An Imperialist Competitive Algorithm for a Real-World Flexible Job Shop Scheduling Problem

Willian Tessaro Lunardi*, Holger Voos* and Luiz Henrique Cherri†
* Interdisciplinary Centre for Security, Reliability and Trust (SnT), Automatic Control Research Group
University of Luxembourg, 29 Avenue J.F Kennedy L-1855 Luxembourg City, Luxembourg
Email: {willian.tessarolunardi;holger.voos}@uni.lu
† Institute of Mathematics and Computer Sciences (ICMC)
University of São Paulo, 400 Avenida Trabalhador São-Carlense, 13566-590 São Paulo, Brazil
Email: lhcherri@icmc.usp.br

***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. The real-world problem addressed here is an extension of flexible job shop scheduling problem and is issued from the modern printing and boarding industry. The precedence between operations of each job is given by an arbitrary directed acyclic graph rather than a linear order. In this paper, we extend the traditional FJSP solutions representation to address the parallel operations. We propose an imperialist competitive algorithm for the problem. Several instances are used for the experiments and the results show that, for the considered instances, the proposed algorithm is faster and found better or equal solutions compared to the state-of-the-art algorithms.**

I. Introduction

The real-world scheduling problem addressed in this paper is issued from the modern printing and boarding industry. The problem is an extended version of the flexible job shop scheduling problem (FJSP) where jobs are allowed to be a set of operations with an arbitrary precedence relation, thereby, including types of jobs like those in Figure 1.

The FJSP is a generalization of the job shop scheduling (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 the completion time of the last operations (makespan) [1]. The FJSP consider that there may be several machines, not necessarily identical, capable of processing an operation. Specifically, 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.

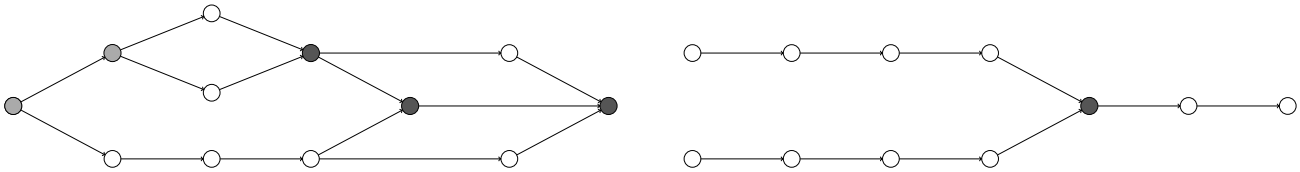
In terms of computational complexity, JSP problem is known to be one of the most difficult combinatorial optimization problems [2], and has been proven to be an NP-hard problem [3]. Since the FJSP problem is at least as difficult as the JSP, it is also NP-hard.

Many methods have been presented to solve the FJSP problem. For the FJSP, when the number of jobs is small, some exact algorithms can be employed to solve it, for example, mathematical programming [1]. But when the number of jobs rises, it is difficult to find an optimal solution in a short time. Many researchers have proposed heuristic methods to solve the FJSP, such as genetic algorithm (GA) [4], firefly algorithm [5], artificial bee colony [6], particle swarm optimization [7], tabu search [8], and memetic algorithm [9].

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*. 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.

Compared to the FJSP, the literature for the EFJSP is scarce. Vilcot and Billaut [8] considered a class of instances where each operation in a job can have more than one predecessor, but at most one direct successor. Similarly, Yu et al. [10] considered a scheduling problem with parallel operations for flight deck simulations considering G-jobs and Y-jobs. Birgin et al. [1] considered both types of jobs with parallel operations and proposed a mixed integer linear programming (MILP) model which allows the precedence between operations of a job to be given by an arbitrary directed acyclic graph (DAG) rather than a linear order.

In this paper, we extend the usual definition of the FJSP to allow the job to be a set of operations with an arbitrary precedence relation. We propose a new representation to improve the searching capabilities of the algorithm. Motivated by the fact that it is difficult to achieve an optimal solution for medium and real size problems with mathematical modeling due to its high computational complexity [10], we design an improved imperialist competitive algorithm to find



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

Fig. 1: 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.

"good enough" solutions in an acceptable time for large-sized instances. In order to compare the ICA with other methods, we implemented our proposed solution representation with a genetic algorithm (GA), and a continuous firefly algorithm (FA). The proposed ICA is experimented and compared with other methods using 60 instances from the literature.

II. 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 $\mathcal{F}(v)$ of M with each operation v. The machines in $\mathcal{F}(v)$ are the ones that can process operation v. Additionally, for each operation v and each machine k in $\mathcal{F}(v)$, 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 $f(v) \in \mathcal{F}(v)$ 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 *makespan* of a schedule s is the number $mks(s) := max_{v\ inV}(s_v + p_v^f)$. This definition does not preclude idle time in the schedule; the next one focus on non-delay schedules.

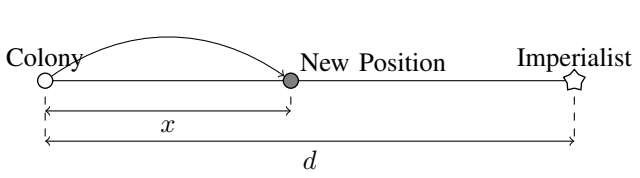
The length of a (directed) path $(v_1, v_2, ..., v_l)$ in the dag $(V, A \cup Y)$ is the number $p_{v_1}^f + p_{v_2}^f + ... + p_{v_l}^f$. For any path P in $(V, A \cup Y)$ ending at v and any schedule s, the length of P is at most s_v. For each v in V, let s_v^* be the maximum of the lengths of all paths in $(V, A \cup Y)$ ending at v. There is a simple dynamic programming algorithm that computes the tight schedule [11]. 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*).

A. MILP Model

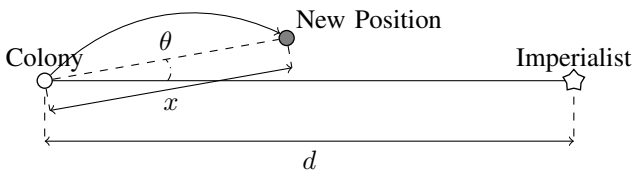
The MILP proposed by Birgin et al. [1] can be given as follows: find a rational number z, rational arrays s and p', and binary arrays x and y that

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

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.



(a) Assimilation operator.



(b) Revolution operator.

Fig. 2: Moving countries toward their imperialist.

III. Imperialist Competitive Algorithm

Encouraged by the human socio-political evolution process, a newly developed evolutionary algorithm denominated Imperialist competitive algorithm (ICA) is proposed by Atashpaz-Gargari and Lucas [12]. Similar to other evolutionary algorithms, the ICA begins with an initial population (countries). The best countries are selected to be the *imperialists*. Every imperialist has its personal set of *colonies* and the amount of colonies of each empire is equivalent to its relative power to other imperialists. The colonies of an empire evolve through operators as *assimilation* and *revolution* using the imperialist as an evolution target. Empires try to possess colonies of other empires through the *imperialistic competition* strategy. The main ICA concepts are illustrated in detail below.

1) Initial Countries: δ solutions (i.e. country) are generated randomly, where each country can be defined in form of an array

$$\zeta_i = [p_1, p_2, ..., p_\eta], \tag{8}$$

where ζ_i represents the ith country, p_i the variables, and η the total number of variables, i.e. η-dimensions of the problem to be optimized.

2) Initial Empires: α best countries are elected to be imperialist. The remaining $\delta - \alpha$ countries are used to create the colony set of each imperialist. The number of colonies that each imperialist possesses is proportional to its relative power to other imperialists defined as

$$P_k = \left| \frac{Y_k}{\sum_{i=1}^{\alpha} Y_i} \right|, \tag{9}$$

where P_k is the power of the imperialist k, $Y_i = max_k\{c_k\} - c_i$ indicates the normalized cost, and c_k is the cost of the kth imperialist. The number of initial colonies possessed by imperialist k is calculated as round$(P_k \times (\delta - \alpha))$, where round is a function that gives the nearest integer of a fractional number.

3) Assimilation and Revolution: Assimilation leads colonies to have similar features with their corresponding imperialist. During the assimilation strategy, colonies move

$$x \sim U[0, \delta d] \tag{10}$$

units towards their relevant imperialist, where x is a random value generated using uniform distribution, δ is the assimilation factor and d is the distance between the colony and the imperialist. Figure 2a shows the movement of a colony towards its imperialist.

Comparable to the mutation operator in GA, the revolution operator is added to ICA to improve the exploration property of the algorithm. In order to search nearby the imperialist, a random amount of deviation

$$\theta \sim U[-y, y] \tag{11}$$

is incorporated in the movement, where θ is a random value generated using uniform distribution, y is a parameter that adjusts the deviation from the original direction. Figure 2b shows the movement of a colony towards its imperialist in a randomly deviated direction.

4) Imperialist Exchange: If a colony is better than its imperialist due to assimilation and revolution operations, the position of the imperialist and the colony are changed, i.e., the colony becomes the imperialist and vice-versa.

5) Imperialist Competition: In this step, the weakest colony of the weakest imperialist is possessed by other stronger imperialists. This is carried out in a stochastic way. The possession probability for each imperialist is related to its total cost. The better the imperialist is, the more likely it will possess the weakest colony of the weakest empire. The total cost, which is used as a comparison criterion in this step, is defined by

$$TC_i = C_i + \xi \frac{1}{n_i^\delta} \sum_{j=1}^{n_i^\delta} C_{ij}, \tag{12}$$

where C_i and TC_i are respectively the cost and total cost values for imperialist i, n_i^δ is the number of colonies of the ith empire, C_{ij} is the cost related to jth colony of empire i, and $\xi < 1$ is the colonies consideration rate. In the imperialist competition step, if the weakest imperialist loses all of its colonies, then this imperialist is collapsed. A collapsed imperialist is possessed by other imperialists as a colony.

IV. Classic FJSP Representation

According to Gao et al. [13], a discrete-valued solution structure can be used to code the solution of FJSP. 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), respectively referred to as ϑ_1 and ϑ_2 in this paper, where ϑ_1 denotes the assigned machine for each particular operation, and ϑ_2 denotes the order in which the operations are to be processed in their

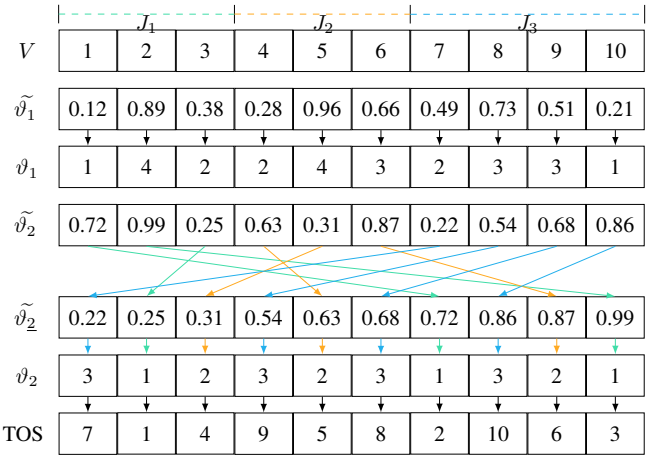


Fig. 3: Representation of a FJSP solution based on a continuous and a discrete valued structures.

assigned machines. Sections IV-A and IV-B respectively describes the discrete representation of the machine assignment and operation sequence.

Since ICA works in continuous domains, it is necessary to define a real-valued solution structure. Sections IV-D and IV-E presents the proposed continuous solution structure, also divided in two parts, where $\widetilde{\vartheta_1}$, similar ϑ_1, defines the machine operation assignment, and $\widetilde{\vartheta_2}$, similar to ϑ_2, defines the operation sequence.

The length of all strings is equal to $\mathcal{T} = \sum_{i=1}^{n} S_i$, where S_i is the number of operations of job i and n is the number of jobs, and $\mathcal{T}$ represents the total number of operations. Figure 3 shows a random example of strings based on the given set of machines and operations presented in Table I.

A. Discrete Machine Assignment String

Let $\vartheta_1 = \{\vartheta_{11}, \vartheta_{12}, ..., \vartheta_{1\mathcal{T}}\}$ represent the discrete machine assignment string where $\mathcal{T}$ is the problem dimension. The ith element in ϑ_1 represents the assigned machine for the ith operation V, and $\vartheta_{1i} \in \mathcal{F}(V_i)$. The index does not vary throughout the whole searching process.

TABLE I: An EFJSP instance with three jobs 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	$1 \rightarrow 2$	
	2	3	5	2	2	$2 \rightarrow 3$	$1, 2, 3$
	3	5	1	4	4	-	
2	4	4	3	4	5	$4 \rightarrow 6$	
	5	3	3	4	2	$5 \rightarrow 6$	$4, 5, 6$
	6	4	3	1	4	-	
3	7	3	1	3	3	$7 \rightarrow 8,\ 9$	
	8	5	3	1	3	$8 \rightarrow 10$	$7, 9, 8, 10$
	9	4	4	2	5	-	
	10	3	5	4	4	-	

$v \rightarrow u$ represents a conjunctive arc (v precedes u in the job route).

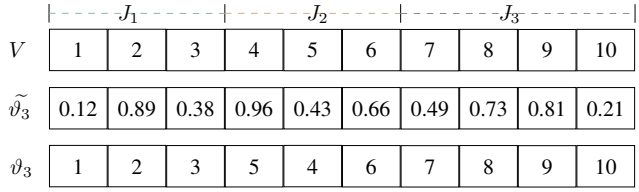


Fig. 4: Representation of the topological order of the jobs of an EFJSP instance defined based on the MBFS algorithm. Based on ϑ_3 the topology of job $1 = \{1, 2, 3\}$, job $2 = \{5, 4, 6\}$, and job $3 = \{7, 8, 9, 10\}$.

Algorithm 1 Modified Traversing Graph MBFS

```
for each job i do
    DAG ← J_i                                  ▷ J_i digraph of job i
    while ∃v ∈ DAG : v ∉ ϑ_3 do
        V ← ∅                                  ▷ V vertices to visit
        for each v ∈ DAG do
            if P_v ⊆ ϑ_3 ∨ P_v = ∅ then        ▷ P_v predec. of v
                Add v to V end if
        end for
        Sort V based on cost                   ▷ Ascending order
        for each v ∈ V do
            Add v to ϑ_3
        end for
    end while
end for
```

B. Discrete Operation Sequence String

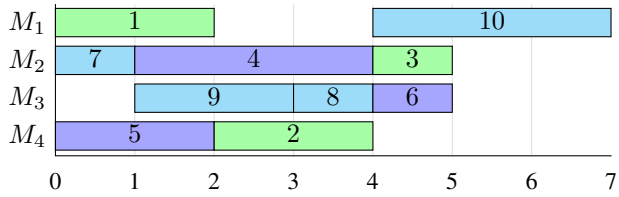
Let $\vartheta_2 = \{\vartheta_{21}, \vartheta_{22}, ..., \vartheta_{2\mathcal{T}}\}$ represent the operation sequence string where $\mathcal{T}$ is the problem dimension. The string ϑ_2 represents the order in which the operations will be processed in their designated machines. To avoid repair mechanisms, this representation uses an unpartitioned permutation with S_i repetitions of the job numbers, i.e., every job i appear S_i times in ϑ_2.

C. Translating the Discrete OS String

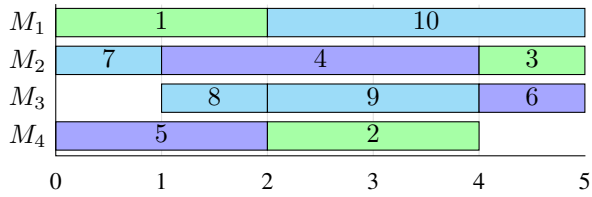
By scanning the ϑ_2 from left to right, the f_ith appearance of a job i refers to the f_ith operation in topological order of the operations of job i, where f_i starts with 0 and counts the appearances of job in the string. For example, scanning ϑ_2 from left to right, for every appearance of job i, 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 leads to a feasible solution. For the instance shown in Table I, one possible OS string, and TOS string is presented in Figure 3, where the procedure of translating the OS string is represented by the arrows between nodes of ϑ_2 and TOS.

D. Continuous MS string

Let $\widetilde{\vartheta_1} = \{\widetilde{\vartheta_{11}}, \widetilde{\vartheta_{12}}, ..., \widetilde{\vartheta_{1\mathcal{T}}}\}$, similar to ϑ_1 in discrete structure, represents the continuous machine assignment string where $\widetilde{\vartheta_{1i}} \in [0, 1)$, and $\mathcal{T}$ is the problem dimension. The



(a) Gantt Chart of the decoded solution based on the strings given in Figure 3, and topological order given in Table I.



(b) Gantt Chart of the optimal solution of the problem presented in Table I.

Fig. 5: Translating the OS string with different topological order.

ith element in $\widetilde{\vartheta_1}$ represents the assigned machine for the ith operation in V, defined as

$$\vartheta_{1i} = \left\lfloor |\mathcal{F}(V_i)|\ \widetilde{\vartheta_{1i}} + 1 \right\rfloor, \quad i = 1, ..., \mathcal{T}, \tag{13}$$

where $\lfloor x \rfloor$ indicates the greatest integer number smaller than x. Equation (13) can be seen represented by the arrows between every node of $\widetilde{\vartheta_1}$ and ϑ_1 on Figure 3.

E. Continuous OS string

Let $\widetilde{\vartheta_2} = \{\widetilde{\vartheta_{21}}, \widetilde{\vartheta_{22}}, ..., \widetilde{\vartheta_{2\mathcal{T}}}\}$, similar to ϑ_2 in discrete structure, represents the continuous operation sequence string, where $\widetilde{\vartheta_{2i}} \in [0, 1]$, and $\mathcal{T}$ is the problem dimension.

In order to define a sequence of jobs as ϑ_2, the elements of $\widetilde{\vartheta_2}$ are ordered (ascending or descending) into $\widetilde{\vartheta_{\underline{2}}}$. In this way, the operations sequence is defined as

$$\vartheta_{2i} = h(\widetilde{\vartheta_{\underline{2}i}}), \quad i = 1, ..., \mathcal{T}, \tag{14}$$

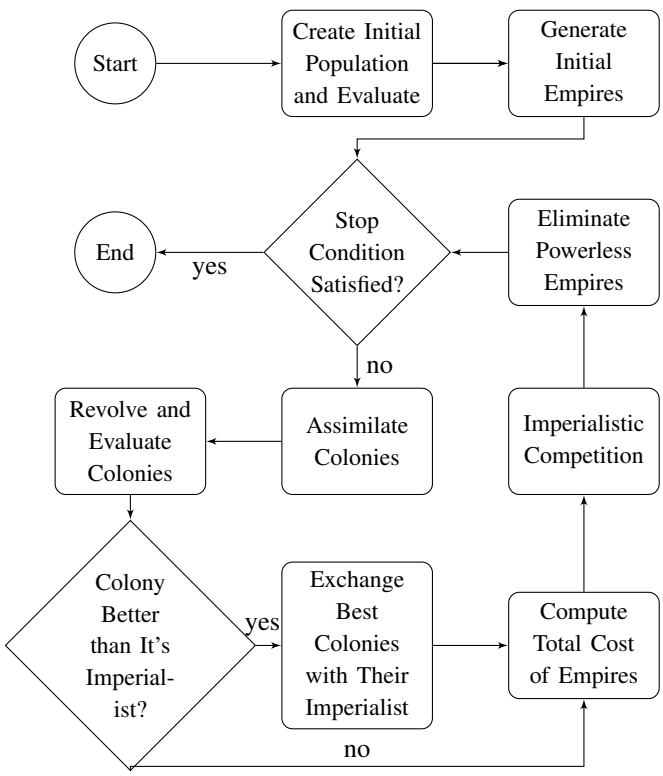


Fig. 6: Flow diagram of the proposed algorithm.

where h is a function that gives the job id of the operation represented by the position (index) of first appearance the value $\widetilde{\vartheta_{\underline{2}i}}$ in the string $\widetilde{\vartheta_2}$. Equation (14) can be seen represented by the colored arrows between nodes of $\widetilde{\vartheta_2}$, $\widetilde{\vartheta_{\underline{2}}}$ and ϑ_2 in Figure 3.

V. Representation for the EFJSP

Due to the arrangement of operations in Y-jobs and G-jobs, a topological sorting algorithm can be used to define the topological order of the operations. The topological order is an essential feature for the translation of the OS string into a feasible sequence that respects the precedence relationship of all operations of a job. There exist well-known linear time algorithms for determining the topological order of a directed graph (digraph), e.g., Cormen [11] applies a depth-first search (DFS) algorithm. Nevertheless, with this strategy, the order in which parallel operations appear in the sequence is strictly related to the strategy used to define it.

Theorem 1. *Due to the parallel operations present in the topology of EFJSP jobs, determining the operation scheduling sequence of jobs based on fixed topological sequences does not empower the searching method to find the optimal solutions in some cases.*

Proof. Consider ϑ_1 and ϑ_2 shown in Figure 3, and the job 3 presented in Table I. We denote V_i as the i operation in V. Since in the route of the job 3 operations V_8 and V_9 are parallel and independent from each other, and V_8 is the only predecessor of operation V_{10}, the minimum starting time of V_{10} is the completion time of operation V_8. Since DFS gives the topological order $\{7, 9, 8, 10\}$, throughout the translation of the OS string, operation V_9 will come before V_8 in the operations sequence. Consequently, in a situation where V_8 and V_9 are assigned to the same machine, V_9 will delay the starting time of V_8, and as consequence, V_{10} will further be delayed. Based on the strings and instance given earlier, the makespan obtained with the topological order settled by DFS is 7. The Gantt Chart for this solution is shown in Figure 5a. Nonetheless, considering a distinct topological order for job 3 as $\{7, 8, 10, 9\}$ or $\{7, 8, 9, 10\}$, the makespan can be minimized to 5. The Gantt Chart for this solution is shown in Figure 5b. □

Therefore, we extend the traditional FJSP representation adding the topological order (TP) string. The TP string $\widetilde{\vartheta_3} =$

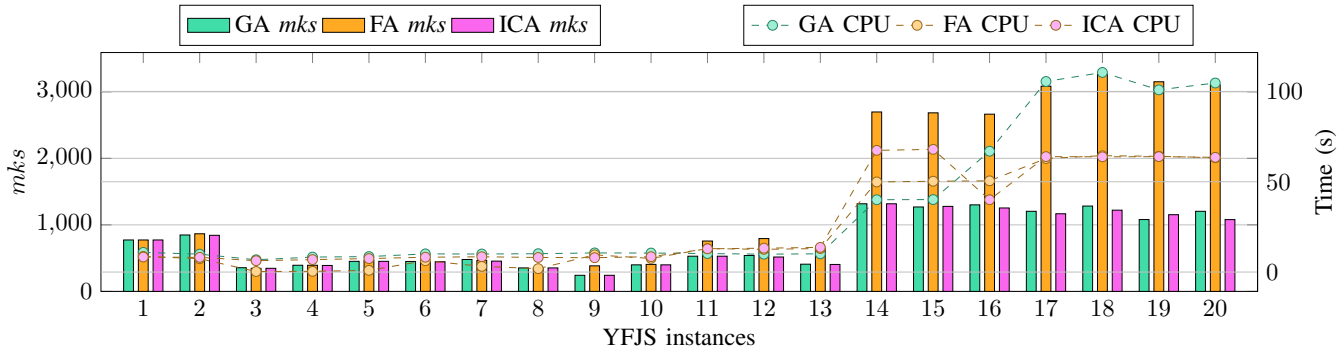


Fig. 7: Best makespan and mean CPU time for the experiment with the YFJS instances.

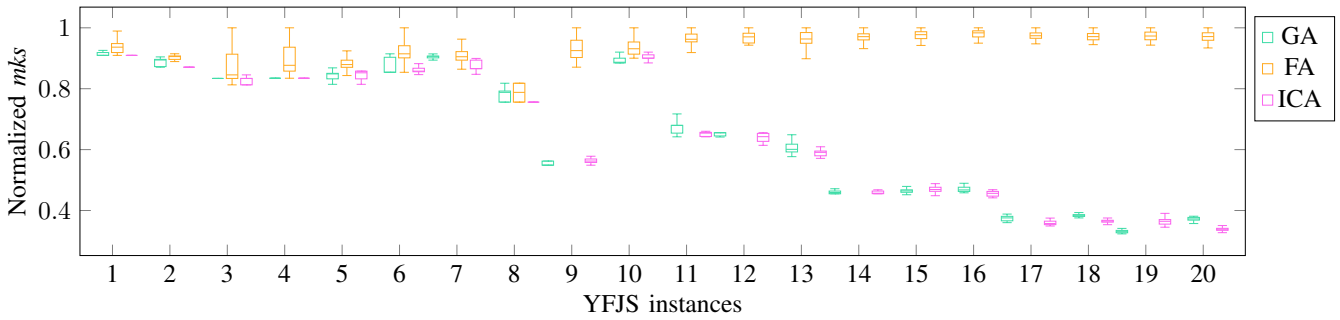


Fig. 8: Box plot of the makespan obtained with the experiments involving the YFJS instances.

$\{\vartheta_{31}, \vartheta_{32}, ..., \vartheta_{3\mathcal{T}}\}$, where $\vartheta_{3i} \in [0, 1]$, denotes a cost c_i for the ith operation in the V_i. In this way, for any two parallel operations, the one with less cost comes before in the topological order.

A modified cost-based version of the Breadth-First Search (BFS) traversing graph algorithm is used to define the topological order of each job. In our modified traversing graph algorithm (MBFS), a set $\mathcal{V}$ of vertices to be visited are defined. The vertices included in $\mathcal{V}$ are those where all its predecessors were already visited. Every vertex in $\mathcal{V}$ is visited in an ascending order based on the cost defined by $\widetilde{\vartheta_3}$. The pseudocode presented in Algorithm 1 illustrates the basic steps of the MBFS. Figure 4 shows a random TP string $\widetilde{\vartheta_3}$ for the instance presented in Table I, where ϑ_3 represents the topological order of each job given by the MBFS algorithm.

VI. The Proposed ICA Algorithm

Difficult problems such as FJSP demand sophisticated techniques to produce reasonable results in acceptable time. It is customarily preferable to get good sub-optimal results quickly than to wait for days to get an optimal solution. With this motivation, we propose an improved imperialist algorithm for the EFJSP.

In our algorithm, each country represents an EFJSP solution. Following the random creation of the initial population, the best countries become imperialists and empires are created. The population evolves with each country moving toward its respective imperialist. Each country has its strings $\widetilde{\vartheta_1}$ and $\widetilde{\vartheta_2}$ assimilated by the imperialist. In our implementation, revolution procedure is performed after every assimilation procedure and it is composed of three steps: (1) pick a random element of $\widetilde{\vartheta_1}$ and assign a random value in $[0, 1)$; (2) pick a random element of $\widetilde{\vartheta_2}$ and assign a random value in $[0, 1]$; (3) pick a random element of $\widetilde{\vartheta_3}$ and assign a random value in $[0, 1]$. Then, every country is evaluated and imperialist exchange is conducted if necessary. Finally, at the end of each iteration, the imperialist competition is performed. Two stop conditions are used: (1) the maximum number of iterations $\mathcal{G}$; (2) the maximum number of iterations without improvement $\mathcal{I}$.

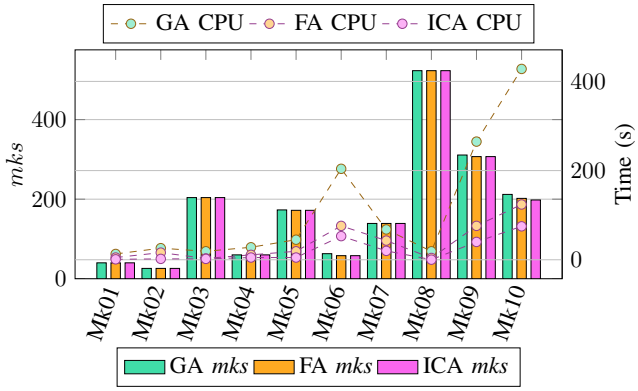


Fig. 9: Best makespan and mean CPU time for the experiments with the Brandimarte path-job instances.

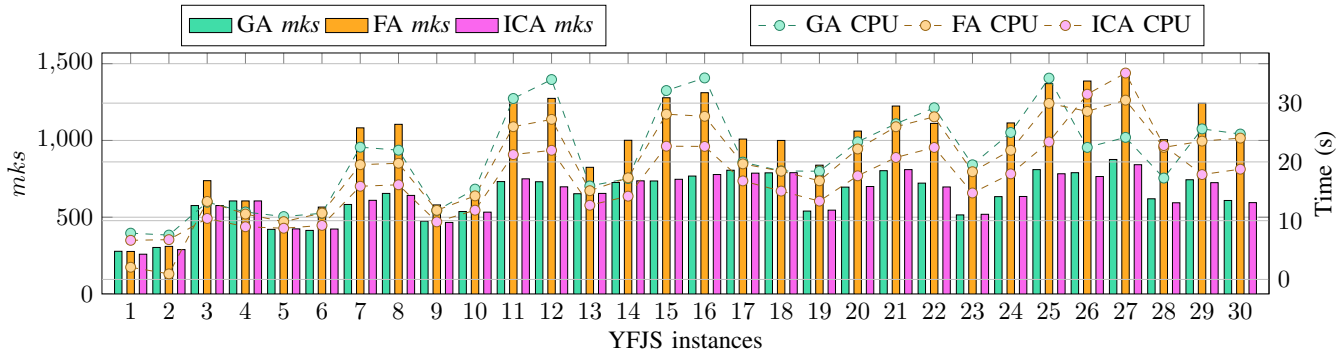


Fig. 10: Best makespan and mean CPU time for the experiment with the DAFJS instances.

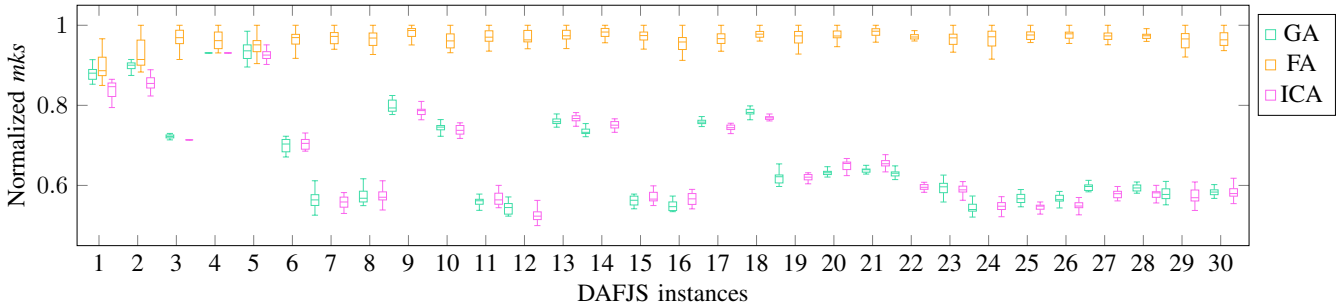


Fig. 11: Box plot of the makespan obtained with the experiments involving the DAFJS instances.

VII. Numerical Results

This section presents the results of computational experiments involving the algorithm. We used as benchmarks 10 FJSP instances (named Mk) introduced by Brandimarte [14], 20 EFJSP instances composed of Y-jobs (named YFJS) proposed in [1], and 30 EFJSP instances composed of G-jobs (named DAFJS) further propose in [1].

In order to compare the proposed ICA, implemented in (C++), we implemented (C++) the most applied method for the FJSP, a Genetic Algorithm (GA) (see Chaudhry and Khan [15], Amjad et al. [16]). Moreover, a discrete Firefly Algorithm (DFA) showed good results for the FJSP (see Lunardi et al. [5], Lunardi and Voos [4], Yang [17]). In this way, in order to check the performance of the continuous version, we implemented (C++) a continuous Firefly Algorithm (FA).

The GA implemented is based on the classic GA's structure and applies a random crossover and a random mutation to the continuous strings. Based on experiments, tournament selection performed better than fitness proportionate selection, and it maintains the diversity of the population. The FA that we implemented is based on the algorithm proposed in [17]. We performed experiments for each algorithm to define the parameters, where the GA parameters are 500 individuals, 1000 generations, 0.9 crossover rate, and 0.3 mutation rate. The FA parameters is 75 fireflies, 500 generations, 1.0 attractiveness, and 0.35 randomization rate. For more details about of the GA and FA check [18].

The algorithms were run by a 2*Intel Xeon E5-2680 v3 @ 2.5 GHz with 24 CPUs on a Linux HPC cluster at University of Luxembourg. We performed each experiment involving each instance 25 times for each algorithm. Based on experiments we find out that the best parameters for the ICA applied for the EFJSP are $\mathcal{G} = 1000, \mathcal{I} = 100, \delta = [200, 500]$, and $\alpha = [5, 15]$. For the experiments involving the instances Mk01,...Mk05, Mk07,...,Mk09, YFJSP ≤ 10, and DAFJS ≤ 10 we used $\delta = [250], \alpha = [5]$; for the instances Mk06, Mk10, YFJS > 10, and DAFJSP > 10 we used $\delta = [400], \alpha = [10]$. Supplemental numerical results, charts, and the instances are provided at https://willtl.github.io/etfa2018/.

Figure 9 presents the results for the instances of Brandimarte [14]. The bars denote the mks and the lines represent the CPU time, in seconds. This set of instances include only path-jobs. The proposed ICA is more effective for the instances Mk06 and Mk10, and more efficient than the GA and FA for all instances.

Figures 7, 8, 10, and 11 present the results for the two sets of instances of Birgin et al. [1]. In the bar charts is presented the mean CPU time (s) and the best obtained mks, and in the box plot is showed distribution of the data of the 25 runs.

On the experiment among the Brandimarte [14] instances, we can perceive that the ICA was able to find lower mks values for the two hardest instances (i.e. Mk06 and Mk10), being more efficient related to the compared algorithms. Based on the investigations with Y-job and G-job instances, both the GA and FA loses efficiency for larger instances. The FA's 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 fireflies,

randomization, and iterations), it was not able to obtain similar solutions to those that ICA and GA found. Based on the numerical results, showed in [1], we can see that the MILP model is, for some instances, more efficient for smaller instances, and as expected, it loses efficiency for larger instances.

Here is a summary of our results. The GA is more effective and efficient than the continuous FA for the EFJSP. The proposed continuous ICA is a consistent and steady algorithm for the EFJSP. The ICA obtained better values and lower variance for most of the experiments. Based on the bounds given by the MILP model, we can see that the GA and the ICA decreased the gap (i.e. the distance between lower and upper bounds) for several instances.

VIII. Conclusion

In the present paper, we extended the definition of the FJSP taking into account parallel operations in the route of the jobs. The usual representation of the FJSP was modified to improve the searching capabilities of the algorithms. We put forward an imperialist competitive algorithm to solve the EFJSP. In order to evaluate the performance of the solution methods, 50 EFJSP instances were used. Computational experiments on 10 famous FJSP instances was performed in order to provide comparisons with other state-of-the-art algorithms. The experiments among the ICA and others recently published algorithms shows that it is a feasible approach for the considered problem.

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*, vol. 8, no. 4, pp. 1417–1431, 2014.

[2] 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*, vol. 4, pp. 445–522, 1993.

[3] M. R. Garey, D. S. Johnson, and R. Sethi, "The complexity of flowshop and jobshop scheduling," *Mathematics of operations research*, vol. 1, no. 2, pp. 117–129, 1976.

[4] 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*. ACM/SIGAPP, 2018, pp. 1–8.

[5] 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 *17th International Conference on Artificial Intelligence and Soft Computing (ICAISC), 2018*. Springer, 2018, pp. 1–8.

[6] 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*, vol. 42, no. 21, pp. 7652–7663, 2015.

[7] G. Moslehi and M. Mahnam, "A pareto approach to multi-objective flexible job-shop scheduling problem using particle swarm optimization and local search," *International Journal of Production Economics*, vol. 129, no. 1, pp. 14–22, 2011.

[8] 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*, vol. 190, no. 2, pp. 398–411, 2008.

[9] Y. Yuan and H. Xu, "Multiobjective flexible job shop scheduling using memetic algorithms," *IEEE Transactions on Automation Science and Engineering*, vol. 12, no. 1, pp. 336–353, 2015.

[10] L. Yu, C. Zhu, J. Shi, and W. Zhang, "An extended flexible job shop scheduling model for flight deck scheduling with priority, parallel operations, and sequence flexibility," *Scientific Programming*, vol. 2017, 2017.

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

[12] E. Atashpaz-Gargari and C. Lucas, "Imperialist competitive algorithm: an algorithm for optimization inspired by imperialistic competition," in *IEEE Congress on Evolutionary computation, 2007.*, 2007, pp. 4661–4667.

[13] J. Gao, M. Gen, and L. Sun, "Scheduling jobs and maintenances in flexible job shop with a hybrid genetic algorithm," *Journal of Intelligent Manufacturing*, vol. 17, no. 4, pp. 493–507, 2006.

[14] P. Brandimarte, "Routing and scheduling in a flexible job shop by tabu search," *Annals of Operations research*, vol. 41, no. 3, pp. 157–183, 1993.

[15] I. A. Chaudhry and A. A. Khan, "A research survey: review of flexible job shop scheduling techniques," *International Transactions in Operational Research*, vol. 23, no. 3, pp. 551–591, 2016.

[16] M. K. Amjad, S. I. Butt, R. Kousar, R. Ahmad, M. H. Agha, Z. Faping, N. Anjum, and U. Asgher, "Recent research trends in genetic algorithm based flexible job shop scheduling problems," *Mathematical Problems in Engineering*, vol. 2018, 2018.

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

[18] W. T. Lunardi and H. Voos, "An extended flexible job shop scheduling problem with parallel operations," *ACM SIGAPP Applied Computing Review*, vol. 18, no. 2, pp. 46–56, 2018.