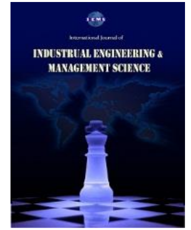
International Journal of Industrial Engineering & Management Science

Journal homepage: www.ijiems.com



A memetic algorithm for multistage hybrid flow shop scheduling problem with multiprocessor tasks to minimize makespan

Hadi Gholami[1], Mohammad Taghi Rezvan[2*]

[1] Department of Computer Engineering, Ayatollah Amoli Branch, Islamic Azad University, Amol, Iran.

[2] Department of Industrial Engineering, Faculty of Engineering, University of Kashan, 6 Km, Ghotbravandi Blvd, Kashan 8731753153, Iran.

***Keywords*:**

1 Multistage hybrid flow shop
2 Multiprocessor tasks
3 Memetic algorithm
4 Makspan

A B S T R A C T

This paper presents an effective memetic algorithm (MA) for a hybrid flow shop scheduling problem with multiprocessor tasks (HFSMT) to minimize the makespan. The problem is modeled as deterministic by a mixed graph. This problem has at least two production stages, each of which has several machines, operating in parallel. Two sub-problems are considered for solving this problem: determining the sequence of jobs in the first stage and reducing the idle time of the processors in the next stages. The developed algorithm uses an operator called Bad Selection Operator. This operator holds the worst chromosomes of each generation and uses them to search in the space of other problems at predetermined timescales. Besides, this algorithm uses a dynamic adjustment structure to improve the ratio of crossover and mutation operators that could reduce the execution time. The efficiency of the proposed MA is investigated by testing it on well-known benchmark instances and also compared with other algorithms introduced in the literature. Computational results show that this algorithm has a good performance on problems which have more than two stages.

* Corresponding author, Email: rezvan@kashanu.ac.ir

1 Introduction

In manufacturing environments, the scheduling consists in assigning of limited resources to a number of jobs required by the production processes. In these problems, some performance goals such as makespan, tardiness, and lateness can be optimized (Pinedo, 2005). One of the most well-known scheduling problems is flow shop problem in which n jobs are scheduled in the sequential stage k, where each stage consists of only one machine (processor) (Abouei Ardakan et al., 2013).

An extension of the flow shop problem is called hybrid flow shop scheduling problem (HFSP) or flexible flow shop scheduling one (FFSP) which combines the flow shop and the parallel machine scheduling problem. In HFSP, a set of jobs flow in the same order through multiple stages, where each stage contains one or more machines in parallel (Ruiz and Vázquez-Rodríguez, 2010).

The literature survey shows that current available research on HFSPs considered these problems at three categories: two stages, three stages and multistage (m-stage, where m is more than 3). Some stages may have only one machine, but at least one stage must have multiple machines. There are exact algorithms such as branch and bound (B&B) (Ruiz and Vázquez-Rodríguez, 2010), approximation ones such as constructive heuristics based on rules (Gupta and Tunc, 1994; Guinet et al., 1996; Riane et al., 1998) or iterative heuristics based on meta-heuristics such as genetic algorithm (GA) (Oğuz and Ercan, 2005; Jin et al. 2002; Besbes et al. 2010; Bathrinath et al. 2016), harmony search (HS) (Bathrinath et al., 2016), particle swarm optimization (PSO) (Bathrinath et al., 2016; Geem et al., 2001; Singh et al., 2016), artificial bee colony (ABC) (Peng et al., 2018), tabu search (TS) algorithm (Chen et al., 2007), and memetic algorithm (MA) (Tavakkoli-Moghaddam et al., 2009), to solve HFSP.

The hybrid flow shop scheduling with multiprocessor task (HFSMT) is a relatively new type of HFSP, where, at each stage, each job needs one or more processors simultaneously. This problem is extremely difficult in combinatorial optimization which is considered as an NP-hard problem (Lenstra et al., 1977). Various applications of HFSMT are in several industrial settings such as airplane engine production, electronics industries, textile manufacturing, petrochemical industry, distributed systems, real-time machine vision systems and parallel computing (Behnamian and Zandieh, 2011).

There are solution approaches based on heuristics to solve the HFSMT problem. Oğuz et al. (2003) proposed heuristic algorithms which considered simple priority rules for sequencing the tasks. They derived some lower bounds to measure the performance analysis of the heuristic algorithms. Ying (2009) presented a simple and effective iterated greedy (IG) heuristic which is an effective stochastic local search algorithm. This algorithm consists of two main phases, called destruction and construction. Some elements of a current candidate solution are eliminated in the destruction phase and the construction phase, and other elements are inserted into a partial solution until a complete solution is reconstructed again. Kahraman et al. (2010) proposed an effective parallel greedy algorithm (PGA) which applies two phases: destruction and construction. Lahimer et al. (2013) presented a discrepancy search (DS) heuristic that is based on the new concept of adjacent discrepancies and describes a new lower bound based on the concept of dual feasible functions.

There are meta-heuristics to solve the HFSMT problem. Ying and Lin (2007) proposed an ant colony system (ACS) approach based on the notion of corresponding the heuristic desirability to the jobs ranking indexes of the selected constructive heuristic at the first stage. Hence, the selected constrictive heuristic was chosen from five of constructive heuristics to optimize the

makespan. Kurdi (2019) proposed an ACS with a novel Non-Daemon Actions procedure for this problem with the objective of makespan minimization. Oguz and Ercan (2005) introduced a new crossover operator for their proposed GA to solve the HFSMT problem by providing the new crossover operator which minimizes the idle time on the processors. Serifoglu and Ulusoy (2004) proposed a GA and showed that the best chromosome of each population does not lead to any statistically significant improvement. Engin et al. (2011) proposed a new mutation operator for their proposed GA to improve available solutions by refining the most promising individuals of each generation. Wang et al. (2011) presented a neighborhood generation mechanism in simulated annealing (SA). They also provided a method for reducing processor's idle time in stages after stage #1. Tseng and Liao (2008) proposed PSO which is debatable in terms of four features: determining the absolute position of the job j for the particle i in the encoding phase, providing three velocity equations in update of velocity, using three topologies for neighborhood and incorporating a local search scheme based on reduced variable neighborhood search (RVNS). Chou (2013) used PSO in which a new lower bound has been presented. They also used five algorithms in stages after stage #1. Lin et al. (2013) developed a hybrid artificial bee colony (HABC) algorithm with bi-directional planning. In this research, an SA-based rule was used to avoid local optimization. Xu et al. (2013) presented an effective immune algorithm (IA) that used the dispatching rules to decide the order of job processing and machine assignment in stages after stage #1 and also used rules to reduce useless searches to find optimal localization. Jouglet et al. (2009) combined B&B algorithm with GA to improve the quality of the provided solution so that B&B improves an offspring and acts as a local search. Ying (2012) proposed a new hybrid immune algorithm based on the features of artificial immune systems and iterated greedy algorithms to minimize the makespan of this problem. Rani and Zoraida (2016) tried to reduce the makespan value in the HFSMT problem using a discrete firefly algorithm (FA). Ying and Lin (2018) also used a self-tuning IG to minimize the makespan in distributed HFSMT. They used an adaptive decoding method to improve the quality of their algorithm results. Cai et al. (2020) presented a dynamic shuffled frog-leaping algorithm and a lower bound for distributed HFSMT. Their algorithm is composed of population division, dynamic search process, a new destruction-construction process and population shuffling based on the evolution that led to better results than previous algorithms.

The problem addressed in this paper is the scheduling of multiprocessor jobs on a hybrid flow shop for minimizing the makespan. To achieve this aim, the study is presented in the following three major contributions:

- A method for dynamically determining the crossover and mutation rates. This method is based on evaluating the results of each generation of GA.
- Using an operator named Bad Selection Operator to escape local optima and search in the problem space.
- Presenting an irregular neighborhood search method; that is, after defining a structure called the Block Search, the block moves along the chromosome to achieve a proper sequence.

The rest of this paper is organized as follows: Section 2 states the HFSMT problem, consists of its definition, assumptions and the used notations, and then presents a mixed graph model. To solve this problem, the details of the proposed algorithm, its parameters, and the newly developed operator are presented in Section 3. The computational results are given in Section 4, and finally, the conclusions and future research directions are made clear in Section 5.

2 Problem statement

In this section, after defining the problem under study, the assumptions and notations used will be introduced. A mixed graph is also modeled for the HFSMT and its aspects are discussed by giving an example. The HFSMT is stated as follows: A set of jobs $J \in \{1,2,\dots,n\}$ with multiprocessor tasks is to be processed in $i = 1,2,\dots,k$ stages. In stage i^{th}, there are m_i parallel and identical processors (machines), with a minimum number of processors per step of 1. Job $j \in J$ in stage i^{th}, takes $size_{ij}$ identical processors for p_{ij} units of time. Not allowing for preemption, the simultaneously operating processors might process at most, one job at a time. The objective is to minimize the maximum completion time (makespan) of all the jobs in the last stage. This problem can be denoted by $Fk(Pm_1,\dots,Pm_k)|size_{ij}|C_{max}$ using the popular three-field notations where $Fk(Pm_1,\dots,Pm_k)$ denotes machine environment, $size_{ij}$ refers to processing characteristics and C_{max} represents performance criterion (Graham et al, 1979). In the literature, various researches proposed heuristic approaches and meta-heuristic approaches for the problem $(Pm_1,\dots,Pm_k)|size_{ij}|C_{max}$ that concentrated on providing near-optimal solutions.

2.1 Assumptions and notations

The following assumptions are invoked for the investigated HFSMT problem.

- All jobs and all processors are available at the beginning of the scheduling period.
- The ready time of each job is zero.
- Preemption of jobs is not allowed.
- All jobs arrive dynamically and have no precedence constraints.
- Each processor processes only one task at a time.
- Processors that are used at each stage cannot be used at other stages.
- The setup times of tasks are included in the processing times.
- The number of jobs and their processing time are fixed.
- The number of stages and the number of processors available in each stage is constant.

For the convenience of description, the notations used in this study are as follows:

Parameters

n	number of jobs
k	number of stages
m_i	number of identical parallel processors at stage $i(i = 1,2,\dots,k)$
π_i	job processing order sequence at stage *i*
p_{ij}	processing time of job *j* at stage *i*
$size_{ij}$	number of parallel processors required to process job j at stage i

Decision Variables

S_{ij}	starting processing time of job *j* at stage *i*
C_{ij}	completion time of job *j* at stage *i*
CP	critical path
C_{max}	Makespan

To give a better understanding of the problem in this paper, an example is presented.

Example: Consider the following HFSMT that consists of five jobs, two stages, and three parallel processors in each stage. The processing time and the number of processors required

to process jobs are given in Table 1. A feasible schedule is illustrated in Fig. 1, where $\pi_1 = [J_1, J_2, J_3, J_4, J_5]$.

Table 1. Processing data of the example

	J_1	J_2	J_3	J_4	J_5
$size_{1j}$	1	2	1	2	2
p_{1j}	4	3	3	4	2
$size_{2j}$	2	1	3	2	1
p_{2j}	3	2	5	3	1

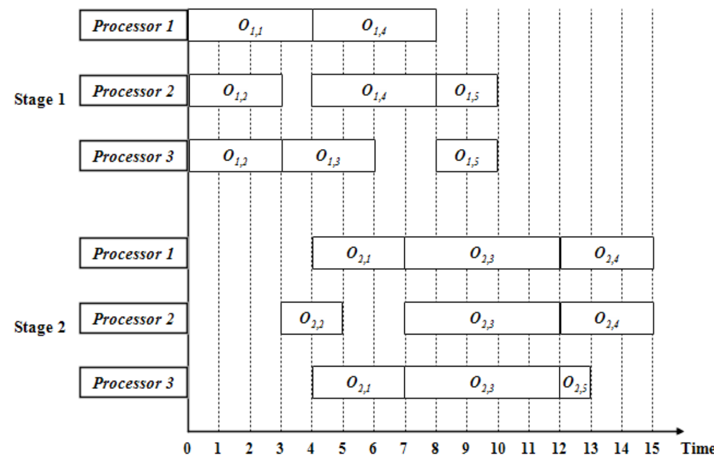


Fig. 1. Gantt chart for the example

2.2 A mixed graph model

The problem $Fk(Pm_1, \dots, Pm_k)|size_{ij}|C_{max}$ described in section 2 can be modeled using a weighted mixed graph $G = \{\mathcal{O}, \mathcal{A}, \mathcal{E}\}$, where $\mathcal{O}$, $\mathcal{A}$, and $\mathcal{E}$ represent the set of operations, the set of arcs and the set of edges, respectively (Tanaev, 1994). Here, $\mathcal{O}$ denotes the set of operations $O_{i,j}$, $j \in \{1, 2, \dots, n\}$, $i \in \{1, 2, \dots, k\}$ which should be processed by π_i processors. In other words, the processing of a job $J_j \in \mathcal{J}$ includes a set O_i of J_j ordered operations where $O_i = \{O_{1,J_j}, O_{2,J_j}, \dots, O_{k,J_j}\}$. There are two dummy operations $O_{0,0}$ and $O_{k+1,0}$, which indicate the beginning of the scheduling and the completion of the processing of all jobs, respectively. The arc $(O_{i-1,j}, O_{i,j}) \in \mathcal{A}$ where $j \in \mathcal{J}$ indicates that the starting time of job j in stage i^{th} is after the completion time of job j in stage $(i-1)^{th}$ (precedence constraint). $p_{ij}$is the weight of the arc $(O_{i-1,j}, O_{i,j})$, this is while the weight of the $(O_{0,j}, O_{1,j})$ is zero. Suppose that $m_2 = 1$, $size_{2l} = size_{2k} = 1$ where $l, k \in \mathcal{J}$ and each of the two $O_{2,l} \in \mathcal{O}$ and $O_{2,k} \in \mathcal{O}$ operations will be processed on the m_2 processor. These two operations cannot use the processor at the same time (resource constraint). This restriction is represented by $[O_{2,l}, O_{2,k}] \in \mathcal{E}$.

Two consecutive operations $O_{i,j}$ and $O_{i+1,j}$ of the same job $J_j \in \mathcal{J}$ are connected with an arc $(O_{i,j}, O_{i+1,j}) \in \mathcal{A}$, where $1 \le i \le k-1$. At stage i^{th}, the operations $O_{i,l}$ and $O_{i,k}$ are connected with an arc $(O_{i,l}, O_{i,k}) \in \mathcal{A}$. Since an operation preemption is not allowed, the sequence of

operations performed at the stage i^{th} can be $\mu_i = [O_{i,1}, O_{i,2}, \dots, O_{i,n}]$ which is based on the start time of each operation, where $S_{i1} \le S_{i2} \le \dots \le S_{in}$. For all operations $\mathcal{O}$, the conjunctive constraint $S_{mn} - S_{ij} \ge p_{ij}$ is satisfied for each arc $(O_{ij}, O_{mn}) \in \mathcal{A}$ and the disjunctive constraint either $S_{mn} - S_{ij} \ge p_{ij}$ or $S_{ij} - S_{mn} \ge p_{mn}$ is satisfied for each edge $[O_{ij}, O_{mn}] \in \mathcal{E}$. For the dummy operation $O_{0,0} \in \mathcal{O}$, the arc $(O_{0,0}, O_{1,j})$ for each job $J_j \in \mathcal{J}$ has zero weight. For the dummy operation $O_{k+1,0} \in \mathcal{O}$, p_{kj} is considered as the $(O_{k,j}, O_{k+1,0})$ arc weight.

In the weighted mixed graph $G = \{\mathcal{O}, \mathcal{A}, \mathcal{E}\}$, to make decisions, arcs $(O_{ij}, O_{mn}) \in \mathcal{A}$ or $(O_{mn}, O_{ij}) \in \mathcal{A}$ are replaced with p_{ij} or p_{mn} and/or the edge $(O_{ij}, O_{mn}) \in \mathcal{E}$, respectively. Note that there is no cycle at each stage. In graph G, the objective function is to have arcs in order to minimize the longest path from $O_{0,0}$ to $O_{k+1,0}$ which is called the critical path. The Example is used to make possible coming up with more intuitions.

Given the order of processing the jobs in the first stage in the Example, which is $\pi_1 = [J_1, J_2, J_3, J_4, J_5]$, the job scheduling table is set to the first stage shown in Table 2. Since jobs are available from the initial moment in this stage, the arrival time is zero.

Table 2. Job scheduling table in stage #1

Operation	Arrival Time	p_{1j}	$size_{1j}$
$O_{1,1}$	0	4	1
$O_{1,2}$	0	3	2
$O_{1,3}$	0	3	1
$O_{1,4}$	0	4	2
$O_{1,5}$	0	2	2

At the beginning of the scheduling, $O_{1,1}$ begins with the acquisition of one processor. Furthermore, $O_{1,2}$ needs two processors to start its process, so it will be able to start processing at time zero with the two remaining processors. $O_{1,3}$ also takes one of the processors after the $O_{1,2}$ is finished and then starts its process, since $S_{13} \le C_{12}$ (Fig 2(c)). $O_{1,4}$, which is waiting at time zero to take up the required processors and starts to process, takes one processor at $time = 3$, but $size_{14} = 2$. The processor dedicated to $O_{1,1}$, is assigned to $O_{1,4}$. It should be noted that $S_{14} = \max(C_{11}, C_{12})$ (Fig 2(d)). The $O_{1,5}$ waits for $time = 0$, at $time = 6$ takes a required processor but still cannot start. At $time = 8$ when the process of $O_{1,4}$ was completed, the processor of $O_{1,5}$ was supplied and the $O_{1,5}$ starts up (Fig 2(e)). Obviously, the completion time of $O_{1,5}$ is calculated as follows: $C_{15} = \max(C_{13}, C_{14}) + p_{15}$.

Assume $\mu_1 = [O_{1,2}, O_{1,1}, O_{1,3}, O_{1,4}, O_{1,5}]$. At stages after stage #1, the start time of operations depends on their completion time at stage #1, i.e. $S_{21} \ge C_{11}$. Accordingly, Table 3 is presented a job scheduler at stage #2. In stage #2, as in stage #1, the processor allocation process is performed to get all operations to $O_{3,0}$. After performing the operations in stage #2, one can conclude that $C_{22} \le C_{21} \le C_{23} \le C_{25} \le C_{24}$.

Fig. 3 illustrates how processors are assigned to operations and the sequence of executing the tasks using modeling by the mixed graph. Clearly, in the example given, if $\pi_1 = [J_1, J_2, J_3, J_4, J_5]$, $CP = [O_{1,2}, O_{1,1}, O_{2,2}, O_{1,3}, O_{2,1}, O_{1,4}, O_{1,5}, O_{2,3}, O_{2,5}, O_{2,4}]$ and C_{max} will be 15. The scheduling length will be shorter if proper arcs between the operations are selected. In such case, the minimum makespan over the digraph that is generated by the mixed graph will be

obtained. Note that the order of running operations in stage #1 is very important and the order of running operations in stages after stage #1 is dependent on it.

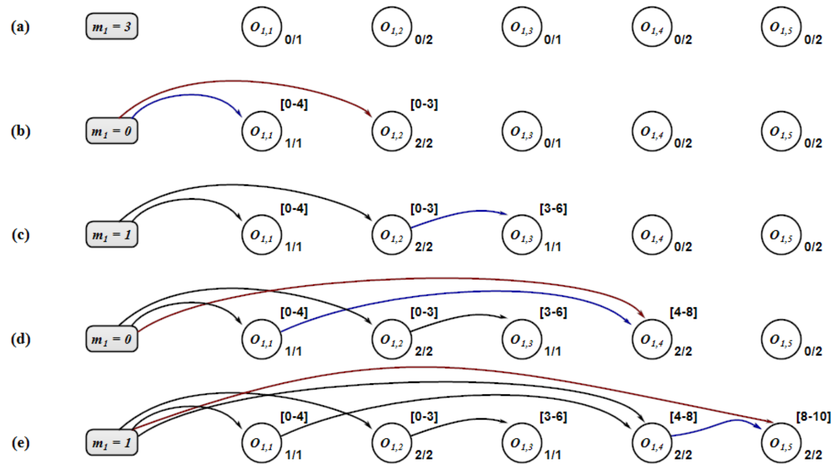


Fig. 2. The process of assigning processors to operations of mixed graph G at first stage.

Table 3. Job scheduling table in stage #2

Operation	Arrival Time	p_{2j}	$size_{2j}$
$O_{2,1}$	4	3	2
$O_{2,2}$	3	2	1
$O_{2,3}$	6	5	3
$O_{2,4}$	8	3	2
$O_{2,5}$	10	1	1

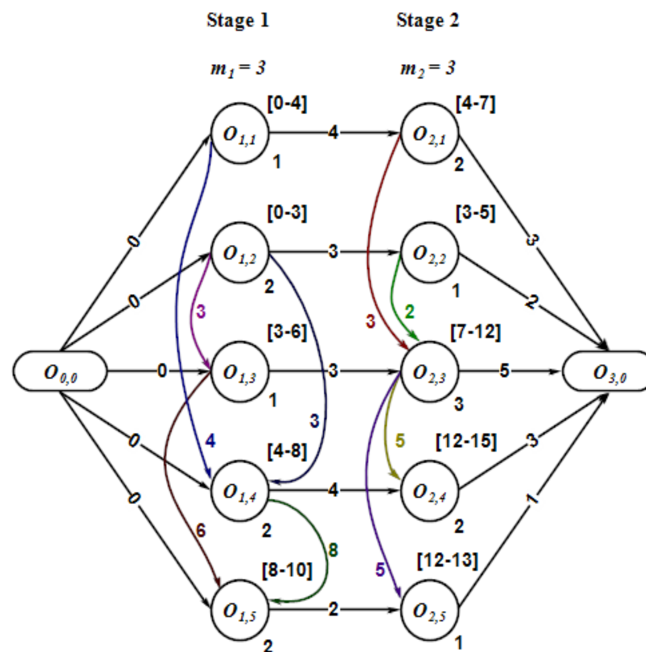


Fig. 3. The mixed graph $G = \{\mathcal{O}, \mathcal{A}, \mathcal{E}\}$ for modeling the example

3 The Memetic Algorithm

Genetic algorithm (GA) as an evolutionary algorithm is a search method based on the structure of chromosomes and has been widely used to solve optimization problems (Goldberg, 1986; Hamadani et al, 2013). The definition of the solution to the problem is carried out in the form of a chromosomal structure. The quality of each solution represented by the chromosome is expressed by the fitness function. In GAs, the predefined number of chromosomes is produced as random or quasi-random, the set of chromosomes is known as initial population (first generation). There are always many solutions for the problem that they have mostly low quality. By crossover and mutation operators, the next generation which usually improves the quality of solutions is produced. In fact, each of these operators led to the formation of one or several new chromosomes which are added to the population. The generation creation process ends when the convergence is observed in the produced chromosomes or the stopping condition of the algorithm happens.

Memetic Algorithm (MA) combines local searches with a stochastic search to achieve better results in less time. In this method, the result of the evolution process is improved by using various local search methods designed for the problem. Accordingly, the proposed MA is a combination of GA and the local search. So, the concepts and the evolutionary process of the GA are described and then the types of local searches which are used in the proposed algorithm are briefly explained.

3.1 Representation of chromosome

The performance of algorithms which solve HFSMT problems depends on the scheduling of the jobs and how processors are assigned to the jobs. Therefore, having a proper framework for each of these two tasks is important. Hence, each gene of this chromosome at first stage is represented by $O_{1,j}$. Consider the example: $[O_{1,2}, O_{1,1}, O_{1,3}, O_{1,4}, O_{1,5}]$ is a chromosome, and $O_{1,4}$ is considered as a gene as well. This chromosome represents a scheduler that starts from $O_{0,0}$ and ends with $O_{k+1,0}$. The length of the chromosome mentioned above is 5.

3.2 Fitness of chromosome

In this study, the fitness function is the makespan (C_{max}). The chromosomes are compared on the basis of their fitness functions.

3.3 Initial Population

The proposed algorithm in this paper creates the initial population in two ways. In the first way, operations of the first stage are randomly selected to build each chromosome so that π_1 is completed. This way generates diverse individuals. In the second way, using $size_{ij}$ and p_{ij}, an appropriate population is produced by creating priority rules. The rules are for assigning the unscheduled jobs in the first stage. This way, which includes the famous rules of the allocation of the processors to the jobs, is as follows (Xu et al, 2013):

- Shortest processing time first (SP), in which jobs are sorted in ascending order based on $p_{1j}\ \forall\ j = 1, 2, \dots, n$.
- Longest processing time first (LP), in which jobs are sorted in descending order based on $p_{1j}\ \forall\ j = 1, 2, \dots, n$.
- Smallest processor requirement (SPR), in which jobs are sorted in ascending order based on $size_{1j} \forall\ j = 1, 2, \dots, n$

- Largest processor requirement (LPR), in which jobs are sorted in descending order based on $size_{1j}\ \forall\ j = 1, 2, \ldots, n$
- Shortest total processing time first (STP), in which jobs are sorted in ascending order based on total $p_{ij}\ \forall\ i = 1, 2, \ldots, k,\ j = 1, 2, \ldots, n$.
- Longest total processing time first (LTP), in which jobs are sorted in descending order based on total $p_{ij}\ \forall\ i = 1, 2, \ldots, k,\ j = 1, 2, \ldots, n$.
- Smallest total processor requirement (STPR), in which jobs are sorted in ascending order based on total $size_{ij}\ \forall\ i = 1, 2, \ldots, k,\ j = 1, 2, \ldots, n$.
- Largest total processor requirement (LTPR), in which jobs are sorted in descending order based on total $size_{ij}\ \forall\ i = 1, 2, \ldots, k,\ j = 1, 2, \ldots, n$.
- Smallest occupied capacity (SOC), in which jobs are sorted in ascending order based on $p_{1j} \times size_{1j}\ \forall\ j = 1, 2, \ldots, n$.
- Largest occupied capacity (LOC), in which jobs are sorted in descending order based on $p_{1j} \times size_{1j}\ \forall\ j = 1, 2, \ldots, n$.
- Smallest total occupied capacity (STOC), in which jobs are sorted in ascending order based on total $p_{ij} \times size_{ij}\ \forall\ i = 1, 2, \ldots, k,\ j = 1, 2, \ldots, n$
- Largest total occupied capacity (LTOC), in which jobs are sorted in descending order based on total $p_{ij} \times size_{ij}\ \forall\ i = 1, 2, \ldots, k,\ j = 1, 2, \ldots, n$

For stages after the first stage, the first-come-first-served (FCFS) rule is used to determine the sequence of operations, so that neither the processor requirement nor the flowshop constraints are violated. To complete π_i where i is larger than one, two below-mentioned methods are used. One method is LS. In this method, the jobs are scheduled in ascending order, according to the completion time in the previous stage. For example, if $C_{12} \leq C_{14} \leq C_{13}$, then $S_{22} \leq S_{24} \leq S_{23}$. Another method is BF. In this method, the jobs are firstly scheduled by LS; then rescheduling occurs whenever the below conditions are satisfied: the processing time for unassigned operations is smaller than processing time of the running operation, while the idle amount of processors is greater than or equal to the number of processors required for unassigned operations.

The BF method is based on the fact that makespan will be better if one can reduce the idle time of the processors. The completion time of the jobs in terms of LS and BF is calculated as the one with a smaller C_{max} is selected as C_{max} of chromosome. In other words, $C_{max} = \min\{C_{max}(LS),\ C_{max}(BF)\}$.

3.4 Crossover

A kind of operation like a compound that extensively affects chromosome encoding is a crossover. The crossover types can be different based on the encoding type. In this paper, there are two types of crossover operators that are based on the One Parent-Two Point Crossover method, sequentially, One Parent-Two Point Crossover (SOTC) and Randomly One Parent-Two Point Crossover (ROTC). In these methods, two points of a selected chromosome are randomly chosen. The operations inside these two points are called part #1, and other operations are called part #2. Afterward, from numbers 0 and 1, a number is randomly selected. If the number is equal to 0, then part #1 will be transferred to the first part of the child's chromosome, and if the number is 1, then the part #2 will go. Part #2 is also the opposite of part #1 in the child's chromosome. In SOTC, operations are in part #2 in the same order as the parent. But in ROTC, the operations are randomly assigned to part #2. The crossover operator for example 1 is shown in Fig.4.

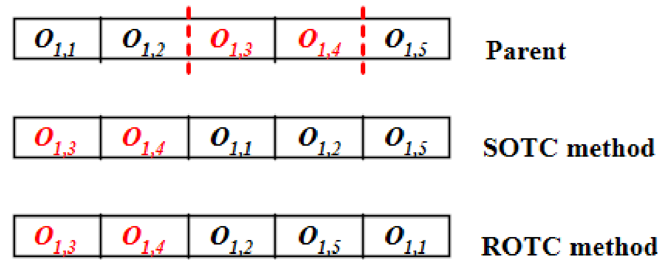


Fig. 4. Representation of crossover operators

3.5 Mutation

The mutation operator causes minor changes in the chromosome and creates a greater variety of new locations to discover better solutions. Three traditional mutations are examined in the MA to minimize the makespan: swap mutation, inversion mutation and scramble mutation (Goldberg, 1986; Hamadani et al, 2013). For example 1, the types of mutation operators are shown in Fig.5.

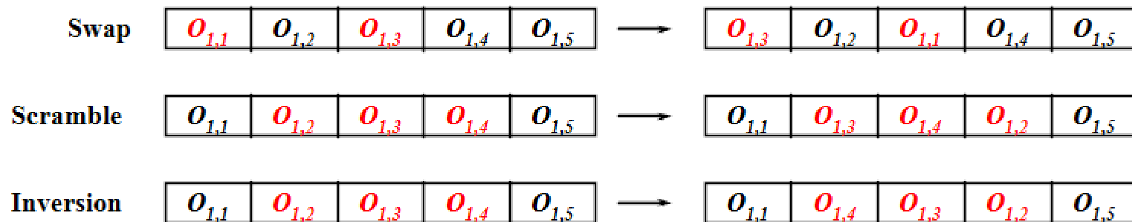


Fig 5. Representation of mutation operators.

3.6 Selection operator

The selection operator chooses the best chromosomes to speed up and also selects the best parents for crossover and the mutation in subsequent periods. The selection operator has different types that can be referred to as Tournament Selection and Elitism Selection (Sharma and Wadhwa, 2014). In the Tournament Selection method, a subset of chromosomes is selected and members of that set compete. After the competition, only one chromosome is transferred to the new generation. Of course, the best previous generation solutions for the new generation will likely be lost in the Tournament Selection method. Therefore, the Elitism Selection method is also used in this paper. In this method, the best solutions produced in the previous generation are transferred directly to the next generation without comparison. So, the advantages of both methods are used.

3.7 Bad selection operator

The evolution algorithms face the problem of premature convergence. To avoid this problem, an operator named Bad Selection Operator (BSO) is used in the proposed algorithm so that the algorithm does not get trapped in the local optimality as much as possible. The BSO moves chromosomes of any generation which do not have an acceptable fitness to a list called Bad List. If the solution does not improve in several generations, the BSO uses them. The chromosomes available in the bad list are used as follows. When the algorithm is trapped in the local optimal, the chromosomes with the best fitness and the ones in the bad list are used as parents for the two parent-one point crossover. This operator increases the diversity of genes in chromosomes. Besides, solutions with fewer similarities are generated by the algorithm (see

Fig. 6). Suppose Parent (a) is a chromosome selected from bad list and Parent (b) is the chromosome that has the best fitness. A point on both parents' chromosomes is picked randomly, and designated as a crossover point. The crossover point is represented by a red long dash line. According to Fig. 6, this crossover produces two children. Thus, the first part of Parent (a) is transferred directly to the Offspring (a). Subsequently, to construct the second part of Offspring (a), the operations of the second part of Parent (b) are scrolled in sequence, and the operations seen in the first part of Offspring (a) are not added to Offspring (a). After the second part of Parent (b) scrolling is completed, if the second part of Offspring (a) is not completed and there are vacancies, operations that are not yet in the Offspring (a) will be randomly assigned to the remaining locations of Offspring (a).

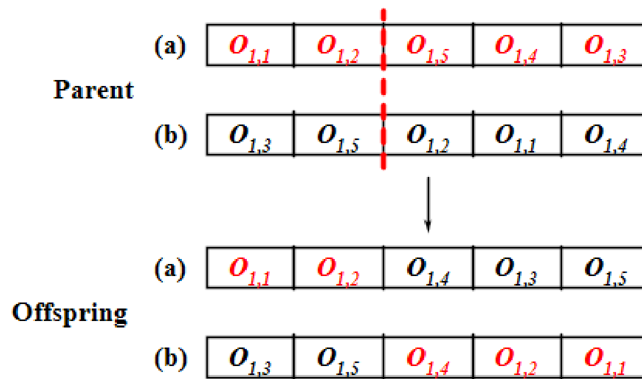


Fig. 6. Two Parent-One Point crossover operator

In the BSO, there is a parameter called Same Fitness Repetition (SFR), which indicates the number of generations that have failed to achieve better makespan. After SFR has exceeded a predefined number, the BSO starts up the crossover operator and adds offspring which produced to the next generation to make diversity in the population. The parameter SFR is determined based on error and trial method that in the proposed algorithm is considered 20. Fig. 7 shows the performance of the BSO. In this figure, the red lines indicate where the BSO improves the fitness function. In fact, the algorithm in iteration 21 to 40 could not improve the fitness function. Hence, BSO began to produce offspring from iteration 41 and succeeded in improving makespan in iteration 52. After success in improving the fitness function, the bad list becomes empty and from iteration 53, the chromosomes with an unacceptable makespan will be added to the bad list. The activity of this operator improves the makespan as shown in Fig. 7.

3.8 Stopping criteria

In the proposed algorithm, two stopping criteria are considered. Limitation on the maximum number of iterations is the first one. The second stopping criterion is the situation in which the makespan reaches the specified value in the lower bound, then the specified number cannot be fulfilled before it can be predefined.

3.9 Neighborhood solution

Tabu search (TS) is a local search algorithm that has memory (Glover, 1989). This algorithm keeps a list of neighbors that have been previously searched in a list called tabu list. The memory feature causes the algorithm to not be cycled. In this algorithm, a solution is first selected (chromosome), then a list of its neighbors is created as X. Neighbors' fitness is examined in the list of X. If a neighbor is better than the best solution ever found, it will be

accepted, otherwise, it will be added to the tabu list and the list will be updated. In this method, TS is modified as follows: Initially, a block of operations is selected. As the block moves along the chromosome, a neighbor is produced. The neighbor will be checked, and the search will finish if it satisfies the ending condition of the search. Otherwise, the neighbor will be in the tabu list and the list will be updated, and again by moving the block, another neighbor will be found. This method is called Block Search in this paper.

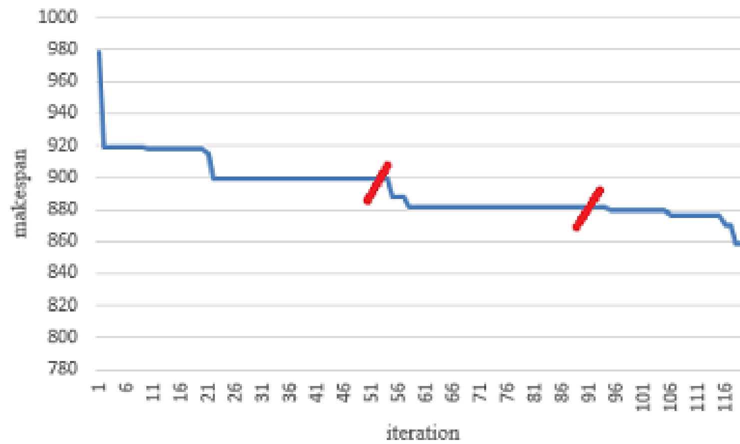


Fig. 7. Performance of Bad Selection Operator

In the Block Search, the length of the block is about one to a quarter of the length of the chromosome. In other words, the block size is in the interval [1 – (*chromosome length* / 4)], which is selected at random. Then a number is selected as the starting point of block, which can be between one and $chroLen - blockSize$. By moving block between operations, a new chromosome is obtained. This chromosome can be the best solution or a member of the tabu list. If the solution is ever found, neighborhood search will stop. Otherwise, this operation related to Block Search continues until the block returns to the starting point.

For example, consider a chromosome that has 20 genes; i.e. $chroLen = 20$. As mentioned above, a number is chosen as the block size from 1 to 5. Assume that $blockSize = 2$. Here, the starting point of the block is 1 to 18. Assume that $O_{1,25}$ and $O_{1,27}$ are selected as a block (Fig.8-a). Then the block between operations is moved (Fig. 8-b). The fitness of the new chromosome is calculated and if it's better than the best solution ever found, it replaces the best solution; otherwise, it will be placed in the tabu list. The list is updated and the block is repositioned again (Fig. 8-c).

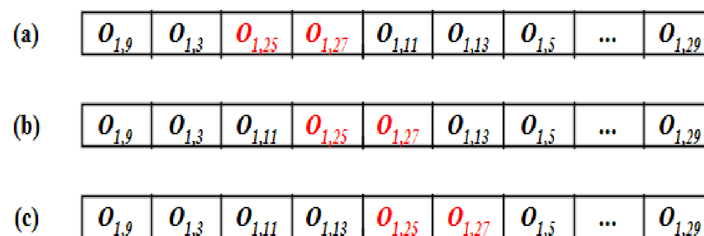


Fig. 8. Representation of Block Search

3.10 Dynamic adjustment of the ratio of crossover and mutation operator

Mutation and crossover operators are key operators in the proposed MA whose ratio is determined before the algorithm is executed. In this case, their ratios are constant and they perform their respective operations for each iteration at the number of times specified in the ratio. If each operator fails to perform its intended function, it must be executed at the ratio specified. Executives that do not lead to better results can increase execution time without any improvement. On the other hand, one of the operators may have a greater impact on the improvement of the objective function, and if the ratio increases, the process of improving the results by that particular operator may continue. Therefore, the dynamic determination of operator ratios is the topic discussed in this section. For this purpose, the percentage of the effect of the operators on the improvement of the solution is calculated and if the efficiency is increased, the ratio of that operator will increase for the next iteration. Increasing the ratio in an operator will result in the decrease of ratio in the other operator. For example, as the ratio of the crossover operator increases, the ratio of the mutation operator decreases, and conversely, as the ratio of mutation operator decreases, the ratio of the crossover operator decreases. It is worth noting that decreasing or increasing the value of the ratio will not be rapid and it will be gently done. Besides to avoid reaching zero in searching the problem space, minimum ratios for each of the operators is intended or considered. Since the increase in the ratio of each operator has an opposite effect on the other operator, the maximum ratio is also set for each operator. The pseudo-code of the dynamic adjusting ratio of the crossover operator is presented in Algorithm 1.

Algorithm 1: The pseudo-code of improving the ratio of crossover operator

```
For 1 to crossoverCount:
        Do crossoverOperator()
        sumOPT += amount of offspring's optimum
        sumFit += offspring's fitness
End For
coefficient = (sumOPT * 100) / sumFit
If (coefficient >lastCoeff):
        pc++  // suitability of the operator
        mc -= (mc/2)  // unsuitability of the operator
        crossoverCount += pc
Else:
        mc ++
        pc -= (pc/2)
        crossoverCount -= mc
End If
lastCoeff = coefficient
// limiting min and max of crossover ratio:
If (crossoverCount<minCrossover):
        crossoverCount = minCrossover
End If
If (crossoverCount>maxCrossover):
        crossoverCount = maxCrossover
End If
```

In this paper, the minimum and maximum ratios for an operator are calculated as follows: suppose crossover is the desired operator, then, maximum ratio: crossover ratio, and minimum ratio: (crossover ratio / 4).

4 Computational experiments

In this section, the findings of the MA implementation on the HFSMT problem are discussed, and the obtained results will be compared with the algorithms presented in the literature. The performance of the proposed algorithm was tested on the benchmark generated by Oguz et al. (2004), which can be accessed from the http://home.ku.edu.tr/~coguz/.

The generated benchmark has the following conditions: p_{ij} and $size_{ij}$ are $[1, 100]$ and [1,5], respectively. The number of jobs are 10, 20, 50 and 100. Also, the number of stages are 2, 5 and 8. In fact, the benchmark has two types of problems: type I and type Π. In the type I, the number of processors varies in different stages, but in the type Π, the number of processors is fixed as 5 at all stages. In both types of problems, ten problem instances were generated for each combination of n and k. Thus, the test bed was composed of 240 benchmark instances. The proposed MA has been coded in Java 8, and the settings of the parameters in the MA are as follows: initial population: 50, 80, 100, selection ratio: 0.2, 0.6, 0.8, crossover ratio: 0.5, 0.7, 0.8, 0.9, mutation ratio: 1-crossover ratio, max iteration: 80, 100, 120. For each instance, a combination of parameter values triggered MA. Any combination of execution that has a smaller makespan was selected as the makespan of that instance. The lower bound presented in Oguz et al. (2004) is used as the performance measure.

Table 4. LS and BF performance on makespan

K	N	Type I		Type Π	
		LS	BF	LS	BF
2	5	0	2	1	1
	10	1	0	1	1
	20	0	3	0	2
	50	0	4	0	5
	100	0	4	0	8
5	5	0	0	0	2
	10	1	3	0	8
	20	1	5	2	7
	50	0	6	0	10
	100	0	9	0	10
8	5	2	2	0	1
	10	1	4	3	6
	20	2	6	2	8
	50	1	7	0	10
	100	0	10	0	10

To evaluate the effects of the LS and BF methods described in the previous section, the STP rule is used to compare the performance of each of the methods in terms of C_{max}. Each problem instance is independently run two times for the SPR rule; once based on the LS method, and one time based on the BF method. For each run, the makespan is obtained and evaluated in three ways: $LS_{C_{max}} = BF_{C_{max}}$, $LS_{C_{max}} < BF_{C_{max}}$ and $LS_{C_{max}} > BF_{C_{max}}$. The obtained results are presented in Table 4. These results indicate that BF has a better performance in reducing makespan. For example, in type Π, $k = 5$, $n = 20$, the number of times that the $LS_{C_{max}} < BF_{C_{max}}$ is 7, in contrast to the case where the $LS_{C_{max}} > BF_{C_{max}}$, which is 2 times. There is only one case where $LS_{C_{max}} = BF_{C_{max}}$. To evaluate the performance of the proposed algorithm, the computational results of the average makespan ($Ave. C_{max}$) for the benchmark problems were

compared with the algorithms GA (Oğuz and Ercan, 2005), ACS (Ying and Lin, 2007), SA (Wang et al., 2011), HABC (Lin et al., 2013), MA (Jouglet et al., 2009), HIA (Ying, 2012), DFA (Rani, and Zoraida, 2016), and ACSNDP (Kurdi, 2019). The obtained results are shown in Table 5.

Table 5. Computational results of the previous algorithms presented and the proposed MA.

Type	K	n	GA	ACS	SA	HABC	MA	HIA	DFA	ACSNDP	The proposed MA
I	2	10	451.1	451.1	451.1	451.1	451.1	451.1	**444.1**	451.1	451.1
		20	876.7	877.1	875.3	875.9	877.7	875.9	**870.1**	875.9	874.2
		50	2049.4	2052.1	2038.5	2038.6	2046.1	2042.3	2038.4	2045.0	**2029.2**
		100	4355.0	4363.8	4349.2	4346.8	4348.6	4349.0	**4346.1**	4348.2	4347.9
	5	10	639.1	645.1	643.0	641.3	**637.8**	641.4	639.6	644.1	638.3
		20	1072.1	1074.7	1064.9	1066.6	1070.3	1066.5	1067.4	1070.3	**1060.7**
		50	2604.0	2577.2	2566.6	2564.2	2571.7	2564.6	2604.0	2573.0	**2563.9**
		100	4755.0	4697.1	4670.7	**4664.0**	4746.9	4696.7	4689.8	4682.6	4665.8
	8	10	836.9	852.4	847.8	851.3	840.3	843.0	**834.6**	848.7	837.5
		20	1319.3	1315.6	1303.2	1302.4	1315.2	1305.3	1315.8	1309.8	**1295.6**
		50	2669.6	2644.8	2609.3	**2599.1**	2623.1	2609.7	**2599.1**	2623.8	**2599.1**
		100	5327.7	5231.3	5206.4	5191.4	5267.2	5229.1	**5168.9**	5213.3	5196.6
Π	2	10	409.5	423.7	422.8	422.0	422.1	422.2	**402.7**	422.9	420.7
		20	757.4	808.3	806.0	806.1	807.6	806.3	**747.9**	806.7	805.6
		50	1671.1	1736.8	1715.6	1716.0	1722.2	1719.9	**1656.0**	1720.8	1702.7
		100	3205.3	3268.3	3191.6	**3166.6**	3215.0	3215.0	**3166.6**	3194.5	3186.7
	5	10	616.3	600.3	598.8	599.9	594.9	595.1	600.3	599.5	**591.8**
		20	948.5	944.3	926.9	924.5	945.3	928.2	931.1	937.7	**918.8**
		50	2074.6	1978.1	1912.5	**1850.5**	1960.7	1884.9	**1850.5**	1940.6	1898.3
		100	4145.3	3635.5	3530.0	**3415.7**	3802.0	3616.1	3427.3	3599.8	3516.7
	8	10	**813.1**	842.2	829.9	827.1	824.0	828.3	816.3	832.7	817.9
		20	1148.0	1134.0	1095.6	1099.4	1133.4	1100.3	1046.8	1119.8	**1042.6**
		50	2321.8	2268.5	2194.4	**2128.0**	2315.7	2175.5	**2128.0**	2210.6	2129.7
		100	4216.6	4078.6	3992.8	3855.7	4330.7	4067.6	3855.7	4025.7	**3855.1**

Results clearly reveal that among the nine approaches, the proposed MA was able to obtain 37.5% of the problems with the lowest value of Ave. C_{max}. For the entire 24 cases, it can be seen from this table that the proposed MA yields better results on 22 cases and the same result on case #1, and worse result on one case in comparison with HIA. In comparison with HABC, the proposed MA yields better results on 15 cases and the same results on cases #1 and #11 and also worse results in 7 cases. In comparison with DFA, the proposed MA yields better results on 10 cases and the same result on case #11 and also worse results on 13 cases; however, in stages after stage #2, the proposed algorithm yields the better results on 9 cases and the worse results on 6 cases. Although the number of cases with better results obtained from the DFA is slightly higher than that of the proposed MA, the proposed algorithm yields better results in stages after stage #2. The results of the proposed MA are better than the ones of ACS, SA, and ACSNDP in all of the cases except one case. In comparison with GA, the proposed MA yields better results on 18 cases, the same result on case #1, and worse results in 5 cases.

In order to make sure that the proposed MA is good and efficient for the benchmark problems, the average percentage deviation (PD) is computed as follows:

$$PD = (C_{max}^{Algorithm(?)} - LB)/LB \times 100$$

where $C_{max}^{Algorithm}$ is the makespan obtained by each one of algorithms included GA, ACS, SA, HABC, MA, HIA, DFA, and ACSNDP. Besides, LB is the lower bound presented in Oguz et al. (2004). The computational results are summarized in Table 6. Based on results of this Table, one can conclude that the proposed MA is efficient for problem Type I and Type II.

Table 6. Computational results of PD.

Type	K	n	GA	ACS	SA	HABC	MA	HIA	DFA	ACSNDP	The proposed MA
I	2	10	2.41	2.41	2.41	2.41	2.41	2.41	**0.82**	2.41	2.41
		20	2.04	2.08	1.87	1.94	2.15	1.94	**1.27**	1.94	1.75
		50	1.93	2.07	1.39	1.40	1.77	1.58	1.39	1.72	**0.93**
		100	0.54	0.74	0.40	0.35	0.39	0.40	**0.33**	0.38	0.37
	5	10	8.69	9.71	9.35	9.06	**8.47**	9.08	8.78	9.54	8.55
		20	5.07	5.32	4.36	4.53	4.89	4.52	4.61	4.89	**3.95**
		50	3.22	2.16	1.74	1.65	1.94	1.66	3.22	1.99	**1.63**
		100	2.30	1.05	0.49	**0.34**	2.13	1.05	0.90	0.74	0.38
	8	10	19.06	21.27	20.61	21.11	19.55	19.93	**18.74**	20.74	19.15
		20	14.58	14.26	13.18	13.11	14.23	13.37	14.28	13.76	**12.52**
		50	5.63	4.65	3.24	**2.84**	3.79	3.26	**2.84**	3.82	**2.84**
		100	3.66	1.78	1.30	1.00	2.48	1.74	**0.57**	1.43	1.11
II	2	10	9.49	13.29	13.05	12.83	12.86	12.89	**7.67**	13.07	12.49
		20	4.21	11.21	10.90	10.91	11.12	10.94	**2.90**	10.99	10.84
		50	4.16	8.26	6.94	6.96	7.35	7.21	**3.22**	7.26	6.13
		100	3.68	5.72	3.24	**2.43**	4.00	4.00	**2.43**	3.34	3.08
	5	10	32.91	29.46	29.14	29.37	28.29	28.34	29.46	29.29	**27.63**
		20	17.62	17.10	14.94	14.65	17.22	15.10	15.46	16.28	**13.94**
		50	18.98	13.44	9.68	**6.12**	12.44	8.10	**6.12**	11.29	8.87
		100	28.23	12.46	9.20	**5.66**	17.61	11.86	6.02	11.36	8.79
	8	10	**20.71**	25.03	23.20	22.79	23.33	22.97	21.18	23.62	21.42
		20	26.98	25.43	21.18	21.60	25.36	21.70	15.78	23.86	**15.32**
		50	24.76	21.90	17.92	**14.35**	24.43	16.90	**14.35**	18.79	14.44
		100	20.67	16.72	14.26	10.34	23.93	16.40	10.34	15.20	**10.32**

5 Conclusion and future research

In this paper, an effective memetic algorithm was proposed to solve the problem $Fk(Pm_1, \ldots, Pm_k)|size_{ij}|C_{max}$. This algorithm is a neighborhood search based on tabu search which selects the operations inside the block. It attempts to find the appropriate neighborhood for each solution by moving the block among other operations. A new operator called the bad selection operator, which is attempting to improve the makespan through the makespan observed at specified intervals. To minimize the execution time of the algorithm, a method to improve the ratio of the crossover was introduced. The experimental results clearly indicate

that for problems that have more than two stages, the proposed algorithm could yield better performance than most of the existing algorithms.

Future research should be focused on studying this problem in uncertain environments where the operation processing time is not deterministic but stochastic, which makes the HFSMT model more practical. Some other crossover techniques should be added to the proposed MA and then the investigation of the solution quality should be carried out.

References

Abouei Ardakan, M., Hakimian, A., & Rezvan, M. T. (2014). A branch-and-bound algorithm for minimising the number of tardy jobs in a two-machine flow-shop problem with release dates. *International Journal of Computer Integrated Manufacturing*, 27(6), 519-528.

Bathrinath, S., Saravanasankar, S., Mahapatra, S. S., Singh, M. R., Ponnambalam, S.G. (2016). An improved meta-heuristic approach for solving identical parallel processor scheduling problem. *Proceedings of the Institution of Mechanical Engineers, Part B: Journal of Engineering Manufacture*, 230(6), 1114-1126.

Behnamian, J., & Zandieh, M. (2011). A discrete colonial competitive algorithm for hybrid flowshop scheduling to minimize earliness and quadratic tardiness penalties. *Expert Systems with Applications*, 38(12), 14490-14498.

Besbes, W., Teghem, J., & Loukil, T. (2010). Scheduling hybrid flow shop problem withnon-fixed availability constraints. *European Journal of Industrial Engineering*, 4(4), 413-433.

Cai, J., Zhou, R., & Lei, D. (2020). Dynamic shuffled frog-leaping algorithm for distributed hybrid flow shop scheduling with multiprocessor tasks. *Engineering Applications of Artificial Intelligence*, 90, 1-13.

Chen, L., Bostel, N., Dejax, P., Cai, J., & Xi, L. (2007). A tabu search algorithm for the integrated scheduling problem of container handling systems in a maritime terminal. *European Journal of Operational Research*, 181(1), 40-58.

Chou, F.D. (2013). Particle swarm optimization with cocktail decoding method for hybrid flow shop scheduling problems with multiprocessor tasks. *International Journal of Production Economics*, 141, 137-145.

Engin, O., Ceran, G., & Yilmaz, M.K. (2011). An efficient genetic algorithm for hybrid flow shop scheduling with multiprocessor task problems. *Applied Soft Computing*, 11, 3056-3065.

Geem, Z. W., Kim, J. H., & Loganathan G. V. (2001). A New Heuristic Optimization Algorithm: Harmony Search. *Simulation*, 76(2), 60-68.

Glover, F. (1989). Tabu Search-Part I. *ORSA Journal on Computing*, 1, 190-206.

Goldberg, D.E. (1989). *Genetic algorithms in search, optimization and machine learning*. Addison Wesley, Redwood City.

Graham, R.L., Lawler, E.L., Lenstra, J.K., & RinnooyKan, A.H.G. (1979). Optimization and approximation in deterministic sequencing and scheduling: a survey. *Annals of Discrete Mathematics*, 4, 287-326.

Guinet, A., Solomon, M. M., Kedia, P. K., & Dussauchoy, A. (1996). A computational study of heuristics for two-stage flexible flowshops. *International Journal of Production Research*, 34(5), 1399-1415.

Gupta, J. N., & Tunc, E. A. (1994). Scheduling a two-stage hybrid flowshop with separable setup and removal times. *European Journal of Operational Research*, 77(3), 415-428.

Hamadani, A. Z., Ardakan, M. A., Rezvan, T., & Honarmandian, M. M. (2013). Location-allocation problem for intra-transportation system in a big company by using meta-heuristic algorithm. *Socio-Economic Planning Science*s, 47(4), 309-317.

Jin, Z. H., Ohno, K., Ito, T., & Elmaghraby, S. E. (2002). Scheduling hybrid flowshops in printed circuit board assembly lines. *Production and Operations Management*, 11(2), 216-230.

Jouglet, A., Oguz, C., & Sevaux, M. (2009). Hybrid flow shop: a memetic algorithm using constraint based scheduling for efficient search. *Journal of Mathematical Modelling and Algorithms*, 8, 271-292.

Kahraman, C., Engin, O., Kaya, İ., & Öztürk, R. E. (2010). Multiprocessor task scheduling in multistage hybrid flow-shops: A parallel greedy algorithm approach. *Applied Soft Computing*, 10(4), 1293-1300.

Kurdi, M. (2019). Ant colony system with a novel Non-Daemon Actions procedure for multiprocessor task scheduling in multistage hybrid flow shop. *Swarm and Evolutionary Computation*, 44, 987-1002.

Lahimer, A., Lopez, P., & Haouari, M. (2013). Improved bounds for hybrid flow shop scheduling with multiprocessor tasks. *Computers & Industrial Engineering*, 66(4), 1106-1114.

Lenstra, J.K., RinnooyKan, A.H.G., & Brucker, P. (1977). Complexity of machine scheduling problems. *Annals of Discrete Mathematics*, 1, 343-362.

Lin, S.W., Ying, K.C., & Huang, C.Y. (2013). Multiprocessor task scheduling in multistage hybrid flow shops: A hybrid artificial bee colony algorithm with bi-directional planning. *Computers & Operations Research*, 40, 1186-1195.

Peng, K., Pan, Q. K., Gao, L., Zhang, B., & Pang, X. (2018). An improved artificial bee colony algorithm for real-world hybrid flowshop rescheduling in steelmaking-refining continuous casting process. *Computers & Industrial Engineering*, 122, 235-250.

Pinedo, M. (2005). *Planning and scheduling in manufacturing and services*. New York, Springer.

Oğuz, C., & Ercan, M. F. (2005). A genetic algorithm for hybrid flow shop scheduling with multiprocessor tasks. *Journal of Scheduling*, 8, 323-351.

Oğuz, C., Ercan, M. F., Cheng, T. E., & Fung, Y. F. (2003). Heuristic algorithms for multiprocessor task scheduling in a two-stage hybrid flow-shop. *European Journal of Operational Research*, 149(2), 390-403.

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

Rani, A.D.C., & Zoraida B.S.E. (2016). Multistage multiprocessor task scheduling in hybrid flow shop problems using discrete firefly algorithm. *International Journal of Advanced Intelligence Paradigms*, 8, 377-391.

Riane, F., Artiba, A., & Elmaghraby, S. E. (1998). A hybrid three-stage flowshop problem: efficient heuristics to minimize makespan. *European Journal of Operational Research*, 109(2), 321-329.

Ruiz, R. & Vázquez-Rodríguez, J. A. (2010). The hybrid flow shop scheduling problem. *European journal of operational research*, 205(1), 1-18.

Sharma, P. & Wadhwa, A., (2014). Analysis of selection schemes for solving an optimization problem in genetic algorithm", *International Journal of Computer Applications*, 93 (11).

Serifoglu, F.S. & Ulusoy, G. (2004). Multiprocessor task scheduling in multistage hybrid flow shops: a genetic algorithm approach. *Journal of the Operational Research Society*, 55, 504-512.

Singh, M. R., Singh, M., Mahapatra, S. S., & Jagadev, N. (2016). Particle swarm optimization algorithm embedded with maximum deviation theory for solving multi-objective flexible job shop scheduling problem. *The International Journal of Advanced Manufacturing Technology*, 85(9-12), 2353-2366.

Tanaev, V., Sotskov, Y., & Strusevich, V. (1994). *Scheduling Theory: Multi-stage Systems*. Kluwer Academic Publishers, Dordrecht, The Netherlands,

Tavakkoli-Moghaddam, R., Safaei, N., & Sassani, F. (2009). A memetic algorithm for the flexible flow line scheduling problem with processor blocking. *Computers & Operations Research*, 36(2), 402-414.

Tseng, C.T., & Liao, C.J. (2008). A particle swarm optimization algorithm for hybrid flow shop scheduling with multiprocessor tasks. *International Journal of Production Research*, 46, 4655-4670.

Wang, H.M., Chou, F.D., & Wu, F.C. (2011). A simulated annealing for hybrid flow shop scheduling with multiprocessor tasks to minimize makespan. *The International Journal of Advanced Manufacturing Technology*, 53, 761-776.

Xu, Y., Wang, L., Wang, S., & Liu, M. (2013). An effective immune algorithm based on novel dispatching rules for the flexible flow shop scheduling problem with multiprocessor tasks. *The International Journal of Advanced Manufacturing Technology*, 67, 121-135.

Ying, K. C. (2009). An iterated greedy heuristic for multistage hybrid flowshop scheduling problems with multiprocessor tasks. *Journal of the Operational Research Society*, 60(6), 810-817.

Ying, K.C., & Lin, S.W. (2007). Multiprocessor task scheduling in multistage hybrid flow shops: an ant colony system approach. *International Journal of Production Research*, 44, 3161-3177.

Ying, K.C. (2012). Minimising makespan for multistage hybrid flow-shop scheduling problems with multiprocessor tasks by a hybrid immune algorithm. *European Journal of Industrial Engineering*, 6(2), 199-215.

Ying, K. C., & Lin, S. W. (2018). Minimizing makespan for the distributed hybrid flow shop scheduling problem with multiprocessor tasks. *Expert Systems with Applications*, 92, 132-141.