the number of available processors is enough for the processor requirement of job 1. As a result both jobs 3 and 1 start at the same time, which is not a violation of the precedence relation in list π_1. Similarly after scheduling job 4 at time 10, since there are processors available for job 7, we schedule job 7 at time 10 as well. However, we do not schedule job 7 at an earlier time even though the number of available processors is enough for the processor requirement of job 7 because this will again violate the order given by list π_1. We continue this way to schedule all jobs at stage 1 in the order given by list π_1.

In the next step we obtain the new list π_2 to be used to schedule jobs at stage 2. π_2 is constructed by listing jobs in non-decreasing order of their completion time at stage 1. Hence the new list π_2 will be 2, 1, 3, 7, 6, 4, 8, 5, 9. We observe that this list is different than π_1 so we do not consider only permutation schedules.

We schedule jobs at stage 2 according to the order given by list π_2 and by considering the completion time of jobs at stage 1; that is, jobs cannot start at stage 2 before their completion time at stage 1. After scheduling jobs 2, 1 and 3 according to their processor requirements at stage 2, the next job to be scheduled is job 7 and since the number of available processors is enough for the processor requirement of job 7 at time 13, we schedule it starting at time 13. Similarly, we schedule the next job in list π_2, which is job 6, starting at time 13 so that all three jobs 3, 7, and 6 start at the same time. This does not violate the precedence relation among jobs 3, 7, and 6. However, we do not schedule job 7 starting from time 11 even though it completes its processing at stage 1 by time 11 because doing so will violate the precedence constraint between jobs 3 and 7. After scheduling job 4 next, we schedule job 8 starting at time 18. We then schedule job 5 and then job 9, starting at times 18 and 19, respectively. We remark that even though jobs 5 and 9 complete at the same time at stage 1, job 5 precedes job 9 in list π_2 because this is the case in π_1.

2.4. Fitness

The fitness of an individual is defined by the $C_{\max}$ value of the corresponding schedule obtained by decoding the individual using Algorithm LS as described above. We observe that even though an individual represents a sequence of jobs at stage 1 only, it is evaluated for its fitness by the $C_{\max}$ value after being decoded to a full schedule corresponding to the $Fk(Pm_1, \ldots, Pm_k)|size_{ij}|C_{\max}$ problem.

2.5. Selection

Individuals from the current population are selected based on their fitness value to form the mating pool for the reproduction step. The aim of selection is to keep good individuals and eliminate the bad ones from one generation to another. This selection is performed by using the roulette wheel method in our implementation. In this method, a roulette wheel with slots assigned to each individual of the population and sized proportional to the fitness of individuals is used for the selection process. We construct such a roulette wheel as follows (Michewicz, 1996):

1. Let $1/f(l)$ denote the fitness value, that is $C_{\max}$, of individual l; $l = 1, 2 \ldots, pop\_size$.
2. Find the total value of the population

$$F = \sum_{l=1}^{pop\_size} f(l).$$

3. Calculate the probability of a selection p_l for each individual l, $l = 1, 2 \ldots, pop\_size$:

$$p_l = f(l)/F.$$

4. Calculate the cumulative probability q_l for each individual l, $l = 1, 2 \ldots, pop\_size$:

$$q_l = \sum_{h=1}^{l} p_h.$$

The selection process is based on spinning the roulette wheel *pop_size* times; each time we select a single individual in the following way:

Algorithm Roulette Wheel

1. Generate a random number r from the range (0..1].
2. If $r < q_1$ then select the first individual; otherwise select the lth individual ($2 \leq l \leq pop\_size$) such that $q_{l-1} < r \leq q_l$.

We remark that with this method some individuals can be selected more than once or not at all.

2.6. Reproduction

New solutions (offspring) for the next generation are obtained by applying the following two genetic operators to the mating pool obtained in the previous step:

- Crossover, which aims to take the best features of each parent and mixing the remaining features in forming the offspring.
- Mutation, which aims to introduce variations into the individuals.

In employing crossover operator our aim is to introduce improvement and to explore more of the solution space. In other words, crossover helps genetic algorithm to converge to the best individual. On the other hand, mutation operator is used to introduce variability and diversity to the population so that the genetic algorithm will be able to escape from local optima.

Related to these two genetic operators, we have to determine the crossover rate (c), which is the probability of applying the crossover operator to the parents, and the mutation rate (m), which is the probability of applying the mutation operator to an individual. These parameters were determined during our preliminary computational experiments to set the best combination of different parameters, and are explained in Section 3.3.

2.6.1. Crossover

We introduce a new crossover operator (NXO) for our proposed genetic algorithm, which will capture some characteristics of $Fk(Pm_1, \ldots, Pm_k)|size_{ij}|C_{\max}$ problem. The aim in developing NXO is to keep the best characteristics of the parents in terms of the neighboring jobs. We can accept that if two jobs are adjacent to each other in both parents, which are having good fitness values, then we want to keep this structure in the offspring. If we cannot find such a structure, then we want to choose the next job which will fit well in terms of the processor allocations. These two selection criteria will help us to minimize the idle time on the processors. To compare the performance of NXO, in our computational experiments, we use one of the best performing crossover operators for the scheduling problems as reported in the literature, which is Partially Matched Crossover (PMX) (Goldberg and Lingle, 1985). PMX can be viewed as an extension of two-point crossover with a repairing procedure to convert infeasible permutations into feasible ones. We note that NXO, unlike other crossover operators designed for permutation representation such as PMX, will always produce legal offspring, that is, the permutation obtained will be feasible and we do not need a repairing procedure.

In the following description of NXO, a gene of a parent is available if this gene is not part of the offspring yet. Similarly, a gene of a parent is not available if this gene is already part of the offspring or if the previous gene is the last gene of the parent. Furthermore, a gene fits better than the other if the processor requirement of its corresponding job at stage 1 is greater than or equal to that of the other.

NXO

Input: Two parents from the current population.

1. **Initialization:**
   Select one of the parents randomly and call it *parent1*.
   Call the other parent *parent2*.
   Select the first gene of parent1 and record it as the first gene of the offspring.
   Call the recorded gene of the offspring *selected_gene*.

2. **Main loop**:
   Call the next gene of the *selected_gene* in parent1 *next_gene*1.
   Call the next gene of the *selected_gene* in parent2 *next_gene*2.
   if *next_gene*1 is available **and** *next_gene* 2 is available **then** select the gene for the offspring that fits better and call it *temp*.
   if *next_gene*1 is available **and** *next_gene*2 is *not* available **then** select *next_gene*1 for the offspring and call it *temp*.
   if *next_gene*1 is *not* available **and** *next_gene*2 is available **then** select *next_gene*2 for the offspring and call it *temp*.
   if both *next_gene*1 and *next_gene*2 are *not* available **then** find the first available gene of parent1 and of parent2, starting from the position of *next_gene*1 and *next_gene*2, respectively (if *next_gene*1 or *next_gene*2 is the last gene of the respective parent, then start from the first gene of that parent to search for the first available gene); select the gene for the offspring that fits better and call it *temp*.

   Record *temp* as the next gene of the *selected_gene* in the offspring.
   Update the *selected_gene* of the offspring with *temp*.
3. **Stopping criterion**:
   if all the genes of the offspring are completed **then** stop; **else** go to step 2.

We consider the following example to show the steps of this proposed crossover operator NXO.

Example 2. Consider a nine-job problem where $m_1 = 8$ and $size_{ij}$ values are as given in Table 2. Suppose the parent chromosomes are $P1 : 1, 2, 3, 4, 5, 6, 7, 8, 9$ and $P2 : 5, 4, 6, 9, 2, 1, 7, 8, 3$.

If we follow the description of the NXO, the construction of the offspring will be as follows:

1. Let us assume P1 is chosen at step 1. So the first gene of the offspring will be 1: *parent*1 $\leftarrow$ P1, *parent2* $\leftarrow$ P2, and *selected_gene* $\leftarrow$ 1.
   Offspring: 1.

Table 2. Data for Example 2

j	1	2	3	4	5	6	7	8	9
$size_{1j}$	8	2	5	2	2	6	4	4	3

Table 3. APD of different algorithms for processor setting with $m_i \sim [1; 5]$

k	n	TS	NXO/insM	NXO/swpM	PMX/insM	PMX/swpM
2	5	**13.94**	**13.94**	**13.94**	**13.94**	14.87
2	10	**1.55**	1.60	2.60	3.08	2.60
2	20	6.68	**0.80**	0.89	0.84	0.89
2	50	4.41	**0.69**	0.79	0.74	0.91
2	100	0.57	**0.35**	**0.35**	0.44	0.79
	Average	5.43	**3.48**	3.71	3.81	4.01
5	5	28.19	25.99	26.03	**25.97**	26.43
5	10	13.30	12.00	12.54	**11.89**	12.88
5	20	12.97	**5.54**	6.17	9.57	9.90
5	50	13.18	**5.11**	5.98	8.13	8.25
5	100	4.24	**3.06**	3.41	4.09	3.21
	Average	14.38	**10.34**	10.83	11.93	12.13
8	5	24.71	**21.65**	22.45	22.68	24.22
8	10	27.85	**16.14**	17.45	18.92	19.01
8	20	27.74	**7.98**	8.25	9.04	8.25
8	50	21.99	**6.03**	9.17	8.42	7.10
8	100	10.22	4.50	4.50	**4.12**	6.76
	Average	22.50	**11.26**	12.36	12.64	13.07
	Total average	14.10	**8.36**	8.97	9.46	9.74

The stopping criterion is the maximum number of generations/iterations performed.

2. $next\_gene1 \leftarrow 2$, $next\_gene2 \leftarrow 7$. Since both $next\_gene1$ and $next\_gene2$ are available, we select the one that fits better, which is 7 ($size_{17} = 4 > size_{12} = 2$).
   Offspring: 1, 7.
3. $next\_gene1 \leftarrow 8$, $next\_gene2 \leftarrow 8$. Since $next\_gene1 = next\_gene2$ and it is available, we select 8 as the next gene of the offspring.
   Offspring: 1, 7, 8.
4. $next\_gene1 \leftarrow 9$, $next\_gene2 \leftarrow 3$. Since both $next\_gene1$ and $next\_gene2$ are available, we select the one that fits better, which is 3 ($size_{13} = 5 > size_{19} = 3$).
   Offspring: 1, 7, 8, 3.
5. $next\_gene1 \leftarrow 4$ and $next\_gene2$ is not available because 3 is the last gene of P2. So we select 4 as the next gene of the offspring.
   Offspring: 1, 7, 8, 3, 4.
6. $next\_gene1 \leftarrow 5$, $next\_gene2 \leftarrow 6$. Since both $next\_gene1$ and $next\_gene2$ are available, we select the one that fits better, which is 6 ($size_{16} = 6 > size_{15} = 2$).
   Offspring: 1, 7, 8, 3, 4, 6.
7. $next\_gene1 \leftarrow 7$, which is not available since 7 is already part of the offspring, and $next\_gene2 \leftarrow 9$. So we select 9 as the next gene of the offspring.
   Offspring: 1, 7, 8, 3, 4, 6, 9.

Table 4. The best and the worst $PD_A(l)$ of different algorithms for processor setting with $m_i \sim [1; 5]$

k	n		TS	NXO/insM	NXO/swpM	PMX/insM	PMX/swpM
2	5	Best	0	0	0	0	0
		Worst	37.68	37.68	37.68	37.68	37.68
2	10	Best	0	0	0	0	0
		Worst	26.49	5.48	7.12	6.04	7.12
2	20	Best	0	0	0	0	0
		Worst	18.74	2.04	3.14	3.77	3.07
2	50	Best	0	0	0	0	0
		Worst	23.21	1.65	1.65	1.95	2.22
2	100	Best	0.06	0	0	0	0
		Worst	18.67	1.98	1.22	1.95	2.76
5	5	Best	4.88	4.88	6.29	7.45	8.95
		Worst	55.68	38.55	42.91	45.67	46.20
5	10	Best	6.93	6.21	7.82	6.93	9.83
		Worst	25.94	22.72	18.99	16.41	33.62
5	20	Best	3.55	3.55	3.55	5.16	7.35
		Worst	23.86	18.83	17.21	21.45	23.30
5	50	Best	2.64	2.64	2.64	2.64	2.64
		Worst	23.78	13.54	14.88	17.43	16.72
5	100	Best	1.23	1.23	1.23	1.23	1.23
		Worst	18.34	15.32	17.29	16.32	15.32
8	5	Best	8.12	3.96	3.96	4.65	3.96
		Worst	57.40	44.16	55.36	57.40	55.36
8	10	Best	4.25	4.25	5.35	7.43	4.25
		Worst	48.94	34.65	39.72	35.03	35.21
8	20	Best	6.22	5.78	5.23	6.78	5.32
		Worst	47.07	16.43	18.09	17.21	22.32
8	50	Best	7.78	3.12	3.04	4.43	3.12
		Worst	20.17	14.98	15.33	13.44	18.92
8	100	Best	2.14	1.15	1.15	1.34	1.56
		Worst	28.16	12.09	10.38	12.36	16.73

The stopping criterion is the maximum number of generations/iterations performed.

8. *next_gene* 1 is not available because 9 is the last gene of P1 and *next_gene*2 $\leftarrow$ 2. So we select 2 as the next gene of the offspring.
   Offspring: 1, 7, 8, 3, 4, 6, 9, 2.
9. *next_gene*1 $\leftarrow$ 3, *next_gene*2 $\leftarrow$ 1. However both of them are not available since they are already part of the offspring. So we select 5 as the next gene of the offspring since 5 is the first gene available in P1 and P2 starting from the positions of *next_gene*1 and *next_gene*2.
   Offspring: 1, 7, 8, 3, 4, 6, 9, 2, 5.
10. Since all genes of the offspring are completed, 1, 7, 8, 3, 4, 6, 9, 2, 5 will be one of the offspring produced from parents P1 and P2. The other offspring will be created in a similar way by choosing P2 at step 1.

Table 5. Average CPU time spent (in seconds) by algorithms TS and GA[a] for processor setting with $m_i \sim [1; 5]$

k	n	TS	GA
2	5	1.86	2.09
2	10	6.30	8.86
2	20	23.72	26.10
2	50	164.08	178.22
2	100	483.09	567.14
5	5	3.93	4.01
5	10	13.73	18.41
5	20	36.65	47.56
5	50	287.00	367.13
5	100	786.00	1024.87
8	5	5.46	7.05
8	10	19.77	35.97
8	20	70.92	103.26
8	50	419.09	553.87
8	100	1338.71	1899.03

The stopping criterion is the maximum number of generations/iterations performed.

[a]Since each genetic algorithm spends similar amount of CPU time to solve a problem, we present here just the average value for four of the genetic algorithms.

We now present the details of PMX. As it can be seen PMX may produce illegal offspring (infeasible solutions) at step 2 so there is a need for a repair mechanism. This repair mechanism is introduced in terms of the mapping relationship between mapping sections as described.

PMX

Input: Two parents from the current population.

1. Select two positions along the parents uniformly at random. The sub-chromosome defined by the two positions is called the mapping section.
2. Exchange two mapping sections between parents to produce proto-children.
3. Determine the mapping relationship between two mapping sections, that is, map each gene in one mapping section with the gene in the other mapping section that occupies the same position.
4. Legalize offspring for the genes which are outside the mapping sections with the mapping relationship.

In the following, we consider Example 2 to show the workings of the PMX operator.

Example 2 (Contd). Let us recall that the parents chosen randomly are P1: 1, 2, 3, 4, 5, 6, 7, 8, 9 and P2 : 5, 4, 6, 9, 2, 1, 7, 8, 3.

Table 6. APD of different algorithms for processor setting with $m_i = 5$

k	n	TS	NXO/insM	NXO/swpM	PMX/insM	PMX/swpM
2	5	26.76	**26.75**	28.87	**26.75**	29.10
2	10	7.89	**6.13**	9.08	7.98	8.02
2	20	9.03	**7.10**	7.78	7.56	7.25
2	50	4.12	**3.34**	3.59	3.88	3.78
2	100	4.75	**2.87**	3.29	3.17	3.10
	Average	10.51	**9.24**	10.52	9.88	10.25
5	5	29.35	**27.11**	28.98	29.88	30.78
5	10	27.57	11.67	**11.32**	11.42	12.19
5	20	23.03	**10.78**	11.88	12.13	12.09
5	50	34.24	**14.91**	14.96	**14.91**	15.71
5	100	27.73	**11.02**	**11.02**	13.58	12.42
	Average	28.38	**15.10**	15.63	16.39	16.64
8	5	**37.08**	**37.08**	38.46	39.89	38.94
8	10	36.36	**25.98**	**25.98**	27.67	26.65
8	20	33.10	**24.13**	**24.13**	24.94	25.06
8	50	34.20	**21.87**	21.98	22.78	22.39
8	100	25.64	**19.46**	19.81	20.17	20.02
	Average	33.28	**25.70**	26.07	27.09	26.61
	Total average	24.06	**16.68**	17.41	17.78	17.84

The stopping criterion is the maximum number of generations/iterations performed.

1. Select two positions along the parents uniformly at random. Assume the positions are 3 and 6. Then the mapping sections are 3, 4, 5, 6 in P1 (1, 2 | 3, 4, 5, 6 | 7, 8, 9) and 6, 9, 2, 1 in P2 (5, 4 | 6, 9, 2, 1 | 7, 8, 3).
2. Exchange two mapping sections between parents. Then
   proto-child 1: 1, 2 | 6, 9, 2, 1 | 7, 8, 9
   proto-child 2: 5, 4 | 3, 4, 5, 6 | 7, 8, 3
   In this example, the underlined genes 1, 2, and 9 are duplicated in proto-child 1, while genes 3, 4, and 5 are missing from it. Similarly, the underlined genes 5, 4, and 3 are duplicated in proto-child 2, while genes 1, 2, and 9 are missing from it. Hence both proto-children are infeasible. In the next step we will determine the mapping relationship to use it in the repair procedure.
3. Determine the mapping relationship: by mapping 6 to 3, 9 to 4, 2 to 5, and 1 to 6, we obtain $1 \leftrightarrow 6 \leftrightarrow 3$, $2 \leftrightarrow 5$, and $9 \leftrightarrow 4$.
4. Legalize offspring with the mapping relationship.
   offspring 1: 3, 5, 6, 9, 2, 1, 7, 8, 4
   offspring 2: 2, 9, 3, 4, 5, 6, 7, 8, 1
   According to the mapping relationship determined in step 3, the excessive genes, 1, 2, and 9 should be replaced by the missing genes 3, 5, and 4, respectively, in P1 while keeping the swapped mapping section unchanged. Similarly, the excessive genes, 5, 4, and 3 should be

Table 7. The best and the worst $PD_A(l)$ of different algorithms for processor setting with $m_i = 5$

k	n		TS	NXO/insM	NXO/swpM	PMX/insM	PMX/swpM
2	5	Best	1.57	1.57	1.57	1.57	1.57
		Worst	68.70	70.03	78.16	70.03	79.76
2	10	Best	0	0.57	4.07	4.07	0.57
		Worst	17.17	20.2	26.90	23.21	23.21
2	20	Best	1.32	0	0	0	0
		Worst	27.47	15.95	21.30	15.95	15.51
2	50	Best	0	0	0	0	0
		Worst	7.52	7.52	7.52	8.94	8.94
2	100	Best	0.09	0.97	0.09	1.03	1.82
		Worst	5.42	4.03	4.67	6.46	6.56
5	5	Best	10.32	9.67	10.32	10.32	10.32
		Worst	69.43	52.83	51.09	56.03	52.83
5	10	Best	2.94	1.81	2.21	2.21	2.90
		Worst	31.71	28.63	27.18	27.99	33.56
5	20	Best	12.80	6.35	7.08	8.12	7.08
		Worst	34.73	22.39	19.72	20.52	21.60
5	50	Best	27.13	12.09	13.34	11.92	12.22
		Worst	37.98	19.52	21.09	20.01	23.27
5	100	Best	15.34	8.79	8.79	8.88	9.24
		Worst	40.96	15.53	15.53	19.20	16.43
8	5	Best	0	0	0	0	0
		Worst	60.55	60.55	60.55	60.55	60.55
8	10	Best	13.94	13.94	13.94	14.12	13.94
		Worst	51.46	49.35	49.35	49.35	49.35
8	20	Best	23.55	12.57	12.57	13.37	13.68
		Worst	56.07	28.38	28.38	28.26	31.07
8	50	Best	18.93	12.61	12.36	11.78	11.12
		Worst	66.21	41.87	46.55	43.18	42.63
8	100	Best	13.45	10.12	11.92	10.45	13.82
		Worst	53.91	33.51	34.38	35.24	33.29

The stopping criterion is the maximum number of generations/iterations performed.

replaced by the missing genes 2, 9, and 1, respectively, in P2 while keeping the swapped mapping section unchanged.

2.6.2. Mutation

We consider two mutation operators, insertion (insM) and swap (swpM), which are widely used in the literature (Gen and Cheng, 1997; Michalewicz, 1996). In our computational experiments, we compare the performance of each mutation operator along with two crossover operators.

insM: Insertion mutation selects a gene from the individual at random and inserts it in a random position. So if the individual is 1, 2, 3, 4, 5, 6, 7, 8, 9 and we select a gene at random, say gene

Table 8. Average CPU time spent (in seconds) by algorithms TS and GA[a] for processor setting with $m_i = 5$

k	n	TS	GA
2	5	3.49	4.03
2	10	14.11	12.14
2	20	53.78	36.51
2	50	290.67	270.23
2	100	1066.69	708.55
5	5	7.57	12.88
5	10	25.77	33.21
5	20	94.39	86.08
5	50	482.83	434.23
5	100	1899.51	1306.12
8	5	10.26	13.98
8	10	35.19	45.09
8	20	124.00	225.12
8	50	857.83	982.12
8	100	3181.32	2109.90

The stopping criterion is the maximum number of generations/iterations performed.

[a] Since each genetic algorithm spends similar amount of CPU time to solve a problem, we present here just the average value for four of the genetic algorithms.

6, and then insert it in a random position, say position 3, then we have the new (mutated) individual as 1, 2, 6, 3, 4, 5, 7, 8, 9.

swpM: Swap mutation selects two genes at random, and exchanges their positions. Again if the individual is 1, 2, 3, 4, 5, 6, 7, 8, 9, we select two genes randomly, say genes 4 and 8. We then exchange their positions, then the new (mutated) individual is 1, 2, 3, 8, 5, 6, 7, 4, 9.

2.7. Stopping criterion

Stopping criterion is one of the important decisions in the design of a genetic algorithm since it affects the performance of the algorithm. In our computational experiments, after setting the best combination of the other control parameters, namely the population size, crossover rate and mutation rate, we used two different stopping criteria separately and analyzed the performance of the proposed genetic algorithm under each stopping criterion. We first considered the maximum number of generations performed as the stopping criterion for each algorithm and the maximum number was chosen as 10,000. We then considered the maximum CPU time allowed to solve a problem as the stopping criterion and we set a limit of 30 min CPU time for each algorithm considered. We selected the maximum value of each stopping criterion to be in line with the tabu search algorithm of Oğuz et al. (2004).

Table 9. APD of different algorithms for processor setting with $m_i \sim [1; 5]$

k	n	TS	NXO/insM	NXO/swpM	PMX/insM	PMX/swpM
2	5	**7.52**	**7.52**	**7.52**	10.33	13.63
2	10	**0.60**	**0.60**	**0.60**	2.70	1.77
2	20	0.77	**0.50**	1.29	1.69	4.85
2	50	0.70	**0.69**	0.80	0.81	0.80
2	100	0.30	**0.24**	0.32	0.50	0.40
	Average	1.98	**1.91**	2.10	3.20	4.29
5	5	18.08	**18.06**	20.18	**18.06**	19.98
5	10	6.66	**6.08**	7.99	9.38	10.81
5	20	3.94	3.27	**2.90**	4.60	3.07
5	50	4.62	**1.85**	2.21	4.15	4.15
5	100	4.43	**2.19**	5.29	6.87	8.03
	Average	7.55	**6.29**	7.71	8.61	9.21
8	5	17.31	**17.26**	**17.26**	18.30	17.70
8	10	14.77	**14.61**	17.07	18.67	18.70
8	20	7.68	**5.83**	7.89	10.63	8.58
8	50	10.10	**3.47**	6.06	10.97	10.35
8	100	8.33	**3.15**	**3.15**	6.91	6.89
	Average	11.64	**8.86**	10.29	13.10	12.45
	Total average	7.05	**5.69**	6.70	8.30	8.65

The stopping criterion is the maximum CPU time allowed to solve a problem.

These two criteria are widely used in the literature (Caraffa et al., 2001; Chen, Vempati, and Aljaber, 1995; Iyer and Saxena, 2004; Reeves, 1995). As stated in Reeves (1995), the second criterion is a good choice for comparing metaheuristics since they do not have a natural stopping point. Another common stopping criterion for metaheuristics is to terminate when there is no improvement of the best solution for a specified number of generations/iterations. We did not use this criterion because the complexity and the size of the problem will affect the number of generations with no improvement, hence setting a proper number of generations is difficult.

The steps of our proposed genetic algorithm can now be described as follows:

Proposed Genetic Algorithm

Input: Parameters for genetic algorithm: *pop_size*, c, m, stopping criterion.

Parameters for scheduling problem: $n, k, m_i, p_{ij}; size_{ij}; i = 1, 2, \ldots, k$ and $i \in J$.

1. Generate initial population with *pop_size* random individuals, where each individual will be a permutation of integers from 1 to n, representing a sequence of jobs at stage 1.
2. Use Algorithm LS to decode an individual into a schedule and then calculate its corresponding $C_{\max}$ value. Evaluate the fitness of all individuals in the population by their corresponding $C_{\max}$ value.

Table 10. The best and the worst $PD_A(l)$ of different algorithms for processor setting with $m_i \sim [1; 5]$

k	n		TS	NXO/insM	NXO/swpM	PMX/insM	PMX/swpM
2	5	Best	0	0	0	0	0
		Worst	28.64	28.64	28.64	37.69	45.12
2	10	Best	0	0	0	0	0
		Worst	5.96	5.96	5.96	21.02	8.66
2	20	Best	0	0	0	0	0
		Worst	4.01	2.37	7.90	6.10	21.05
2	50	Best	0	0	0	0	0
		Worst	2.93	2.72	2.93	2.93	2.93
2	100	Best	0	0	0	0	0
		Worst	1.40	0.93	1.40	2.42	1.665
5	5	Best	2.09	2.09	4.88	2 .09	7.27
		Worst	45.48	45.48	45.48	45.48	45.48
5	10	Best	0	0	0	0	0
		Worst	10.42	8.62	19.12	22.93	24.68
5	20	Best	0	0	0	0	0
		Worst	10.32	9.62	7.17	11.77	7.37
5	50	Best	0	0	0.27	0.49	0.49
		Worst	10.65	5.10	5.10	10.66	10.66
5	100	Best	0.78	0.30	0.67	1.38	1.40
		Worst	17.31	5.42	33.37	33.37	33.37
8	5	Best	3.97	3.97	3.97	3.97	3.97
		Worst	51.43	51.43	51.43	57.40	51.43
8	10	Best	0.41	0.41	0.41	0.41	0.41
		Worst	26.18	24.43	37.21	37.21	37.21
8	20	Best	2.34	0	2.34	2.34	2.34
		Worst	28.31	26.56	26.56	26.56	26.56
8	50	Best	0.94	0.94	0.94	2.75	0.94
		Worst	20.18	5.44	24.98	24.98	24.98
8	100	Best	2.15	1.94	3.34	2.15	2.29
		Worst	28.17	7.29	18.61	18.61	18.61

The stopping criterion is the maximum CPU time allowed to solve a problem.

3. Create the mating pool from the current population by using the roulette wheel method for reproduction.
4. Select parents from the mating pool according to crossover rate c for crossover.
5. Mate these selected individuals randomly by applying the crossover operator. The crossover operators to be applied are NXO and PMX.
6. Select individuals from the mating pool according to mutation rate m for mutation.
7. Mutate these selected individuals by applying the mutation operator. The mutation operators to be applied are insM and swpM.
8. Evaluate the fitness of all new offspring obtained in steps 5 and 7 based on their $C_{\max}$ value by using Algorithm LS.

Table 11. Average number of generations/iterations performed by algorithms TS and GA[a] for processor setting with $m_i \sim [1; 5]$

k	n	TS	GA
2	5	13317884.1	6073878.7
2	10	4248106.4	1937430.6
2	20	975404.5	444851.6
2	50	122587.0	55907.8
2	100	44668.2	26371.2
5	5	6723988.4	3066604.8
5	10	1404115.9	640373.8
5	20	456146.5	208033.9
5	0	67741.0	30894.2
5	100	17778.9	2184.8
8	5	3568848.0	1627642.0
8	10	927922.3	423196.4
8	20	258968.2	118107.1
8	50	67177.0	38437.0
8	100	11679.8	15444.0

The stopping criterion is the maximum CPU time allowed to solve a problem.

[a] Since each genetic algorithm performs similar number of generations to solve a problem, we present here just the average value for four of the genetic algorithms.

9. Among all individuals of the current population and the offspring created, select the best *pop_size* as the individuals of the population in the next generation.
10. Apply steps 3–9 until the stopping criterion is met.

In order to see the effects of two crossover operators and two mutation operators, we will run four versions of the proposed genetic algorithm, one for each combination of these genetic operators. We will refer to these algorithms as NXO/insM, NXO/swpM, PMX/insM and PMX/swpM according to the genetic operators used in each one of them. Next we will describe our computational experiments to test the performance of these four versions of the proposed genetic algorithm and to compare them with the tabu search algorithm of Oğuz et al. (2004).

3. COMPUTATIONAL EXPERIMENTS

Our computational study aims to analyze the performance of four versions of the proposed genetic algorithm to minimize the makespan, as well as to investigate the effects of task numbers and processor configurations on the performance. We also compare these results with the earlier study of Oğuz et al. (2004). All the algorithms were implemented using C++ and run on a PC Pentium 4 processor-M with 2 GHz and 256 MB memory. Results are presented in terms of Average Percentage Deviation (APD) of the solution from the lower bound. APD of an algorithm A is defined as $\mathrm{APD}_A = (\sum_{l=1}^{I} \mathrm{PD}_A(l))/I$ with $\mathrm{PD}_A(l) = 100 \times \frac{C^A_{\max}(l) - LB(l)}{LB(l)}$, where $C^A_{\max}(l)$ denotes the

Table 12. APD of different algorithms for processor setting with $m_i = 5$

k	n	TS	NXO/insM	NXO/swpM	PMX/insM	PMX/swpM
2	5	**18.78**	**18.78**	**18.78**	18.95	**18.78**
2	10	**4.61**	**4.61**	**4.61**	**4.61**	**4.61**
2	20	2.63	2.05	2.31	2.57	**1.84**
2	50	2.12	1.71	1.92	2.07	**1.48**
2	100	2.13	1.72	1.93	2.08	**1.49**
	Average	6.05	5.77	5.91	6.06	**5.64**
5	5	23.03	**15.77**	**15.77**	22.51	16.12
5	10	9.83	6.89	6.89	9.61	**6.88**
5	20	11.45	**7.98**	**7.98**	11.19	8.01
5	50	20.32	18.95	15.98	19.86	**14.22**
5	100	29.47	**20.10**	21.78	28.80	20.63
	Average	18.82	13.94	13.68	18.39	**13.17**
8	5	12.60	11.10	**10.02**	12.31	11.82
8	10	18.51	14.73	14.59	18.09	**12.96**
8	20	22.88	**17.67**	17.96	22.36	18.02
8	50	27.58	20.83	20.83	26.95	**19.30**
8	100	32.42	25.09	25.09	31.69	**22.70**
	Average	22.80	17.89	17.70	22.28	**16.96**
	Total average	15.89	12.53	12.43	15.58	**11.92**

The stopping criterion is the maximum CPU time allowed to solve a problem.

makespan value obtained by algorithm A for instance l, LB(l) denotes the lower bound on the optimal makespan value for instance l, and I denotes the number of instances considered. In our experiments A will stand for four versions of the proposed genetic algorithm, namely NXO/insM, NXO/swpM, PMX/insM and PMX/swpM, and for the tabu search algorithm of Oğuz et al. (2004), namely TS.

3.1. Data generation

All data for the computational experiments were generated randomly in line with the previous study of Oğuz et al. (2004). We form a database from these randomly generated problem instances to provide a common basis for further studies and the data can be obtained from the authors. For the processor configurations, we considered two different settings to see the effects of the processor configuration. In the first setting m_i, that is the number of processors at stage i, $i = 1, 2, \ldots, k$, is randomly generated from the range [1; 5], whereas in the second setting $m_i = 5$ is considered for all stages. The processor requirements, $size_{ij}$, and the processing times, p_{ij}, were generated randomly from the ranges of [1; m_i] and [1;100], respectively, for $i = 1, 2, \ldots, k$ and $j \in J$. For each combination of n and k, 10 problem instances were generated ($n = 5, 10, 20, 50, 100$, $k = 2, 5, 8$). This resulted in 300 problems, each to be run 10 times due to the combination of five algorithms and two stopping criteria. Hence, in total, we performed 3000 runs in our computational experiments.

Table 13. The best and the worst $PD_A(l)$ of different algorithms for processor setting with $m_i = 5$

k	n		TS	NXO/insM	NXO/swpM	PMX/insM	PMX/swpM
2	5	Best	0	0	0	0	0
		Worst	28.98	28.98	28.98	28.98	28.98
2	10	Best	0	0	0	0	0
		Worst	12.03	12.03	12.03	12.03	12.03
2	20	Best	0	0	0	0	0
		Worst	10.41	10.41	10.41	12.39	10.41
2	50	Best	0	0	0	0	0
		Worst	9.82	8.83	9.51	10.02	8.83
2	100	Best	0	0	0	0.11	0
		Worst	7.64	6.02	7.23	7.87	6.03
5	5	Best	9.45	7.03	12.01	11.02	10.35
		Worst	56.83	49.04	56.83	67.63	50.01
5	10	Best	4.82	2.09	3.17	3.17	2.12
		Worst	34.18	17.08	19.33	22.18	16.02
5	20	Best	7.09	3.23	4.56	7.19	4.56
		Worst	19.04	14.91	15.03	21.12	13.98
5	50	Best	10.09	10.93	9.87	10.84	10.03
		Worst	29.01	29.12	18.93	18.93	19.74
5	100	Best	13.89	11.09	10.45	12.09	10.87
		Worst	56.96	49.02	45.04	48.48	56.84
8	5	Best	8.93	7.04	6.76	8.14	6.76
		Worst	33.76	34.76	36.31	38.01	35.56
8	10	Best	12.67	9.62	8.89	9.45	9.06
		Worst	39.95	45.93	43.03	36.95	38.09
8	20	Best	16.04	12.80	11.38	12.93	10.09
		Worst	56.08	45.18	34.86	63.09	52.45
8	50	Best	15.12	10.87	10.19	12.05	11.76
		Worst	46.91	43.29	39.34	40.12	35.19
8	100	Best	16.10	9.83	10.15	12.19	10.92
		Worst	52.12	39.66	40.00	53.13	35.22

The stopping criterion is the maximum CPU time allowed to solve a problem.

3.2. *Lower bound*

In order to compare the performance of the proposed genetic algorithm, we use a lower bound on the optimum makespan value since we can obtain the optimal solution only for small size problem instances. In our experiments we use the following proposed lower bound, which is the maximum of a stage-based and a job-based lower bound.

Lower bound: $\mathrm{LB} = \max\{\mathrm{LB}_1, \mathrm{LB}_2\}$

Stage-based: $\mathrm{LB}_1 = \max_{i=1,\ldots,k} \mathrm{LB}_1(i)$, where for $i = 1, \ldots, k$:

$$\mathrm{LB}_1(i) = \min_{j \in J} \left\{ \sum_{l=1}^{i-1} p_{lj} \right\} + \max\{M_1(i), M_2(i)\} + \min_{j \in J} \left\{ \sum_{l=i+1}^{k} p_{lj} \right\},$$

Table 14. Average number of generations/iterations performed by algorithms TS and GA[a] for processor setting with $m_i = 5$

k	n	TS	GA
2	5	11113065.6	5068290.8
2	10	3544771.4	1616623.4
2	20	813858.8	371137.9
2	50	102223.5	46582.9
2	100	37204.0	21936.1
5	5	5610776.1	2558865.1
5	10	1171597.9	534291.8
5	20	380562.9	173524.9
5	50	56457.2	25710.3
5	100	14766.2	1753.7
8	5	2977962.9	1358119.7
8	10	774237.2	353067.6
8	20	216027.1	98485.3
8	50	55986.5	32004.4
8	100	9676.8	12817.8

The stopping criterion is the maximum CPU time allowed to solve a problem.
[a] Since each genetic algorithm performs similar number of generations to solve a problem, we present here just the average value for four of the genetic algorithms.

and

$$M_1(i) = \left\lceil \frac{1}{m_i} \sum_{j \in J} (p_{ij} size_{ij}) \right\rceil, \quad M_2(i) = \sum_{j \in A_i} p_{ij} + \left\lceil \frac{1}{2} \sum_{j \in B_i} p_{ij} \right\rceil$$

with

$$A_i = \left\{ j \mid size_{ij} > \frac{m_i}{2} \right\}, \quad B_i = \left\{ j \mid size_{ij} = \frac{m_i}{2} \right\}.$$

Job-based:

$$\mathrm{LB}_2 = \max_{j \in J} \left\{ \sum_{i=1}^{k} p_{ij} \right\}.$$

The above stage-based lower bound determines the maximum over all stages of the lower bound $\mathrm{LB}_1(i)$ corresponding to stage i, $i = 1, 2, \ldots, k$. This lower bound $\mathrm{LB}_1(i)$ for stage i, $i = 1, 2, \ldots, k$, is obtained by considering a single-stage multiprocessor task problem, that is $P|size_{ij}|C_{\max}$ problem, and then by adding a head and a tail value as a lower bound for the preceding stages and the succeeding stages, respectively. A lower bound for $P|size_{ij}|C_{\max}$ problem can be obtained in two different ways. First, we can consider the preemptive version of the problem and the lower bound will be similar to that of a $P\|C_{\max}$ problem. Namely, the preemptive lower bound for stage

i will be

$$M_1(i) = \left\lceil \frac{1}{m_i} \sum_{j \in J} (p_{ij} size_{ij}) \right\rceil.$$

Second, we can use the observation that two multiprocessor tasks with processor requirements strictly greater than the half of the number of processors at a stage cannot be processed simultaneously. Hence the sum of the processing times of such multiprocessor tasks will form a lower bound for that stage. We can further tighten this lower bound by adding the preemptive lower bound for only those multiprocessor tasks with a processor requirement of exactly half of the number of processors at that stage (this value will be zero if the number of processors at a stage is an odd number). This is true because as an extension of the above observation, a multiprocessor task with a processor requirement strictly greater than the half of the number of processors at a stage cannot be processed simultaneously with a multiprocessor task with a processor requirement of exactly half of the number of processors at that stage. However, since a multiprocessor task with a processor requirement strictly smaller than the half of the number of processors at a stage can be processed simultaneously with any other multiprocessor tasks, we cannot add them to the preemptive lower bound here. Hence, we have the second lower bound for stage i as follows:

$$M_2(i) = \sum_{j \in A_i} p_{ij} + \left\lceil \frac{1}{2} \sum_{j \in B_i} p_{ij} \right\rceil$$

with

$$A_i = \left\{ j \mid size_{ij} > \frac{m_i}{2} \right\}, \quad B_i = \left\{ j \mid size_{ij} = \frac{m_i}{2} \right\}.$$

The head value as a lower bound of the preceding stages can be obtained by simply taking the minimum over the job set J of the sum of processing times of job $j \in J$ from stage 1 to stage $i-1$, which is $\min_{j \in J}\{\sum_{l=1}^{i-1} p_{lj}\}$. Similarly, the tail value as a lower bound of the succeeding stages can be obtained as $\min_{j \in J}\{\sum_{l=i+1}^{k} p_{lj}\}$. It should be noted that these head and tail values cannot be tightened as in Santos, Hunsucker, and Deal (1995) since the argument used there is not valid for hybrid flow-shops with multiprocessor tasks.

The job-based lower bound presented above is similar to those used in other flow-shop scheduling problems in the sense that it considers the maximum over the job set J of the sum of processing times of job $j \in J$ from the first stage to the last stage.

Hence $\mathrm{LB} = \max\{\mathrm{LB}_1, \mathrm{LB}_2\}$ is the lower bound on the optimum makespan value.

3.3. Preliminary computational experiments

The efficiency of a genetic algorithm depends on the selection of control parameters used, such as population size, crossover rate and mutation rate. Hence, before presenting and discussing the results of the performance of four versions of the proposed genetic algorithm, we first analyze the parameter setting for each one of them.

In our preliminary computational experiments, we performed a $2 \times 3 \times 3$ factorial design of experiments involving three parameters (factors) with the following values. We considered two population size values, $pop\_size = 50$ and 100, representing a small population and a large

population, respectively. We further took three values both for the crossover rate, c, and for the mutation rate, m. We have $c = 1.0, 0.8, 0.6$ and $m = 0.001, 0.01, 0.1$. These values are concluded to be reasonable for $Fk(Pm_1, \ldots, Pm_k)|size_{ij}|C_{\max}$ problem considering the genetic algorithm studies reported in the literature for flow-shop environments (Caraffa et al., 2001; Chen, Vempati, and Aljaber, 1995; Iyer and Saxena, 2004; Reeves, 1995).

Combining these parameters results in 18 different test runs for each version of the proposed genetic algorithm. We consider one particular case of $Fk(Pm_1, \ldots, Pm_k)|size_{ij}|C_{\max}$, where $n = 50$ and $k = 5$, to choose the best combination of above parameters for each version of the proposed genetic algorithm. For each of 18 test runs, we generated 10 random instances. We further created five random initial populations to be used by each of 10 instances. These five randomly generated initial populations were kept unchanged for each of 10 instances for each version of the proposed genetic algorithm to eliminate the bias of the starting population. We refer to an instance with a certain initial population as a replica of that instance. Hence we run each version of the proposed genetic algorithm for 18 different combinations of the parameter values by using 10 random instances which are replicated five times. This resulted in a total of 3600 runs in our preliminary computational experiments.

We used maximum CPU time allowed to solve a problem as the stopping criterion during these preliminary computational experiments and we set a limit of 30 min CPU time. We then analyzed the APD values for each test run to determine the best combination of parameters for each version of the proposed genetic algorithm. We chose the parameter combination that gives the smallest APD value for each version of the proposed genetic algorithm. These preliminary computational experiments with the above settings resulted in the following best combinations of the parameters, which are given in the order of population size, crossover rate and mutation rate, for each version of the proposed genetic algorithm:

	insM	swpM
NXO	100, 0.8, 0.1	100, 0.6, 0.1
PMX	100, 1.0, 0.1	100, 0.6, 0.01

Hence we use the above parameter combinations for the respective versions of the proposed genetic algorithm in our computational experiments presented in Section 3.4.

3.4. Computational results

After setting the best combination of the control parameters, that is population size, crossover rate and mutation rate, for four versions of our proposed genetic algorithm, we employed an extensive computational experiment for the following reasons:

1. To test the effects of different crossover and mutation operators on the performance of the genetic algorithm. To this effect, we used throughout our experiments four versions of the proposed genetic algorithm, namely NXO/insM, NXO/swpM, PMX/insM and PMX/swpM, as described previously in Section 3, and compared their performance with each other.
2. To compare the performance of these four versions of the proposed genetic algorithms with that of the tabu search algorithm of Oğuz et al. (2004), denoted by TS. These results are presented in Tables 3 through 14. In the tables regarding the APD results, the best result corresponding to a combination of k and n is given in bold.

3. To test the effects of different stopping criteria on the performance of the algorithms (both the genetic algorithm and the tabu search). To this effect, we run all algorithms under two different stopping criteria: the maximum number of generations/iterations performed and the maximum CPU time allowed to solve a problem. We note that, in the first stopping criterion, when we consider TS algorithm, we need to set the stopping criterion in terms of the iterations since generations are meaningless for that algorithm. The results of the experiments for the maximum number of generations/iterations performed as the stopping criterion are given in Tables 3 through 8. On the other hand, Tables 9 through 14 present the results of the experiments when the maximum CPU time allowed to solve a problem is the stopping criterion.
4. To test the effects of the processor settings on $Fk(Pm_1, \ldots, Pm_k)|size_{ij}|C_{\max}$ problem. To this effect, we perform our experiments for two different processor settings as explained in Section 3.1:

- Variable number of processors at each stage, that is $m_i \sim [1; 5]$, and
- Fixed number of processors at each stage, that is $m_i = 5$,
  for $i = 1, 2, \ldots, k$. The results of the experiments for variable number of processors at each stage are given in Tables 3–5 and 9–11. Tables 6–8 and 12–14, on the other hand, present the results of the experiments for fixed number of processors at each stage.

From the results presented in Tables 3 through 14, the following observations can be made about the $Fk(Pm_1, \ldots, Pm_k)|size_{ij}|C_{\max}$ problem:

1. Comparison of four versions of the proposed genetic algorithm: When we compare four versions of the proposed genetic algorithms with each other, except for the case where we have the fixed number of processors at each stage, that is $m_i = 5$, and the maximum CPU time allowed to solve a problem as the stopping criterion, we observe that NXO/insM outperforms all other versions of the proposed genetic algorithm in terms of the APD values. In the above exception case, PMX/swpM performs better than NXO/insM in half of the k and n combinations. From these, we can conclude that the proposed genetic algorithm performs better when we use NXO as the crossover operator and insertion as the mutation operator with $c = 0.8$, $m = 0.1$ and $pop\_size = 100$. However, we caution the reader that both the parameters and the genetic operators applied can be problem specific even though they were determined after extensive computational experiments.
2. Comparison of genetic algorithm with the tabu search algorithm: Tables 3, 6, 9 and 12 indicate that NXO/insM is the best among five algorithms in terms of the APD produced. NXO/insM performs better than the other algorithms in most of the cases and for those cases in which another algorithm is the best, APD of NXO/insM is not far from that of the best algorithm.
3. Effects of different stopping criteria: The results given in Tables 3, 6, 9 and 12 also reveal that when we use the stopping criterion of the maximum CPU time allowed to solve a problem, the performance of each algorithm in terms of APD gets much better, which is an expected result. We see the most substantial improvement in TS algorithm, however it still does not perform better than NXO/insM algorithm.
4. Effects of k and n: In all cases, APD values decrease as the number of jobs, n, increases for a given k. This is an expected result due to the increasing values of the makespan. On the other hand, APD values increase as the number of stages, k, increases for a given n. This is again

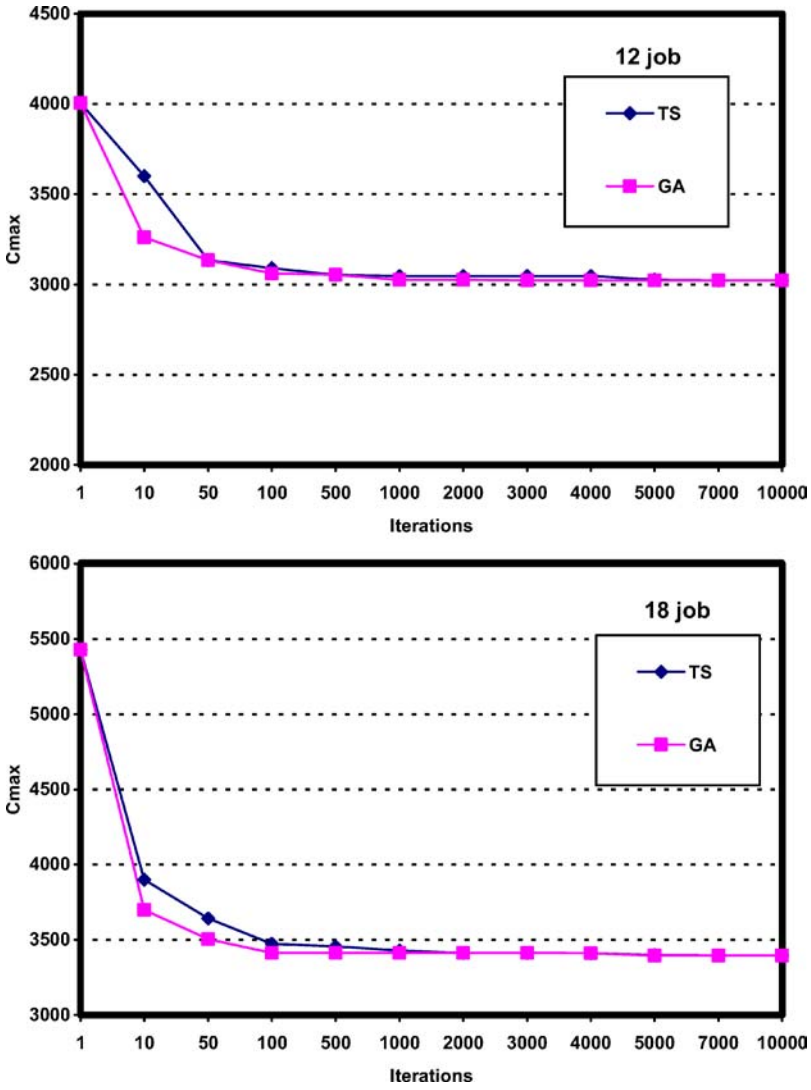

Figure 2. Convergence of the genetic and the tabu search algorithms for the machine-vision problem for $n = 12, 18$.

an expected result. First, the lower bound used becomes less effective and hence distant from the optimal criterion value as the number of stages, k, and the number of processors at stages, m_i, increase. We attribute the lessening effect of the lower bound to the head and tail values used in our stage-based lower bound, which are not very tight. Second, the performance of Algorithm LS worsens as the number of stages, k, and the number of processors at stages, m_i, increase, which will affect the performance of the proposed genetic algorithm.

5. Effects of the processor settings: It appears that when the number of processors at each stage is fixed, that is $m_i = 5, i = 1, 2, \ldots, k$, $Fk(Pm_1, \ldots, Pm_k)|size_{ij}|C_{\max}$ problem is more

Table 15. Percentage deviation and CPU time of different algorithms for machine-vision problem; $n = 8, 10$

k	n	Problem number	Percentage deviation of					CPU time of	
			TS	NXO/insM	NXO/swpM	PMX/insM	PMX/swpM	TS	GA
2	8	1	0.25	0.25	0.25	0.25	0.25	0.70	0.93
		2	0.82	0.82	0.82	0.82	0.82	0.68	0.95
		3	3.91	3.91	3.91	3.91	3.91	0.71	0.98
		4	7.71	7.71	7.71	7.71	7.71	0.69	0.85
		5	0.06	0.06	0.06	0.06	0.06	0.76	0.97
		6	0.06	0.06	0.06	0.06	0.06	0.77	0.95
		7	0.10	0.10	0.10	0.10	0.10	0.77	0.98
		8	0.48	0.36	0.36	0.36	0.36	0.66	0.84
		9	0.11	0.11	0.11	0.11	0.11	0.80	0.99
		10	2.63	2.63	2.63	2.63	2.63	0.73	0.97
		Average	1.61	1.60	1.60	1.60	1.60	0.73	0.94
2	10	1	1.51	1.51	1.51	1.51	1.51	1.00	1.40
		2	1.39	1.39	1.39	1.39	1.39	1.02	1.28
		3	3.77	3.77	3.77	3.77	3.77	0.90	1.33
		4	0.34	0.11	0.11	0.11	0.11	0.98	1.21
		5	1.22	0.52	0.52	0.52	0.52	1.08	1.22
		6	0.00	0.00	0.00	0.00	0.00	1.34	1.42
		7	0.00	0.00	0.00	0.00	0.00	1.00	1.23
		8	2.95	2.95	2.95	2.95	2.95	1.23	1.62
		9	6.83	6.78	6.78	6.78	6.78	0.99	1.19
		10	1.61	1.45	1.45	1.45	1.45	1.10	1.34
		Average	1.96	1.85	1.85	1.85	1.85	1.07	1.32

difficult to solve. This result can be attributed to the fact that the effectiveness of both the lower bound and Algorithm LS lessens as the number of processors at each stage, m_i, increases as explained above. We further observe that when the number of processors at each stage is not fixed, there might exist some bottleneck machines which restrict the number of feasible schedules and hence the search space.

6. Apart from the APD, we also analyzed the best and the worst percentage deviation of different algorithms. We observe that similar conclusions as above can be drawn for both the best and the worst percentage deviation results. The results indicate that in general the proposed genetic algorithm with NXO and insertion performs much better than the others in terms of the quality of the solutions achieved.

Last, we run four versions of the proposed genetic algorithm as well as the tabu search algorithm on the data obtained for the machine-vision systems. The data are same as in Oğuz et al. (2003) and can be obtained from the authors. The number of jobs are 8, 10, 12 and 18 in a two-stage system. We first analyzed the convergence of the algorithms. As it can be seen from Figure 2, the proposed genetic algorithm converges much faster than the tabu search, hence it needs less

Table 16. Percentage deviation and CPU time of different algorithms for machine-vision problem; $n = 12, 18$

k	n	Problem number	Percentage deviation of					CPU time of	
			TS	NXO/insM	NXO/swpM	PMX/insM	PMX/swpM	TS	GA
2	12	1	6.83	2.11	2.11	2.11	2.11	1.36	1.83
		2	1.26	1.12	1.30	1.30	1.30	1.39	1.85
		3	0.04	0.04	0.04	0.04	0.04	1.99	2.30
		4	1.48	4.19	4.19	4.19	4.19	1.55	1.87
		5	0.08	0.08	0.32	0.32	0.20	1.46	1.82
		6	0.04	0.04	0.04	0.04	0.04	1.42	1.86
		7	0.13	0.13	0.13	0.13	0.13	1.63	1.92
		8	0.19	0.98	1.57	1.01	1.67	1.51	1.91
		9	1.06	1.06	1.06	1.06	1.06	1.52	1.90
		10	0.17	0.09	0.09	0.09	0.09	1.32	1.83
		Average	1.13	0.98	1.08	1.03	1.08	1.52	1.91
2	18	1	0.00	0.00	0.00	0.00	0.00	2.68	3.05
		2	0.33	3.29	3.53	3.53	3.40	2.63	2.99
		3	0.17	0.64	0.64	0.64	0.64	2.72	3.12
		4	1.97	3.26	3.26	3.26	3.26	2.75	3.07
		5	1.03	0.75	0.75	0.75	0.75	2.76	3.10
		6	1.30	2.59	2.59	2.66	2.59	2.67	2.98
		7	3.56	2.22	2.22	2.22	2.22	2.47	2.86
		8	0.57	0.09	0.22	0.09	0.09	3.02	3.24
		9	3.76	7.25	7.25	7.25	7.25	3.07	3.25
		10	0.22	0.67	0.67	0.67	0.67	3.07	3.25
		Average	1.29	2.08	2.11	2.11	2.09	2.79	3.09

CPU time to achieve the solution. Nevertheless, since both the genetic algorithm and the tabu search algorithm converge quite early, we set the generation/iteration number for this set of runs as 1000 to compare the algorithms. The computational results are summarized in Tables 15 and 16. We observe that the genetic algorithm with NXO as the crossover operator and insertion as the mutation operator performs better than the others in terms of the percentage deviation of the $C_{\max}$ value from the lower bound in most of the cases. Tabu search algorithm performs better than the genetic algorithm only for the 18-job instances. When we analyze the CPU time spent by each algorithm, we see that even though the tabu search algorithm is more efficient, algorithms are comparable overall. We note that the maximum CPU time spent among all application data by the proposed genetic algorithm and by the tabu search algorithm is 3.25 s and 3.07 s, respectively, which is acceptable for the machine-vision systems. Furthermore, since the genetic algorithm converges around 500 iterations, the CPU time spent by the genetic algorithm will be much less than the reported ones for the given percentage deviations. Hence, we conclude that the proposed genetic algorithm is a promising solution tool to be used for machine-vision systems in practice.

4. CONCLUSIONS

In this paper we propose a genetic algorithm for $Fk(Pm_1, \ldots, Pm_k)|size_{ij}|C_{\max}$ problem and describe its implementation. We propose a new crossover operator to be used in the genetic algorithm and compare it with PMX crossover. We employed a preliminary test to establish the best combination of the control parameters to be used along with different genetic operators. Then we carried out an extensive computational experiment to test the performance of the genetic algorithm with different genetic operators. The results show that the genetic algorithm performs better in terms of the percentage deviation of the solution from the lower bound value when new crossover operator is used along with the insertion mutation. The genetic algorithm utilizing this structure outperforms other versions of the proposed genetic algorithm with different mutation and crossover operators.

The computational experiments also involved a comparison of our proposed genetic algorithm with the tabu search algorithm from the literature developed for the same problem. Again our results revealed that even though the performance of the tabu search algorithm is quite close to that of our genetic algorithm, it appears that genetic algorithm is a better approach for $Fk(Pm_1, \ldots, Pm_k)|size_{ij}|C_{\max}$ problem. From these, we can conclude that NXO/insM genetic algorithm is an effective approach for $Fk(Pm_1, \ldots, Pm_k)|size_{ij}|C_{\max}$ problem.

We also analyed the effects of the number of stages, the number of processors at each stage as well as the number of jobs on $Fk(Pm_1, \ldots, Pm_k)|size_{ij}|C_{\max}$ problem. However, our results indicate that the observations we can make are mostly related to the effectiveness of the lower bound used. It is apparent that there is a need for a stronger lower bound or for an exact algorithm. Hence for further research we are considering to improve the lower bound proposed and also to develop an exact algorithm for $Fk(Pm_1, \ldots, Pm_k)|size_{ij}|C_{\max}$ problem.

ACKNOWLEDGMENTS

The authors are grateful to anonymous reviewers for their helpful comments and suggestions. The work described in this paper was partially supported by a grant from The Hong Kong Polytechnic University (Project No. G-T247) for the first author.

REFERENCES

Błażewicz, J., M. Drabowski, and J. Węglarz, "Scheduling multiprocessor tasks to minimize schedule length," *IEEE Transactions on Computers*, **C-35**(5), 389–393 (1986).

Błażewicz, J., M. Drozdowski, G. Schmidt, and D. de Werra, "Scheduling independent two-processor tasks on a uniform duo-processor system," *Discrete Applied Mathematics*, **28**, 11–20 (1990).

Błażewicz, J., K. H. Ecker, E. Pesch, G. Schmidt, and J. Węglarz," *Scheduling Computer and Manufacturing Processes*, 2nd edition, Springer-Verlag, Berlin, 2001.

Brucker, P., S. Knust, D. Roper, and Y. Zinder, "Scheduling UET tasks systems with concurrency on two parallel identical processors," *Mathematical Methods of Operations Research*, **52**(3), 369–387 (2000).

Brucker, P. and A. Krämer, "Shop scheduling problems with multiprocessor tasks on dedicated processors," *Annals of Operations Research*, **57**, 13–27 (1995).

Brucker, P. and A. Krämer, "Polynomial algorithms for resource-constrained and multiprocessor task scheduling problems," *European Journal of Operational Research*, **90**, 214–226 (1996).

Caraffa, V., S. Ianes, T. P. Bagchi, and C. Sriskandarajah, "Minimizing makespan in a blocking flowshop using genetic algorithms," *International Journal Production Economics*, **70**, 101–115 (2001).

Chen, J. and C.-Y. Lee, "General multiprocessor task schduling," *Naval Research Logistics*, **46**, 57–74 (1999).

Chen, C.-L., V. S. Vempati, and N. Aljaber, "An application of genetic algorithms for flow shop problems," *European Journal of Operational Research*, **80**, 389–396 (1995).

Della Croce, F., R. Tadei, and G. Volta, "A genetic algorithm for the job shop problem," *Computers and Operations Research*, **22**, 15–24 (1995).

Dorndorf, U. and E. Pesch, "Evolution based learning in a job shop scheduling environment," *Computers and Operations Research*, **22**, 25–40 (1995).

Drozdowski, M., "Scheduling multiprocessor tasks—An overview," *European Journal of Operational Research*, **94**, 215–230 (1996).

Drozdowski, M., "Selected problems of scheduling tasks in multiprocessor computer systems," Monograph No. 321, Poznan University of Technology Press, 1997. (Downloadable from website http://www.cs.put.poznan.pl/mdrozdowski#prepr).

Du, J. and J. Y.-T. Leung, "Complexity of scheduling parallel task systems," *SIAM Journal on Discrete Mathematics*, **2**, 473–487 (1989).

Ercan, M. F. and Y.-F. Fung, "Real-time image interpretation on a multi-layer architecture," in *Proceedings of the IEEE TENCON'99*, 1999, pp. 1303–1306.

Gen, M. and R. Cheng, *Genetic Algorithms and Engineering Design*, Wiley, New York, 1997.

Goldberg, D. and R. Lingle, "Alleles, loci, and the traveling salesman problem," in J. Grefenstette (ed.), *Proceedings of the First International Conference on Genetic Algorithms and their Applications*, Hillsdale, 1985, pp. 154–159.

Gupta, J. N. D., "Two stage hybrid flowshop scheduling problem," *Journal of Operational Research Society*, **39**(4), 359–364 (1988).

Haouari, M. and R. M'Hallah, "Heuristic algorithms for the two-stage hybrid flowshop problems," *Operations Research Letters*, **21**, 43–53 (1997).

Holland, J. H., *Adaption in Natural and Artificial Systems*, University of Michigan Press, Ann Arbor, 1975.

Iyer, S. K. and B. Saxena, "Improved genetic algorithm for the permutation flowshop scheduling problem," *Computers and Operations Research*, **31**, 593–606 (2004).

Krawczyk, H. and M. Kubale, "An approximation algorithm for diagnostic test scheduling in multicomputer systems," *IEEE Transactions on Computers*, **34**, 869–872 (1985).

Lee, C.-Y. and X. Cai, "Scheduling one and two-processor tasks on two parallel processors," *IIE Transaction*, **31**, 445–455 (1999).

Lee, C.-Y., L. Lei, and M. Pinedo, "Current trends in deterministic scheduling," *Annals of Operations Research*, **70**, 1–41 (1997).

Lloyd, E. L., "Concurrent task systems," *Operations Research*, **29**, 189–201 (1981).

Michalewicz, Z., *Genetic Algorithms+Data Structures=Evolution Programs*, 3rd edition, Springer-Verlag, Berlin, 1996.

Moursli O. and Y. Pochet, "A branch-and-bound algorithm for the hybrid flowshop," *International Journal of Production Economics*, **64**, 113–125 (2000).

Oğuz, C. and M. F. Ercan, "Scheduling multiprocessor tasks in a two-stage flow-shop environment," *Computers and Industrial Engineering*, **33**, 269–272 (1997).

Oğuz, C., M. F. Ercan, T. C. E. Cheng, and Y.-F. Fung, "Heuristic algorithms for multiprocessor task scheduling in a two-stage hybrid flow-shop," *European Journal of Operational Research*, **149**, 390–403 (2003).

Oğuz, C., Y. Zinder, V. H. Do, A. Janiak, and M. Lichtenstein, "Hybrid flow-shop scheduling problems with multiprocessor task systems," *European Journal of Operational Research*, **152**, 115–131 (2004).

Portmann, M.-C., A. Vignier, D. Dardilhac, and D. Dezalay, "Branch and bound crossed with GA to solve hybrid flow shops," *European Journal of Operational Research*, **107**, 389–400 (1998).

Reeves, C. R., "A genetic algorithm for flowshop sequencing," *Computers and Operations Research*, **22**, 5–13 (1995).

Riane, F., A. Artiba, and S. E. Elmaghraby, "A hybrid three-stage flowshop problem: Efficient heuristics to minimize makespan," *European Journal of Operational Research*, **109**, 321–329 (1998).

Santos, D. L., J. L. Hunsucker, and D. E. Deal, "Global lower bounds for flow shops with multiple processors," *European Journal of Operational Research*, **80**, 112–120 (1995).

Vignier, A., J.-C. Billaut, and C. Proust, "Hybrid flowshop scheduling problems: State of the art," *Rairo-Recherche Operationnelle-Operations Research*, **33**, 117–183 (1999).