An Extended Flexible Job Shop Scheduling Problem with Parallel Operations

Willian Tessaro Lunardi
Interdisciplinary Centre for Security, Reliability and Trust (SnT)
University of Luxembourg
29 Avenue J.F Kennedy L-1855
Luxembourg City, Luxembourg
willian.tessarolunardi@uni.lu

Holger Voos
Interdisciplinary Centre for Security, Reliability and Trust (SnT)
University of Luxembourg
29 Avenue J.F Kennedy L-1855
Luxembourg City, Luxembourg
holger.voos@uni.lu

ABSTRACT

Traditional planning and scheduling techniques still hold important roles in modern smart scheduling systems. Realistic features present in modern manufacturing systems need to be incorporated into these techniques. Flexible job-shop scheduling problem (FJSP) is one of the most challenging combinatorial optimization problems. FJSP is an extension of the classical job shop scheduling problem where an operation can be processed by several different machines. In this paper, we consider the FJSP with parallel operations (EFJSP) and we propose and compare a discrete firefly algorithm (FA) and a genetic algorithm (GA) for the problem. Several FJSP and EFJSP instances were used to evaluate the performance of the proposed algorithms. Comparisons among our methods and state-of-the-art algorithms are also provided. The experimental results demonstrate that the FA and GA achieved improvements in terms of efficiency and efficacy. Solutions obtained by both algorithms are comparable to those obtained by algorithms with local search. In addition, based on our initial experiments, results show that the proposed discrete firefly algorithm is feasible, more effective and efficient than our proposed genetic algorithm for the considered problem.

CCS Concepts

•Theory of computation → Discrete optimization; Evolutionary algorithms; Optimization with randomized search heuristics;

Keywords

Firefly algorithm, Genetic algorithm, Swarm Optimization, Evolutionary Algorithm, Flexible job-shop scheduling problem

1. INTRODUCTION

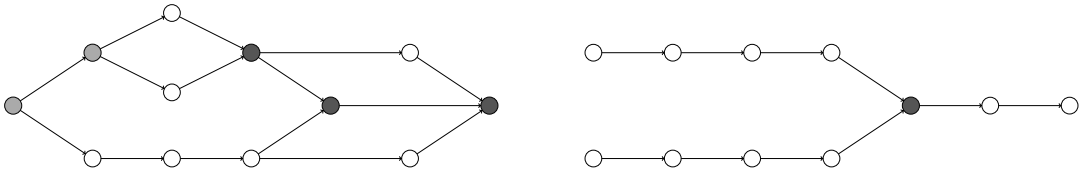
Production scheduling is one of the most important issues in the planning and scheduling of modern manufacturing systems. There are several workshop styles in the manufacturing system (including the Job Shop Scheduling Problem, JSP). The JSP can be stated as follows. Consider a set of machines and a set of jobs. Each job consists of a sequence of operations to be processed in a given order. Each operation must be processed individually on a specific machine. The objective is to find a processing sequence for each machine that minimizes an objective function, e.g. the completion time of the last operation (makespan).

The inexistence of flexibility on the resources of each operation presented on JSP may meet the requirements of a traditional manufacturing system. However, with the ushering of the fourth industrial revolution (Industry 4.0) many computing devices, flexible manufacturing systems, and numerical control machines are introduced in order to achieve a higher level of autonomously, customization, and flexibility. Therefore, the assumption that one machine only processes one type of operation works in JSP but does not reflect the reality of the modern manufacturing systems.

The Flexible Job Shop Problem (FJSP) is an extension of the classical JSP problem where is considered that there may be several machines, not necessarily identical, capable of processing an operation. Particularly, for each operation, a set of machines on which that operation can be processed is given. The goal is to decide on which machine each operation will be processed and in what order the operations will be processed on each machine so that the makespan is minimized.

In terms of computational complexity, JSP problem is known to be one of the most difficult combinatorial optimization problems [10], and has been proven to be an NP-hard problem [8]. Since the FJSP and EFJSP problems are at least as difficult as the JSP, both are also NP-hard. Many methods have been presented to solve the FJSP problem, e.g. exact methods [6, 5, 1], and heuristic methods [15, 17, 7, 11, 13, 14].

In the literature, each job in the FJSP consists of a simple sequence of operations, so-called *path-jobs*. In some industrial environments, it is common to have jobs whose operations can be processed simultaneously. Mutually independent sequences of operations may feed into an "assembling" operations. Similarly, there may be "disassembling" operations which split the sequences of subsequent operations into two or more mutually independent sequences, so-called *G-job*.



(a) G-job type. **(b) Y-job type.**

Figure 1: Types of jobs present in the FJSP with parallel operations (EFJSP).

Figure 1a shows a representation of G-job. Moreover, some jobs may consist of two independent sequences of operations followed by a third that puts together the results of the first two, so-called *Y-job*. Figure 1b shows a representation of Y-job. This problem will be referred to as "EFJSP" in this paper. Figure 1 shows the types of jobs present in the EFJSP, where each node represents an operation. The arcs represent precedence constraints and all arcs are directed from left to right. The black nodes are assembling operations and gray nodes are disassembling operations.

In recent years, several evolutionary algorithms (EA) were proposed to solve the FJSP. Recently, in [3], it was shown that hybrid techniques have been applied more often than other methods for solving FJSP. Furthermore, the technique most frequently chosen for performing exploration in hybrid algorithms is the genetic algorithm (GA). Although GA has powerful global searching ability [11], a recently developed algorithm called the Firefly Algorithm (FA), proposed in [16], has been shown [12, 16] to outperform Particle Swarm Optimization (PSO) and GA for continuous optimization problems.

In this paper, we propose and compare a Genetic Algorithm and a discrete Firefly Algorithm for solving the EFJSP. Several FJSP and EFJSP instances are used to evaluate the performance of both methods. Through experimental studies, the merits of each algorithm are demonstrated clearly. Furthermore, the proposed algorithms are compared with other state-of-the-art algorithms. The remainder of this paper is structured as follows. The problem formulation is discussed in Section 2. The solution representation for both algorithms is defined in the Section 3. The discrete FA is proposed in Section 4 and the GA is proposed in Section 5. Experimental results related to the proposed approaches are reported in Section 6. Section 7 addresses the conclusions and potential future works.

2. PROBLEM FORMULATION

Let (V, A) be a directed acyclic graph (DAG), where the vertices represent the operations, and the arcs represent precedence constraints. We are also given a set M of machines and a function F that associates a non-empty subset $F(v)$ of M with each operation v. The machines in $F(v)$ are the ones that can process operation v. Additionally, for each operation v and each machine k in $F(v)$, we are given a positive rational number p_{vk} representing the processing time of operation v on machine k. A machine assignment is a function f that assigns a machine $f(v) \in F(v)$ with each operation v. Given a machine assignment f, let $p_v^f := p_{v,f(v)}$.

For each machine k, let V_k be the set of operations that can be processed on machine k, that is, $V_k = \{v \in V : k \in F(v)\}$. Let B_k be the set of all ordered pairs of distinct elements of V_k. The pairs (v, w) in B_k are designed to prevent v and w from using machine k at the same time. Let B denote the union of all B_k. Hence, $(v, w) \in B$ if and only if $v \neq w$ and $F(v) \cap F(w) \neq \emptyset$.

Given a machine assignment f, let B^f be the set of all ordered pairs of distinct operations to be processed on the same machine, that is, $B^f = \{(v, w) \in B : f(v) = f(w)\}$. A *selection* is any subset Y of B^f such that, for each $(v, w) \in B^f$, exactly one of (v, w) and (w, v) is in Y. A selection corresponds to an ordering of the operations to be processed on the same machine. A selection Y is admissible if $(V, A \cup Y)$ is a dag.

Given a machine assignment f and a admissible selection Y, a *schedule* for $(V, A \cup Y, p^f)$ is a function s from V to the set of non-negative rational number such that $s_v + p_v^f \leq s_w$ for each (v, w) in $A \cup Y$. The number s_v is the starting time of operation v. The length of a (directed) path $(v_1, v_2, ..., v_l)$ in the dag $(V, A \cup Y)$ is the number $p_{v_1}^f + p_{v_2}^f + ... + p_{v_l}^f$. For any path P in $(V, A \cup Y)$ ending at v and any schedule s, the length of P is at most s_v. For each v in V, let s_v^* be the maximum of the lengths of all paths in $(V, A \cup Y)$ ending at v. There is a simple dynamic programming algorithm that computes the tight schedule [4]. Not surprisingly, the makespan of the tight schedule s^* is determined by the longest path: there exists a path $P = (v_1, v_2, ..., v_l, v_{l+1})$ in $(V, A \cup Y)$ such that the length of P plus $p_{v_{l+1}}$ equals $mks(s^*)$ (such P is known as a *critical path*).

Table 1: EFJSP instance with three jobs and a total of ten operations and four machines. Job 1 is a path-job, job 2 is a Y-job, and job 3 is a G-job. The column DFS represents the topological order of each respective job given by depth-first search algorithm.

Job	v	$p_{v,1}$	$p_{v,2}$	$p_{v,3}$	$p_{v,4}$	Route	DFS
1	1	2	3	4	3	$1 \rightsquigarrow 2$	
	2	3	5	2	2	$2 \rightsquigarrow 3$	$1, 2, 3$
	3	5	1	4	4	-	
2	4	4	3	4	5	$4 \rightsquigarrow 6$	
	5	3	3	4	2	$5 \rightsquigarrow 6$	$4, 5, 6$
	6	4	3	1	4	-	
3	7	3	1	3	3	$7 \rightsquigarrow 8$, 9	
	8	5	3	1	3	$8 \rightsquigarrow 10$	$7, 9, 8, 10$
	9	4	4	2	5	-	
	10	3	5	4	4	-	

$v \rightsquigarrow u$ represents that v precedes u in the job route.

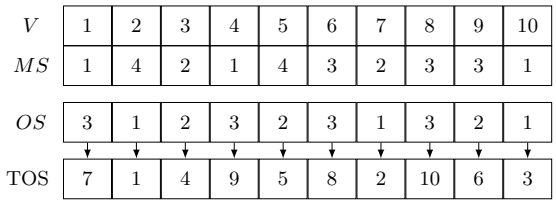


Figure 2: Representation of a EFJSP solution.

2.1 EFJSP MILP Model

The MILP model for the EFJSP proposed in [1] can be given as follows: find a rational number z, rational arrays s and p', and binary arrays x and y that

$$\begin{aligned}
&\text{Minimize } z && && \\
&\text{subject to} && && \\
&s_v + p'_v \leq z && \forall v \in V, && (1)\\
&\textstyle\sum_{k \in F(v)} x_{vk} = 1 && \forall v \in V, && (2)\\
&p'_v = \textstyle\sum_{k \in F(v)} p_{vk} x_{vk} && \forall v \in V, && (3)\\
&y_{vw} + y_{wv} \geq x_{vk} + x_{wk} - 1 && \forall k \in M, \forall (v,w) \in B_k, && (4)\\
&s_v + p'_v \leq s_w && \forall (v,w) \in A, && (5)\\
&s_v + p'_v - (1 - y_{vw})L \leq s_w && \forall (v,w) \in B, && (6)\\
&s_v \geq 0 && \forall v \in V. && (7)
\end{aligned}$$

As x is binary, constraint (2) ensures that x is a machine assignment. Then constraint (3) makes array p' represent the processing times of operations. In fact, p' can be seen as an intermediate value, not a variable, that helps to simplify the presentation of the model. Since $p_{v,k} > 0$ for all v and k, thus $p'_v > 0$ and so constraint (6) makes sure that y_{vw} and y_{wv} are not both equal to 1. Hence, as y is binary, constraint (4) implies that y represents a selection. Indeed, if $x_{vk} = x_{wk} = 1$, which means v and w are assigned to machine k, then (4) forces y to decide whether v comes before or after w. Otherwise, constraint (4) is trivially satisfied. Once y is a selection and p' represents the processing times, constraints (5), (6), and (7) make s represent a schedule. Finally, the objective function and constraint (1) make sure z is the makespan of the schedule, and is as small as possible. Finally, L is an upper bound on the makespan of an optimal solution of the FJSP problem.

3. ENCODING AND DECODING

Since the FJSP is composed of two sub-problems, its solution representation is composed of two strings, i.e., machine assignment string (MS) and operation sequence string (OS). The MS string denotes the assigned machine for each particular operation and the OS string represents the order in which the operations are to be processed in their assigned machines. We denote ζ_{ime} as the eth element of the mth string of the ith individual (or firefly), where ζ_{i1} and ζ_{i2} are respectively the MS string and the OS string. Figure 2 shows a solution for the given set of machines and operations presented in Table 1. The solution ζ_i is composed of the MS string (ζ_{i1}) and the OS string (ζ_{i2}). The OS is translated onto TOS string and it maps the appearances of job id and operation.

Algorithm 1 Longest path $l(0, *)$

```
T ← topological sort of V
for each v ∈ T do
    E ← out-edges of v
    for each ε ∈ E do
        u ← target vertex of ε
        if s_u < s_v + p_{v,f(v)} then
            s_u ← s_v + p_{v,f(v)}
        end if
    end for u
end for v
```

3.1 Machine Assignment

Each element of the MS string is associated with a unique operation and the eth element represents the selected machine k for the corresponding operation V_e, where the index V_e denotes the eth operation in V and $k \in F(V_e)$. For example, on Figure 2, ζ_{i13} expresses that the machine 2 is assigned for operation 3. The length of the MS string is equal to $\sum_{i=1}^{n} J_i$, where J_i is the number of operations of job i and n is the number of jobs. The index does not vary throughout the whole searching process. Figure 2 shows an example of MS strings based on the given set of machines and operations presented in Table 1.

3.2 Operation Sequence

The OS string represents the order in which the operations will be processed in their designated machines. To avoid repair mechanisms, this representation uses an unpartitioned permutation with J_i repetitions of the job numbers, i.e., the index of job i appear in the string J_i times. Figure 2 displays an example of OS strings based on the jobs shown in Table 1.

3.3 Translating the OS String

By scanning the OS string from left to right, the f_ith appearance of a job i refers to the f_ith operation in topological order of the operations of job i. For example, scanning the OS string from left to right, for every appearance of job i, f_i is increased by 1, and the f_ith operation on the topological order is added to the TOS string. The translation mechanism bypasses the use of repair mechanism since any permutation can be decoded into a DAG, leading to a feasible solution. For the instance shown in Table 1, one possible OS string, and its TOS string is presented in Figure 2.

3.4 Calculation of the Makespan

Following the translation of the OS string and the assignment based on the MS strings, the starting time of every operation has to be defined in order to achieve the makespan (mks) of the solution. The calculation of the mks can be done by using graph traversal algorithms, commonly used in temporal planning. Additionally to the notations given in Section 2, two dummy nodes are introduced. In this way, the set of vertices is $\underline{V} = V \cup \{0, *\}$, 0 being the source and *

Table 2: Movement update of ζ_i towards a brighter firefly ζ_j.

	MS string	OS string
Firefly ζ_j	1 4 2 1 4 3 2 3 3 1	3 1 2 3 2 3 1 3 2 1
Firefly ζ_i	2 4 3 1 2 3 2 3 1 1	3 2 1 3 2 3 1 2 1 3
H_{ij} and S_{ij}	{1, 3, 5, 9}	{(2,3), (8,10), (9,10)}
$\underline{d}_{ij}$ and $\overline{d}_{ij}$	$\|2-1\|+\|3-2\|+\|2-4\|+\|1-3\|=6$	$\|2-3\|+\|8-10\|+\|9-10\|=4$
Attractiveness $\beta(r)$	0.217	0.384
$\underline{R}$ and $\overline{R}$	{0.13, 0.78, 0.17, 0.24}	{0.35, 0.72, 0.11}
Position after β-step	**1** 4 3 1 **3** 3 2 3 1 1	3 **1 2** 3 2 3 1 2 **3 1**
Position after α-step	1 4 3 1 3 3 2 3 1 **4**	3 1 2 **2 3** 3 1 2 3 1

being the sink node, respectively representing the start and the end of the planning period.

The length of a path is defined as $\sum_{i=1}^{q-1}\mu(v_i,v_{i+1})^f$, where q is the number of nodes in the path, v_i is the ith node, and $\mu(v,u)$ gives maximal length of all edges between node v and u. Denoting the value of some longest path from node v to node u by $l(v,u)$, the starting time of operation v_i in a left justified schedule is equal to $l(0,v_i)$ in the corresponding solution graph. Therefore, the mks of a solution is equal to the length of some longest path from 0 to *, $l(0,*)$. The pseudo code shown in Algorithm 1 summarizes the steps to calculate the longest path $l(0,*)$.

4. FIREFLY ALGORITHM

Firefly algorithm (FA) is a simple yet quite efficient nature-inspired search technique for global optimization. Since FA was developed, it has attracted a lot of attentions and becomes more popular in solving various real-world problems. FA is a swarm-based intelligence algorithm, which mimics the flashing behavior of fireflies [16]. A firefly flashes as a signal to attract others for some purposes, e.g. predating or mating. Accordingly, this biological phenomenon is formulated as a meta-heuristic algorithm depending on following three rules [16]:

- One firefly will be attracted to other fireflies regardless of their sex;
- The attractiveness is proportional to the brightness and decrease as their distance increases. For any two flashing fireflies, the less bright one will move towards the brighter one;
- The brightness of a firefly is determined by the landscape of the objective function.

Algorithm 2 Firefly Algorithm

```
Objective function f(t), t = (ζ_1, ..., ζ_d)^T
Generate initial pop. P of fireflies
Light intensity I_i = f(ζ_i)
Define light absorption coefficient γ
while (t < MaxGeneration) do
    for each ζ_i ∈ P do
        for each ζ_j ∈ P do
            if (I_i < I_j) then Move ζ_i towards ζ_j end if
            Vary β with distance r via exp[−γr]
            Evaluate solutions and update light intensity
        end for j
    end for i
    Rank fireflies and find the current global best
end while
```

The pseudocode shown in Algorithm 2 summarizes the basic steps of the FA which consists of the three addressed rules.

4.1 Classic Firefly Algorithm

In the firefly algorithm, there are two important issues: the variation of light intensity and formulation of the attractiveness. For simplicity, we can always assume the attractiveness of a firefly is determined by its brightness, which in turn is associated with the encoded objective function.

The attractiveness function $\beta(r)$ can be any monotonically decreasing functions such as the following generalized form

$$\beta(r)=\beta_0 e^{-\gamma r^m},\quad m\geqslant 1, \tag{8}$$

where β_0 is the attractiveness at $r=0$, and r is the distance between two fireflies. As it is often faster to calculate $1/(1+r^2)$ than an exponential function [16], the Equation (8) can be approximated as

$$\beta(r)=\frac{\beta_0}{1+\gamma r^2}. \tag{9}$$

The distance between any two fireflies i and j, at position x_i and x_j, respectively can be defined as a Cartesian distance:

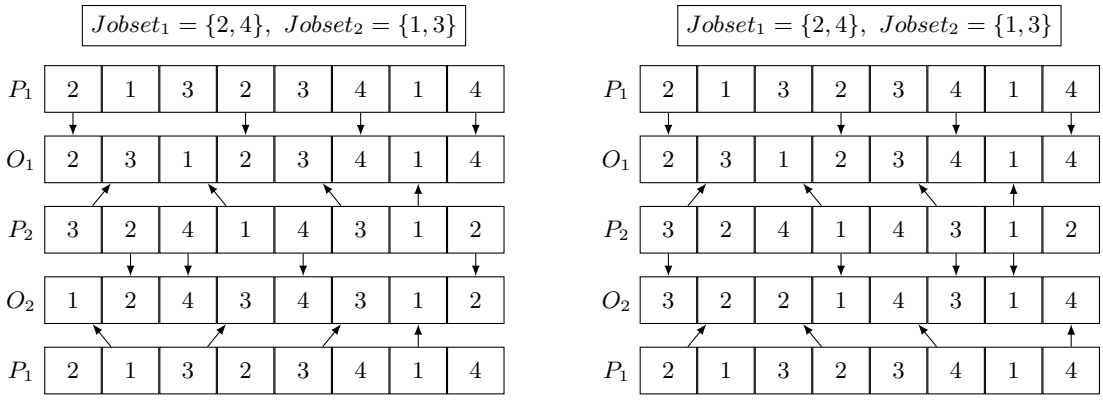
$$r_{\zeta_i\zeta_j}=\|x_{\zeta_i}-x_{\zeta_j}\|=\sqrt{\sum_{k=1}^{d}(x_{\zeta_i k}-x_{\zeta_j k})^2}, \tag{10}$$

where $x_{\zeta_i k}$ is the kth component of the spatial coordinate x of the ζ_ith firefly.

The random movement of a firefly i towards another more brighter firefly j is determined by

$$x_{\zeta_i}=x_{\zeta_i}+\beta_0 e^{-\gamma r^2_{\zeta_i\zeta_j}}(x_{\zeta_i}-x_{\zeta_j})+\alpha\,\epsilon_{\zeta_i}, \tag{11}$$

where the second term considers a firefly's attractiveness, the third term is randomization with α being the randomization parameter, and ϵ_i is a vector of random numbers drawn from a Gaussian distribution or uniform distribution. In a simplest form, ϵ_i can be replaced by $x-1/2$, where x is a random number uniformly distributed in [0,1]. For most applications we can take $\beta_0=1$ and $\alpha\in[0,1]$.



(a) POX crossover **(b) JBX crossover**

Figure 3: Crossover operators for OS string.

4.2 Discrete Firefly Algorithm for the FJSP

The FA has been originally developed for solving continuous optimization problems and cannot be directly applied to solve discrete optimization problems. The main challenges for using the FA to solve combinatorial optimization problems are the calculation of the discrete distance between two fireflies, and how they move in the coordination.

4.3 Distance

The discrete distance between two fireflies is defined by the distance between the permutation of its strings. The discretization of the distance, assimilation and revolution operators are described in details below.

4.4 Machine Assigment Distance

To calculate the distance between two MS strings, first, we apply the Hamming distance to verify positions (indexes) of the non-corresponding elements (values) of the strings, i.e. if $\zeta_{i1e} \neq \zeta_{j1e}$, $\forall e \in V$, then the index V_e is store in H_{ij}, $\{V_e \in H_{ij} : \zeta_{i1e} \neq \zeta_{j1e}\}$, where V_e is the eth element in V. Afterwards, these positions are used to calculate the MS distance as

$$\underline{d}_{ij} = \sum\nolimits_{\forall e \in H_{ij}} \left| m(\zeta_{i1e}, F(V_e)) - m(\zeta_{j1e}, F(V_e)) \right|, \quad (12)$$

where m is a function that maps the index of the machine ζ_{i1e} in the set F of the operation V_e. To illustrate Equation (12), consider strings $\zeta_{1,1} = \{5, 3, 1\}$ and $\zeta_{2,1} = \{1, 3, 4\}$, therefore $H_{1,2} = \{1, 3\}$. Supposing that $F(V_1) = F(V_3) = \{1, 3, 4, 5\}$, therefore $\underline{d}_{1,2} = |4 - 1| + \left|(1 - 3)\right|$.

4.5 Operation Sequence Distance

To determine the distance between any two OS strings we apply the so-called Swapping distance. The Swapping distance is the minimal number of swaps needed to perform in a string in order to produce another. In our implementation, every swap is considered a pair, where the first and second values in each pair designate the positions of the elements required to be swapped in order to advance one string towards another. The minimal number of swaps in order to move ζ_{i2} towards ζ_{j2} is stored in pair-wise fashion in S_{ij}. Following the calculation of the minimal amount of swaps, the distance between two OS strings can be determined as

$$\overline{d}_{ij} = \sum\nolimits_{\forall p \in S_{ij}} |p_1 - p_2|, \quad (13)$$

where p_1 and p_2 are respectively the first and second value of the pth pair. To illustrate Equation (13), consider $\zeta_{1,2} = \{2\ 3\ 1\}$ and $\zeta_{2,2} = \{1\ 3\ 2\}$, $S_{1,2} = \{\underline{(1, 3)}\}$, where $\underline{(1, 3)}$ means that ζ_{i21} should be swapped with ζ_{i23}, therefore $\overline{d}_{1,2} = 2$.

4.6 Moving Towards Another Firefly

In this study we split the movement into two sub-steps: assimilation β-step and revolution α-step. The steps β and α are not interchangeable, thereby, β-step must be computed before α-step while finding the new position. Both steps are described in details in Section 4.7 and Section 4.8. Both steps are illustrated in details on Table 2, where the firefly ζ_i updates its position towards a brighter firefly ζ_j. The parameters used in this illustration are $\beta_0 = 1, \gamma = 0.1$. The machines, jobs, operations, and topology considered in this illustration are based on the values shown in Table 1.

The β-step brings the iterated firefly closer to the brighter firefly by moving the MS string and the OS string towards the strings of the brighter firefly. Sections 4.7 and 4.8 explains in details the β-step. The α-step allows shifting the permutation into one of the neighboring permutations. Section 4.9 explains the α-step in details.

4.7 Moving the MS String

For every element e in H_{ij} a random number $x \in \underline{X}$ is generated uniformly distributed over $[0, 1]$, where $\underline{X}_e$ is the eth random number for eth element of H_{ij}. In the case that $\underline{X}_e \leq \beta(\underline{d}_{ij})$, the distance of the element ζ_{i1e} respective to ζ_{j1e} is decreased by

$$floor\Big((\beta(\underline{d}_{ij}) \times \left|m\big(\zeta_{i1e}, F(V_e)\big) - m\big(\zeta_{j1e}, F(V_e)\big)\right|) + 1\Big). \quad (14)$$

4.8 Moving the OS String

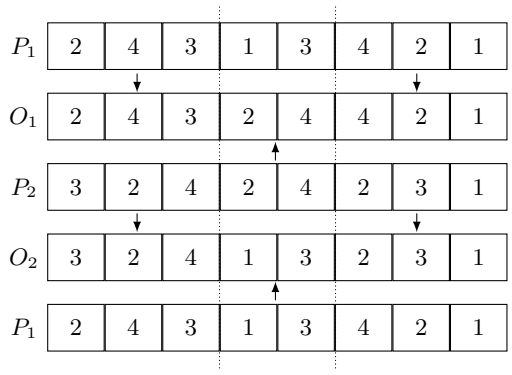


Figure 4: Two-point crossover.

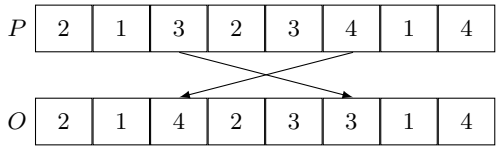


Figure 5: Swapping mutation.

For every pair p in S_{ij} a random number $x \in \overline{X}$ is generated uniformly distributed over $[0, 1]$, where $\overline{X}_p$ is the pth random number for pth pair of S_{ij}. In case that $\overline{X}_p \leq \beta(\overline{d}_{ij})$, the value in ζ_{i2p_1} is swapped with the value in ζ_{i2p_2}.

4.9 Random Movement

The α-step is performed on both strings as follows: (1) an element $e \in \zeta_{i1}$ is chosen at random and a new machine $k \in F(V_e)$ is assigned at random; (2) elements $e \in \zeta_{i2}$ and $e' \in \zeta_{i2}$ are chosen at random and swapped.

5. GENETIC ALGORITHM

Genetic algorithm (GA) is a class of algorithms based on the abstraction of Darwinian evolution of biological systems. Starting from an initial population, the algorithm applies genetic operators in order to produce offspring. At each generation, every new individual corresponds to a solution, i.e., a schedule of the given FJSP instance. The overall structure of our GA can be described as follows:

- *Coding:* the genes of the chromosomes describe the assignment of operations to the machines, and the operations sequences, as presented in Section 3.
- *Initial population:* the initial chromosomes are obtained by random permutation of jobs and machines.
- *Fitness evaluation:* the makespan is computed for each individual in the current generation as described in Section 3.4.
- *Selection:* the selection procedure is described in Section 5.1.
- *Offspring generation:* the new generation is obtained by changing the assignment of the operations and by changing the sequencing of operations. Each string is subjected to its own genetic operators. The crossover and mutation operators are discussed in Sections 5.2 and 5.5.
- *Stop criterion:* fixed number of generations is reached.

Table 3: Experiments with the YFJS instances. These instances are composed of Y-jobs.

Instance	Size	BRG		GA				FA			
		mks	CPU	*mks*	Mean	StDev	CPU	*mks*	Mean	StDev	CPU
YFJS01	4, 10, 7	773	11.5	773	773	0.0	3.98	773	773.0	0.0	2.16
YFJS02	4, 10, 7	825	9.88	825	825	0.0	6.49	825	825.0	0.0	2.89
YFJS03	6, 4, 7	347	3.72	347	348.8	3.6000	7.19	347	347.0	0.0	4.59
YFJS04	7, 4, 7	390	7.82	390	390	0.0	7.64	390	390.0	0.0	4.97
YFJS05	8, 4, 7	445	357.55	445	447.8	3.4292	12.87	445	447.1	3.2078	5.49
YFJS06	9, 4, 7	[425.29;449]	3600	447	447.5	1.0246	13.94	447	447.0	0.0	6.44
YFJS07	9, 4, 7	444	1392	444	445.2	3.6000	13.19	444	444.0	0.0	5.49
YFJS08	9, 4, 12	353	0.67	353	356.3	5.0408	11.49	353	353.0	0.0	4.88
YFJS09	9, 4, 12	242	14.03	242	243.8	2.7495	17.95	242	242.0	0.0	7.56
YFJS10	10, 4, 12	399	4.03	399	399.0	0.0	12.31	399	399.0	0.0	5.94
YFJS11	10, 5, 10	526	177.43	526	527.2	1.4696	37.56	526	526.9	1.3747	21.55
YFJS12	10, 5, 10	512	3218.89	512	512.4	0.4898	44.11	512	512.4	0.4898	28.74
YFJS13	10, 5, 10	405	1624.66	405	405.4	0.8000	174.12	405	405.2	0.6000	69.98
YFJS14	13, 17, 26	1317	3293.58	1317	1317.4	0.4898	186.22	1317	1317.2	0.4000	71.56
YFJS15	13, 17, 26	[1239;1244]	3600	1242	1246.6	6.2000	158.38	1239	1243.5	6.4691	75.23
YFJS16	13, 17, 26	[1200;1245]	3600	1222	1233.6	8.2849	135.32	1222	1225.2	2.1354	56.97
YFJS17	17, 17, 26	[1133;2379]	3600	1139	1141.5	4.2720	249.82	1134	1135.8	2.2271	105.32
YFJS18	17, 17, 26	[1220;2082]	3600	1230	1281.1	23.0844	305.13	1226	1237.2	10.9526	129.45
YFJS19	17, 17, 26	[926;1581]	3600	948	959.5	7.7103	326.79	943	952.3	6.8709	149.62
YFJS20	17, 17, 26	[968;2312]	3600	1001	1010.4	10.3169	311.33	974	978.0	2.0000	127.61

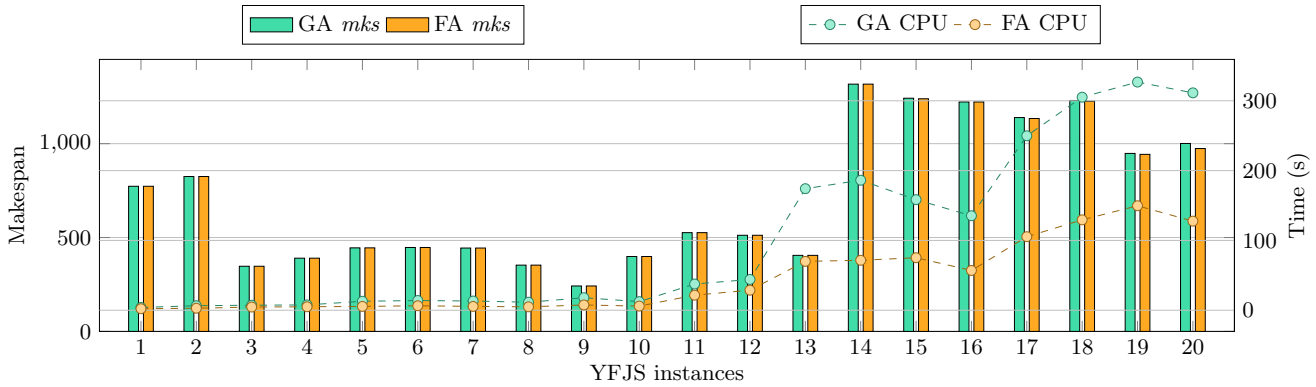


Figure 6: Best makespan and mean CPU time for the experiment with the YFJS instances.

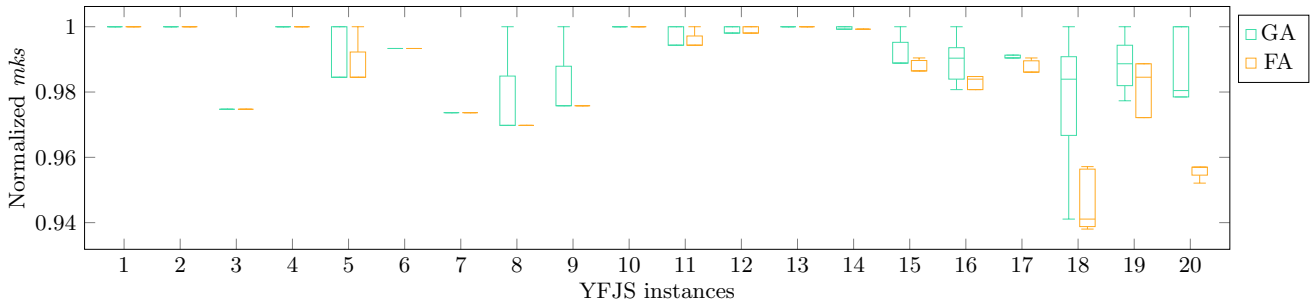


Figure 7: Box plot of the makespan obtained with the experiments involving the YFJS instances.

5.1 Selection

In GAs, the selection operator is used to select the individuals according to their fitness and maintain the highest quality chromosomes and characteristics within the population. In our algorithm, the selection strategy includes two parts: the method of keeping the best individuals and tournament selection. The method of keeping the best individuals is to copy the 1% of the best individuals for the next generation. The tournament selection strategy, proposed in [9], works as follows: two solutions are selected randomly as the parent solutions, if a random number generated between 0 and 1 is smaller than the probability r which usually is set to 0.8, then the better one is selected; otherwise, the worst one is selected.

5.2 Crossover

Crossover is the recombination of two parent chromosomes through the exchange of a part of one chromosome with a corresponding part of another in order to produce offspring. We denote the crossover probability as p_c. In this paper, two crossover operators: (a) precedence operation crossover (POX), proposed in [18]; (b) job-based crossover (JBX), proposed in [19]; are adopted for the OS string. During the OS string crossover procedure, one crossover operator is selected randomly.

5.3 Operation Sequence Crossover

The basic working procedure of POX is described as bellow (two parents are denoted as P_1 and P_2; two offspring are denoted as O_1 and O_2). Figure 3 shows an example of POX crossover operator.

1. The Job set $J = \{J_1, J_2, J_3, ..., J_n\}$ is divided into two groups $Jobset_1$ and $Jobset_2$ randomly;

2. Any element in P_1 which belongs to $Jobset_1$ are appended to the same position in O_1 and deleted in P_1; any element in P_2 which belongs to $Jobset_1$ are ap-

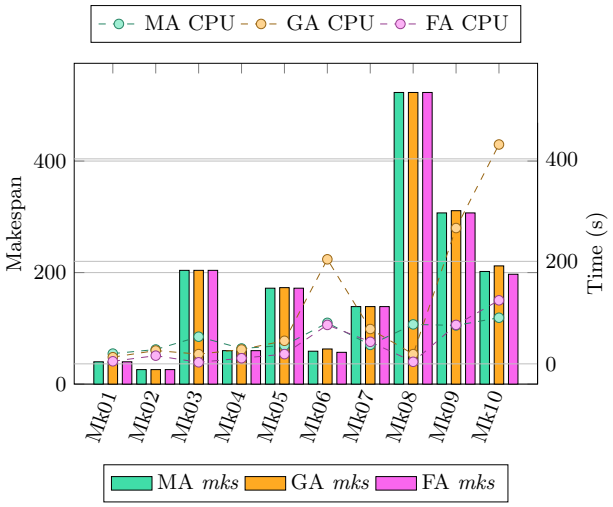


Figure 8: Experiment with FJSP instances.

Table 4: Experiment with the DAFJS instances. These instances are composed of G-jobs.

Instance	Size	BRG		GA				FA			
		mks	CPU	*mks*	Mean	StDev	CPU	*mks*	Mean	StDev	CPU
DAFJS01	4, 5-9,5	257	78.93	257	257.3	0.4714	10.33	257	257.0	0.0	4.95
DAFJS02	4, 5-7, 5	289	1271.7	289	290.7	1.1235	12.87	289	289.5	0.8844	5.46
DAFJS03	4, 10-17, 10	576	15.8	576	582.6	13.2	15.73	576	576.0	0.0	10.53
DAFJS04	4, 9-14, 10	606	1.22	606	606.0	0.0	31.55	606	606.0	0.0	9.17
DAFJS05	6, 5-13, 5	[347.53;403]	3600	395	398.5	2.7293	29.75	389	395.2	4.7638	7.23
DAFJS06	6, 5-13, 5	[326;435]	3600	416	421.3	3.2139	33.49	412	417.0	3.2455	6.16
DAFJS07	6, 7-23, 10	[497;562]	3600	519	524.7	4.0277	45.32	512	517.9	4.2968	9.43
DAFJS08	6, 6-23, 10	[628;631]	3600	628	633.1	5.1623	40.42	628	632.1	3.660	17.51
DAFJS09	8, 4-9, 5	[315;475]	3600	470	473.6	3.4020	35.16	464	469.4	1.9253	11.97
DAFJS10	8, 4-11, 5	[336;575]	3600	534	542.0	3.8471	49.62	533	538.2	5.2051	20.15
DAFJS11	8, 10-23, 10	[658;708]	3600	659	664.5	4.4252	110.64	659	662.1	2.205	55.43
DAFJS12	8, 9-22, 10	[530;720]	3600	652	656.6	4.0792	135.64	645	651.1	3.9643	66.48
DAFJS13	10, 5-11, 5	[304;718]	3600	662	666.5	3.5188	106.73	656	660.4	3.3823	46.98
DAFJS14	10, 4-10, 5	[358.95;860]	3600	736	748.2	9.4671	115.39	735	741.7	6.0955	51.97
DAFJS15	10, 8-19, 10	[512;818]	3600	677	686.8	5.7643	155.61	671	681.5	6.1738	71.95
DAFJS16	10, 6-20, 10	[640;819]	3600	679	688.9	7.9872	125.33	679	684.1	4.2719	61.73
DAFJS17	12, 4-11, 5	[300;909]	3600	804	813.4	6.2268	151.13	800	806.7	4.5821	59.49
DAFJS18	12, 5-9, 5	[322;951]	3600	803	808.2	4.3543	119.64	799	803.3	4.7703	40.95
DAFJS19	8, 7-13, 7	[512;592]	3600	525	527.5	2.3343	103.43	524	525.4	1.9596	49.27
DAFJS20	10, 6-17, 7	[434;815]	3600	712	720.9	5.3971	119.78	705	710.3	3.4538	69.72
DAFJS21	12, 5-16, 7	[504;965]	3600	815	819.9	5.1441	164.85	808	811.7	2.5157	75.81
DAFJS22	12, 5-17, 7	[464;902]	3600	720	724.5	5.3275	168.53	708	713.9	3.2958	85.49
DAFJS23	8, 6-17, 9	[450;538]	3600	476	480.5	2.9635	109.71	476	479.9	2.6297	61.51
DAFJS24	8, 6-25, 9	[476;666]	3600	568	572.8	3.4871	135.13	564	567.2	2.8566	55.32
DAFJS25	10, 9-19, 9	[584;897]	3600	755	758.5	3.5752	119.55	752	757.7	4.8397	69.84
DAFJS26	10, 8-17, 9	[565;903]	3600	751	756.9	6.1413	188.64	745	748.6	2.9394	78.81
DAFJS27	12, 7-22, 9	[503;981]	3600	838	845.1	6.9557	193.56	831	835.0	2.7568	81.55
DAFJS28	8, 8-15, 10	[535;671]	3600	545	552.9	4.3889	129.91	543	547.2	2.4000	60.48
DAFJS29	8, 7-19, 10	[609;726]	3600	657	663.7	4.2026	131.14	654	656.3	2.4129	61.41
DAFJS30	10, 8-19, 10	[467;656]	3600	557	564.9	5.5554	137.11	555	558.3	1.8856	65.36

pended to the same position in O_2 and deleted in P_2;

3. the remaining elements in P_2 are appended to the remaining empty positions in O_1 seriatim; and the remaining elements in P_1 are appended to the remaining empty positions in O_2 seriatim.

The second crossover operator for OS string is the job-based crossover (JBX). The basic working procedure of JBX is described below. Figure 3 shows an example of JBX crossover operator.

1. The Job set $J = \{J_1, J_2, J_3, ..., J_n\}$ is divided into two groups $Jobset_1$ and $Jobset_2$ randomly;

2. Any element in P_1 which belongs to $Jobset_1$ are appended to the same position in O_1; any element in P_2 which belongs to $Jobset_2$ are appended to the same position in O_2;

3. Any element in P_2 which belongs to $Jobset_2$ are appended to the remaining empty positions in O_1 seriatim; and any element in P_1 which belongs to $Jobset_1$ are appended to the remaining empty positions in O_2 seriatim.

5.4 Machine Selection Crossover

For the MS string, a two-point crossover has been adopted as the crossover operation. In this operation, two positions are selected at random. Based on the selected positions, two children strings are created by swapping all elements between the positions of the two parent strings. Figure 4 shows an example of two-point crossover.

5.5 Mutation

Mutation operator is applied on a single chromosome for the purpose of changing a gene at its respective location. The gene 1011 can be mutated as 1111, as the gene at location 2 is flipped from 0 to 1. The mutation operator is used to change some information in a selected chromosome or diversify the solution space for further exploration. The

Table 5: Parameters of the GA and FA.

Instance	F/I	$T(fa)/T(ga)$	γ	p_c	p_m
Small	100/250	400/300	0.10	0.85	0.02
Medium	250/500	800/500	0.10	0.85	0.02
Large	500/750	1200/750	0.10	0.85	0.02

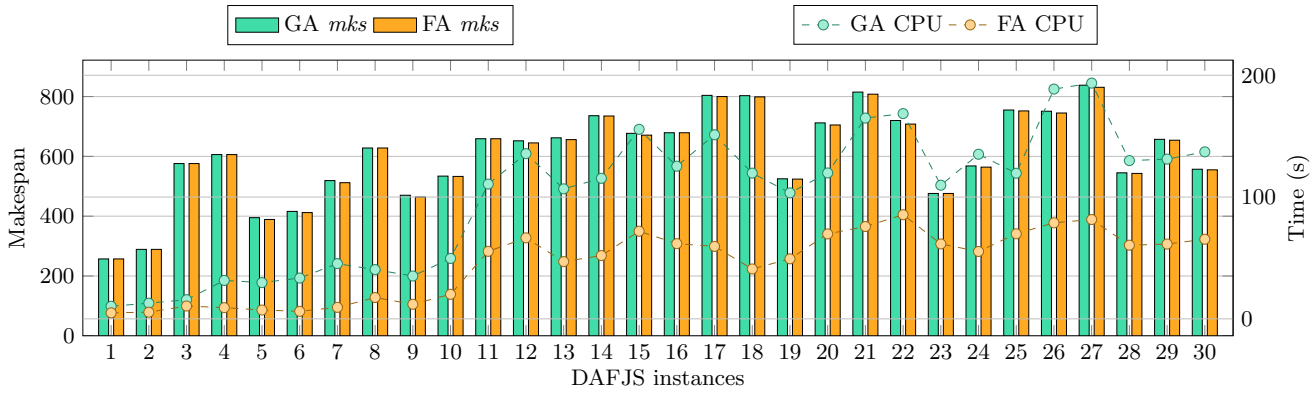


Figure 9: Best makespan and mean CPU time for the experiment with the DAFJS instances.

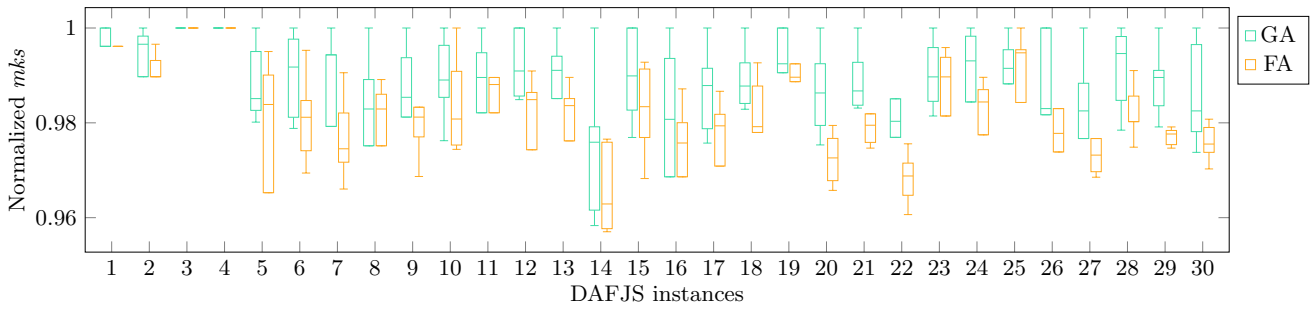


Figure 10: Box plot of the makespan obtained with the experiments involving the DAFJS instances.

mutation probability p_c should be small since a high probability will be adverse for the information preservation of the good chromosomes. In this paper, the swapping mutation [19], has been adopted for the OS string. In this procedure, two positions are selected and its respective elements are swapped. A single point mutation is used for the MS string. In this procedure, a position of the MS string is selected, and a new machine is assigned for its respective operation. Figure 5 shows an example of swapping mutation.

6. NUMERICAL RESULTS

This section presents the results of computational experiments involving the algorithms. We used as benchmarks 10 FJSP instances (named Mk) introduced in [2], 20 EFJSP instances composed of Y-jobs (named YFJS) proposed in [1], and 30 EFJSP instances composed of G-jobs (named DAFJS) further proposed in [1]. Experiments involving the metaheuristics were conducted on the same computer, an Intel Core i7 2.70GHz. The computer used for the MILP model has an Intel Xeon E5440 2.83GHz processor. The proposed algorithms were implemented in C++. A supplementary material (e.g., the instances used in this work) is provided at `https://willtl.github.io/acr2018_summer/`.

Based on experiments we find out the best parameters for the GA and FA based on the size of the instances. We divided into three categories (i.e., small, medium, large), and for each category. Small instances, i.e., less or equal to 6 jobs and 5 machines; medium instances, i.e., less or equal to 10 jobs and 8 machines; larger instances, i.e., instances that does not belong to another group. The number of individuals, fireflies, and the number of generations are shown in Table 5, where F is the number of fireflies, I is the number of individuals, T is a function that gives the number of generations for each algorithm. The attraction coefficient was kept as $\beta_0 = 1.0$ for all experiments.

Figure 8 presents the results for the FJSP instances proposed in [2]. The bars denote the *mks* and the lines represent the CPU time, in seconds. This set of instances include only path-jobs. We analyze our proposed methods with another state-of-the-art algorithm, a memetic algorithm (MA) proposed in [17]. The MA was implemented on an Intel Core i7-3520M 2.9 GHz processor in Java.

We experiment our algorithms with two sets of EFJSP instances proposed in [1]. Figure 6 and Figure 9 shows the best *mks* and CPU time for the experiments with the YFJS and DAFJS instances. Figure 7 and Figure 10 shows the variation of observed data through quartiles for the experiments with the YFJSP and DAFJS instances. Tables 3 and 4 present the numerical results for the experiments involving the YFJSP and DAFJS instances. The instance column designates the names of the instances. For the MILP model [1] (we name it BRG), the *mks* column designates the optimal makespan or the lower and upper bounds found by CPLEX. The CPU column records the CPU time in seconds. The CPLEX was limited to 3600 seconds. Regarding the metaheuristics, for each instance, the experiment was performed 25 times. Therefore, column *mks* designates the best-obtained makespan, column Mean records the mean of

the best-obtained values, column StDev records the standard deviation, and the column CPU designates the mean CPU times.

On the experiment among the [2] instances, we can perceive that the FA was able to find the optimal solution for most of the instances except the two largest instances (i.e. Mk06 and Mk10). The FA is more efficient related to the GA and MA, however, the MA was more efficient for the MK10. Based on the investigations with Y-job and G-job instances given on Tables 3 and 4, we can see that both the GA and FA loses efficiency for larger instances due to the fact that the parameters had to be adjusted in order to increase the searching capabilities of the algorithms. In that case, even with readjusted parameters (e.g., a higher number of individuals, fireflies, and iterations), the GA was not able to obtain similar solutions to those that FA found for the largest instances. The MILP model is more efficient for smaller instances, and as expected, it does not scale well for larger instances.

Here is a summary of our results. The proposed FA is more effective and efficient than the proposed GA. As expected, both the GA and FA are more efficient than the MILP. Based on the bounds given by the MILP model, we can see that the FA decreased the gap (i.e. the distance between lower and upper bounds) for several instances.

7. CONCLUSION

In the present paper, we extended the definition of the FJSP taking into account parallel operations in the route of the jobs. We put forward a discrete firefly algorithm and a genetic algorithm to solve the EFJSP. In order to evaluate the performance of the solution methods, 50 EFJSP instances were used in the computational experiments. Furthermore, 10 famous FJSP instances were used to provide comparisons with other state-of-the-art algorithms. To evaluate the performance of the proposed algorithms, a MILP model was used to provide optimal solutions or solution bounds. The experiments with the GA and FA shows that both are possible approach for the considered problem.

8. REFERENCES

[1] E. G. Birgin, P. Feofiloff, C. G. Fernandes, E. L. De Melo, M. T. Oshiro, and D. P. Ronconi. A milp model for an extended version of the flexible job shop problem. *Optimization Letters*, 8(4):1417–1431, 2014.

[2] P. Brandimarte. Routing and scheduling in a flexible job shop by tabu search. *Annals of Operations research*, 41(3):157–183, 1993.

[3] I. A. Chaudhry and A. A. Khan. A research survey: review of flexible job shop scheduling techniques. *International Transactions in Operational Research*, 23(3):551–591, 2016.

[4] T. H. Cormen. *Introduction to algorithms*. MIT press, 2009.

[5] Y. Demir and S. K. İşleyen. Evaluation of mathematical models for flexible job-shop scheduling problems. *Applied Mathematical Modelling*, 37(3):977–988, 2013.

[6] P. Fattahi, M. S. Mehrabad, and F. Jolai. Mathematical modeling and heuristic approaches to flexible job shop scheduling problems. *Journal of intelligent manufacturing*, 18(3):331, 2007.

[7] K. Z. Gao, P. N. Suganthan, T. J. Chua, C. S. Chong, T. X. Cai, and Q. K. Pan. A two-stage artificial bee colony algorithm scheduling flexible job-shop scheduling problem with new job insertion. *Expert systems with applications*, 42(21):7652–7663, 2015.

[8] M. R. Garey, D. S. Johnson, and R. Sethi. The complexity of flowshop and jobshop scheduling. *Mathematics of operations research*, 1(2):117–129, 1976.

[9] D. E. Goldberg and K. Deb. A comparative analysis of selection schemes used in genetic algorithms. *Foundations of genetic algorithms*, 1:69–93, 1991.

[10] E. L. Lawler, J. K. Lenstra, A. H. R. Kan, and D. B. Shmoys. Sequencing and scheduling: Algorithms and complexity. *Handbooks in operations research and management science*, 4:445–522, 1993.

[11] X. Li and L. Gao. An effective hybrid genetic algorithm and tabu search for flexible job shop scheduling problem. *International Journal of Production Economics*, 174:93–110, 2016.

[12] M. Lohrer. *A comparison between the firefly algorithm and particle swarm optimization*. PhD thesis, 2013.

[13] W. T. Lunardi, L. H. Cherri, and H. Voos. A mathematical model and a firefly algorithm for an extended flexible job shop problem with availability constraints. In *International Conference on Artificial Intelligence and Soft Computing*, pages 548–560. Springer, 2018.

[14] W. T. Lunardi and H. Voos. Comparative study of genetic and discrete firefly algorithm for combinatorial optimization. In *33rd Symposium On Applied Computing (SAC), 2018*, pages 1–8. ACM/SIGAPP, 2018.

[15] G. Vilcot and J.-C. Billaut. A tabu search and a genetic algorithm for solving a bicriteria general job shop scheduling problem. *European Journal of Operational Research*, 190(2):398–411, 2008.

[16] X.-S. Yang. *Nature-inspired metaheuristic algorithms*. Luniver press, 2010.

[17] Y. Yuan and H. Xu. Multiobjective flexible job shop scheduling using memetic algorithms. *IEEE Transactions on Automation Science and Engineering*, 12(1):336–353, 2015.

[18] C. Zhang, P. Li, Y. Rao, and S. Li. A new hybrid ga/sa algorithm for the job shop scheduling problem. *Evolutionary computation in combinatorial optimization*, pages 246–259, 2005.

[19] G. Zhang, L. Gao, and Y. Shi. An effective genetic algorithm for the flexible job-shop scheduling problem. *Expert Systems with Applications*, 38(4):3563–3573, 2011.

ABOUT THE AUTHORS:

Willian Tessaro Lunardi received his M.S. degree in computer science from the Pontifical Catholic University of Rio Grande do Sul, Brazil, in 2016. Currently, he is a Ph.D. candidate at the University of Luxembourg, Automation & Robotics Research Group. His current research interests include optimization techniques, approximate algorithms, planning and scheduling, and metaheuristics.

Holger Voos received his Ph.D. degree in electrical engineering from the University of Kaiserslautern, Germany, in 2000. Currently, he is a professor at the University of Luxembourg and head of the Automatic Control Laboratory. From 2004 to 2010, he served as a professor at the University of Applied Sciences Ravensburg-Weingarten, Germany. He was the head of the Laboratory of Mechatronics and Mobile Robotics, chairman of the international Master program mechatronics, and a member of the Board of Governors. From 2000 to 2004, he worked as a systems engineer and a project manager in the research and development department at Bodenseewerk Geretechnik GmbHerlingen, Germany (now: Diehl-BGT-Defence GmbH). His current research interests include optimization techniques, distributed and networked control, autonomous robots, safety, and mechatronic systems.