Intelligent Automation & Soft Computing
DOI: 10.32604/iasc.2023.028604
Article

Tech Science Press

Minimizing Total Tardiness in a Two-Machine Flowshop Scheduling Problem with Availability Constraints

Mohamed Ali Rakrouki[1,2,*], Abeer Aljohani[1], Nawaf Alharbe[1], Abdelaziz Berrais[2] and Talel Ladhari[2]

[1]Applied College, Taibah University, Saudi Arabia
[2]University of Tunis, Tunis, Tunisia
*Corresponding Author: Mohamed Ali Rakrouki. Email: mrakrouki@taibahu.edu.sa
Received: 13 February 2022; Accepted: 24 March 2022

Abstract: In this paper, we consider the problem of minimizing the total tardiness in a deterministic two-machine permutation flowshop scheduling problem subject to release dates of jobs and known unavailability periods of machines. The theoretical and practical importance of minimizing tardiness in flowshop scheduling environment has motivated us to investigate and solve this interested two-machine scheduling problem. Methods that solve this important optimality criterion in flowshop environment are mainly heuristics. In fact, despite the $\mathcal{NP}$-hardness in the strong sense of the studied problem, to the best of our knowledge there are no approximate algorithms (constructive heuristics or metaheuristics) or an algorithm with worst case behavior bounds proposed to solve this problem. Thus, the design of new promising algorithms is desirable. We develop five metaheuristics for the problem under consideration. These metaheuristics are: the Particle Swarm Optimization (PSO), the Differential Evolution (DE), the Genetic Algorithm (GA), the Ant Colony Optimization (ACO) and the Imperialist Competitive Algorithm (ICA). All the proposed metaheuristics are population-based approaches. These metaheuristics have been improved by integrating different local search procedures in order to provide more satisfactory, especially in term of quality solutions. Computational experiments carried out on a large set of randomly generated instances provide evidence that the Imperialist Competitive Algorithm (ICA) records the best performances.

Keywords: Optimization; machine scheduling; flowshop; evolutionary algorithms

Nomenclature

M_i	machine
p_{ij}	processing time
r_j	release date
d_j	due date
h_i	non-availability periods
C_j	completion time
T_j	tardiness
N	population size (number of solutions)

This work is licensed under a Creative Commons Attribution 4.0 International License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

X_i^t	population member
V_i^t	velocity of particle
w^t	inertia weight
σ	partial sequence of scheduled jobs
J	job set
π_k	position of job
F	mutant factor
β	decrement factor
U^t	trial population
τ	matrix of the pheromone density
δ	matrix of the distance
κ	pheromone decay coefficient
N_{imp}	number of imperialists
N_{col}	number of colonies Ψ normalized power
ϕ	cost of the imperialist
TC_{imp}	total cost of the empire
f	due date slack factor
T	tardiness factor
R	dispersion factor

1 Introduction

Scheduling has been often considered in the literature and has many practical issues in domains like manufacturing, computer processing, transportation, production planning, etc., In these domains, scheduling involves a decision-making process. It is concerned with allocating resources (machines) to tasks (jobs) throughout certain time periods with the goal of optimizing one or more objectives [1]. The resources can be machinery resources, human resources, CPU, Web server, etc. and referred to as "machines". The tasks to be scheduled can be production operations, CPU tasks, timetabling activities, etc., and referred to as "jobs". Basic scheduling problems consider that machines are available during the scheduling, or in practice all machines may be unavailable during several periods of time due to machine breakdown (known as stochastic) or a preventive maintenance (known as deterministic). In real-world scheduling problems, the total tardiness criterion is highly crucial, especially in industry, since failure to meet deadlines can harm company's reputation and lead to a loss of confidence, increased costs, and customer loss.

This paper deals with the two-machine flowshop scheduling problem where the machines are subject to non-availability constraints and the jobs are subject to release dates. More precisely, we are given n jobs $(j = 1, \ldots, n)$ to be scheduled in the same processing sequence on two machines M_1 and M_2. Each job must be processed on machine M_1 and on machine M_2 during p_{1j} and p_{2j} time units, respectively. There exists a due date d_j associated with the completion of each job and each job can not start before a release date r_j. The aim is minimizing the total tardiness of jobs.

Moreover, the following assumptions are considered. Each machine $M_i(i = 1, 2)$ is unavailable during h_i periods of time (hole). The number of holes, their starting times and their finish times are known in advance. No two holes overlap on the same machine. The two machines can have the same holes. Zero, one or several holes can occur during the processing of one job. Job processing can be interrupted by a machine unavailability and resumed after.

Let C_j be the completion time of job j on machine M_2, the objective is to find a schedule which minimizes the total tardiness of the jobs $\sum_{j=1}^{n} T_j$ where $T_j = \max\left(0, C_j - d_j\right)$. According to the notation

specified by Pinedo 2012 [1] and Lee [2], this problem is denoted $F2, h_{lo}|r-a, r_j|\sum T_j$. It's an extension of the $F2||\sum T_j$ problem and known to be $\mathcal{NP}$ -hard in the strong sense [3].

The rest of this paper is organized as follows: A literature review is provided in Section 2. Section 3 discusses the five proposed metaheuristics. In Section 4, the experimental findings and the comparative analysis are discussed. Finally, conclusions are set out in Section 5.

2 Literature Review

This problem has been studied only once before, despite its theoretical and practical importance. Indeed, Berrais et al. [4] proposed a mixed-integer formulation as well as some constructive heuristics for the problem under consideration.

The m-machine flowshop ($F||\sum T_j$) has been widely studied in the literature. Kim [5] developed a branch-and-bound algorithm using some proposed lower bounds and a dominance rule. A tabu search algorithm with diversification, intensification, and neighborhood restriction were proposed by Armentano et al. [6] to solve the same problem. Hasija et al. [7] proposed a simulated annealing (SA) algorithm. Chung et al. [8] proposed an exact method based on a branch-and-bound algorithm. Ghassemi Tari et al. [9] proposed four heuristics based on cost over time and more after Ghassemi Tari et al. [10] proposed seven heuristic algorithms, based on shortest processing time (SPT) and earliest due date (EDD) rules and then modified and combined to develop others algorithms. Chung et al. [11] developed a genetic algorithm. Liu et al. [12] proposed five dispatching rules, and a constructive heuristic for the m-machine no-wait flowshop with total tardiness criterion ($Fm|no-wait|\sum T_j$). Ghassemi Tari et al. [13] proposed some heuristic procedures for tardiness problem with intermediate due dates. Fernandez-Viagas et al. [14] as well as Karabulut [15] proposed some iterated-greedy-based algorithms for the $F||\sum T_j$.

Sen et al. [16] considered the two-machine problem ($F2||\sum T_j$) and developed a branch-and-bound procedure in many presented cases. Koulamas [17] developed a heuristic denoted by guaranteed accuracy shifting bottleneck algorithm. Schaller [18] treated the same problem and proposed three dominance conditions to improve some previously proposed ones, and then proposed a new dominance rule as well as a branch-and-bound algorithm. Kharbeche et al. [19] proposed an exact method based on mixed-integer programming models.

Schmidt [20] and Ma et al. [21] proposed good descriptions and details about availability constraints. Blazewicz et al. [22] proposed three heuristic methods as well as a simulated annealing algorithm for minimizing the makespan in the resumable case ($F2, h_{ij}|r-a|C_{\max}$). Ben Chihaoui et al. [23] proposed several lower and upper bounds and used them in a branch-and-bound algorithm to solve the two-machine no-wait problem subject to release dates under the non-resumable case ($F2, h_{1j}|nr-a, r_j|C_{\max}$) with only 1 hole considered on each machine. Under the assumption of the non-availability on the first machine, Lee et al. [24] proposed a branch-and-bound algorithm with some dominance properties and lower bounds as well as heuristic algorithm. For the parallel machine problem, Lee et al. [25] proposed some dominance properties and lower bounds as well as a branch-and-bound algorithm for minimizing the total tardiness where the machines need preventive maintenance.

3 Evolutionary Algorithms

In this section, five evolutionary algorithms are described. These algorithms are combined with several local search procedures to take the advantages of rapid exploitation and global optimization.

3.1 Particle Swarm Optimization Algorithm

Particle Swarm Optimization (PSO) algorithm is an evolutionary meta-heuristic proposed by Kennedy et al. [26]. PSO is similar to genetic algorithms but it hasn't neither crossover nor mutation operators. Instead, it uses randomness of real numbers and the global communication among the swarm particles. Since its introduction, PSO algorithm has been improved with different techniques and proposed for various problems [27–30].

A PSO algorithm consists of a population $\left[X_1^t, X_2^t, \ldots, X_N^t\right]$ of N particles. Each particle $X_i^t (i = 1, \ldots, N)$ at iteration t of the algorithm is denoted $X_i^t = \left[x_{i1}^t, x_{i2}^t, \ldots., x_{in}^t\right]$ corresponding to its position in the swarm. The quality of position is represented by a fitness (objective function value). At each iteration t, the velocity $V_i^t = \left[v_{i1}^t, v_{i2}^t, \ldots, v_{in}^t\right]$ and the position of each particle $X_i^t = \left[x_{i1}^t, x_{i2}^t, \ldots., x_{in}^t\right]$ are updated toward its current best position $\left(X_i^* = \left[x_{i1}^*, x_{i2}^*, \ldots., x_{in}^*\right]\right)$ and the global best position $\left(G^* = \left[g_1^*, g_2^*, \ldots., g_n^*\right]\right)$ in the swarm. So, at each step and for each particle, a new velocity value is calculated based on its current one. This new value is used to compute the next position of the particle in the swarm. This process is repeated until a termination condition is reached.

The inertia weight w^t determines the impact of a particle's previous velocity to its current one. A large weight directs the algorithm to a global search while a small weight implies a local search. According to Bansal et al. [31], good results can be found when inertia weight is from 0.9 down to 0.4. This latter is updated at each iteration as $w^t = w^{t-1}\beta$, where β is a decrement factor ($\beta = 0.975$).

The velocity $V_i^t = \left[v_{i1}^t, v_{i2}^t, \ldots, v_{in}^t\right]$ of each particle is updated according to:

$$v_{ij}^t = w^{t-1}v_{ij}^{t-1} + c_1 r_1\left(x_{ij}^* - x_{ij}^{t-1}\right) + c_2 r_2\left(g_j^* - x_{ij}^{t-1}\right); \forall j = 1, \ldots, n \tag{1}$$

where c_1, c_2 are the acceleration coefficients and r_1, r_2 two randomly generated constant drawn on $[0..1]$. Usually $c_1 + c_2 \leq 4$. In this algorithm, $c_1 + c_2 = 4$ (empirically chosen values).

The position of each particle $X_i^t = \left[x_{i1}^t, x_{i2}^t, \ldots., x_{in}^t\right]$ is updated as follows:

$$x_{ij}^t = x_{ij}^{t-1} + v_{ij}^t; \forall j = 1, \ldots, n \tag{2}$$

Since real numbers are used in the particle representation, real numbers are transformed to a feasible solution (permutation). Two approaches are used in our algorithm:

The Smallest Position Value (SPV): It consists of transforming the real values $X_i^t = \left[x_{i1}^t, x_{i2}^t, \ldots., x_{in}^t\right]$ to a permutation by sorting these values in ascending order. Example: $X = [x_1 = 0.24, x_2 = -0.12, x_3 = 1.21, x_4 = -1.25]$ with $\pi = (1, 2, 3, 4)$. After sorting, $X = [x_4 = -1.25, x_2 = -0.12, x_1 = 0.24, x_3 = 1.21]$ so $\pi = (4, 2, 1, 3)$.

The Biggest Position Value (BPV) have the same principle as SPV but the positions are sorted in descending order.

3.1.1 Particles Initialization

The population is initialized by generating randomly the initial position and velocity vectors for each particle. The initial position values $X_i^0 = \left[x_{i1}^0, x_{i2}^0, \ldots, x_{in}^0\right]$ are drawn uniformally on $[x_{\min}, x_{\max}] = [-4.0, 4.0]$. The velocity $V_i^0 = \left[v_{i1}^0, v_{i2}^0, \ldots, v_{in}^0\right]$ of each particle X_i^0 is generated randomly, namely $v_{ij}^t = [v_{\min}, v_{\max}] = [-4.0, 4.0]$.

In order to obtain a heterogeneous population, most of the particles are randomly generated while some particles are generated using five constructive heuristics $H_i (i = 1, 2, 3, 4, 5)$ developed by Berrais et al. [4].

3.1.2 Local Search Hybridization

In order to produce high-quality solutions, a local search (LS) procedure is integrated to the PSO algorithm. The proposed local search is based on the Nawaz et al. (NEH) algorithm [32].

Let σ be a partial sequence of scheduled jobs and $\bar{J}$ be the set of unscheduled jobs, this procedure can be described as follows:

Algorithm 1. NEH_Based_LS

Step 1: Let $\Pi = (\pi(1), \pi(2), \ldots, \pi(n))$ be a sequence. Select among the partial sequences $\sigma = \pi(1), \pi(2)$ and $\sigma = \pi(2), \pi(1)$ the one having the minimum partial total tardiness. Set $\bar{J} = \bar{J} \backslash \{\pi(1), \pi(2)\}$ and $k = 2$

Step 2: Insert the job $\pi(k)$ to the $k+1$ possible position of σ. Select the sequence σ with the minimum partial total tardiness among $k+1$ sequences. Set $\bar{J} = \bar{J} \backslash \{\pi(k)\}$

Step 3: Repeat **Step 2** until $\bar{J} = \varnothing$

3.1.3 Pseudo-code of the proposed PSO algorithm

An outline of the proposed PSO is the following:

Algorithm 2. PSO

Step 1: Initialize the population by generating randomly $N(N = 20)$ particles. Set $t = 0$.

For each $X_i^0 (i = 1, \ldots, N)$ **Do**

. Set the initial position vector $X_i^0 = \left[x_{i1}^0, \ldots, x_{in}^0\right]$

. Set the initial velocity vector $V_i^0 = \left[v_{i1}^0, \ldots, v_{in}^0\right]$

. Set the current best X_0^* and the global best G^*

End For

. Apply **NEH_Based_LS** to G^*

. $t = t + 1$

Step 2:

Repeat

For each $\ldots (i = 1, \ldots, N)$ **Do**

. Update inertia weight: $w^t = w^{t-1}\beta$

. Update velocity $v_{ij}^t = w^{t-1}v_{ij}^{t-1} + c_1 r_1 \left(x_{ij}^* - x_{ij}^{t-1}\right) + c_2 r_2 \left(g_j^* - x_{ij}^{t-1}\right)$

. Update position $x_{ij}^t = x_{ij}^{t-1} + v_{ij}^t$

. Apply SPV rule to generate the permutation π_i^t corresponding to X_i^t

. Update the current best X_i^* if $\left(TT\left(\pi_i^t\right) < TT\left(\pi_i^{t-1}\right)\right)$

. Apply **NEH_Based_LS** to π_i^* associated to the current best X_i^*

End For

. Update the global best G^*

. Apply **NEH_Based_LS** to π^* associated to the global best G^*

. Update the iteration counter $t = t + 1$

Until (the maximum number of iterations is reached)

3.2 Differential Evolution Algorithm

Differential Evolution (DE) is an evolutionary algorithm firstly proposed by Storn et al. [33] for minimizing non differentiable continuous space functions. Like PSO, a DE algorithm is a population-based search meta-heuristic. It is one of the most powerful stochastic algorithms and it has been widely and successfully applied in many areas. DE algorithms were firstly designed to work with continuous variables. Then different strategies have been proposed to adapt the DE algorithm to optimize integer variables. Lampinen et al. [34] used a simple function to convert real numbers to integers. Onwubolu et al. [35] developed two strategies known as Forward Transformation (FT) and Backward Transformation (BT). Tasgetiren et al. [36] used the Smallest Position Value (SPV) rule inspired from random key representation encoding scheme of Beans [37].

In DE algorithm the first population is called target population. It consists of N randomly chosen chromosomes (solutions). Each solution $X_i = [x_{i1}, x_{i2}, \ldots, x_{in}]$ is represented by a random floating-point numbers vector. As a mutation operator, two solutions are randomly chosen and the weighted difference between them is added to a third solution to generate a new population (called mutant population). After that, the crossover operator was applied. In this step, the mutated solutions obtained in the previous process are combined with the target population to generate a new one (known as trial population). Finally, the fitness value of the target and trial populations were compared using a selection operator to determine who can survive for the next generation. An outline of our proposed DE algorithm is the following:

Algorithm 3. DE

Step 0:

. Initialize target population

. Evaluate target population

. Apply **NEH_Based_LS** to the best solution

Step 1:

Repeat

. Generate mutant population

. Generate trial population

. Selection

. Apply local search for each selected member of the trial population

. $t = t + 1$

Until (Termination condition)

3.2.1 Target Population

Our DE starts with the initialization of the target population $P^t = \left[X_1^t, X_2^t, .., X_N^t\right]$ of N members. Each member $X_i^t = \left[x_{i1}^t, x_{i2}^t, \ldots, x_{in}^t\right]$ at the iteration t is a vector of continuous random number x_{ij}^t, where n is the number of jobs and $x_{ij}^0 = \dfrac{x_{\min} + (x_{\max} - x_{\min}) \cdot r}{d_j}$ with $r \in [0, 1], x_{\min} = -1$ and $x_{\max} = 1$.

To find the corresponding permutation, H_1, H_2, H_3, H_4 and H_5 are applied in order to generate the five first solutions and the best solution between the Smallest Position Value (SPV) and Biggest Position Value (BPV) for the remaining solutions. Then each member X_i^t in the target population is evaluated by computing the total tardiness.

3.2.2 Mutant Population Generation

A weighted difference between two randomly chosen members from the target population X_a^t and X_b^t is added to a third one X_c^t $(a \neq b \neq c \in [1, N])$ to obtain the mutant population $V^t = \left[V_1^t, V_2^t, \ldots, V_N^t\right]$ where $V_i^t = \left[v_{i1}^t, v_{i2}^t, \ldots, v_{in}^t\right]$ denotes a mutant individual. v_{ij}^t is computed as in Onwubolu et al. [35]. Indeed, they have proved that is the best strategy for the total tardiness criterion. So, in our DE procedure $V_i^t = X_a^t + F \cdot \left(X_b^t - X_c^t\right)$ as a mutant factor that controls the differential variation amplification $\left(X_b^t - X_c^t\right)$. In our implementation $F = 0.5$.

3.2.3 Trial Population Generation

Let U^t denotes the trial population with $U^t = \left[U_1^t, U_2^t, \ldots, U_N^t\right]$. The U_i^t is the member of the trial population with $U_i^t = \left[u_{i1}^t, u_{i1}^t, \ldots, u_{in}^t\right]$. To generate this population, a crossover operator is applied as follows:

Let $\lambda \in [1, n]$ be a random integer number, $r \in [0, 1]$ be a uniform random number, $C \in [0, 1]$ be a user-defined crossover constant, each element of $U_i^t = \left[u_{i1}^t, u_{i1}^t, \ldots, u_{in}^t\right]$ is generated according the following equation:

$$u_{ij}^t = \begin{cases} u_{ij}^t \text{ if } r \leq C \text{ or } j = \lambda \\ x_i^{t-1} \text{ otherwise} \end{cases} \tag{3}$$

The SPV rule is used to obtain the permutation and then evaluate each member U_i^t in the trial population by computing its total tardiness. In our implementation, λ is fixed at 0.05.

3.2.4 Selection

To make selection and determine the members who will survive for the next generation, the fitness of the member X_i^t is compared with the fitness of the member U_i^t :

$$X_i^t = \begin{cases} U_i^t \text{ if } f\left(\pi\left(U_i^t\right)\right) < f\left(\pi_i^t\right) \\ X_i^{t-1} \text{ otherwise} \end{cases} \tag{4}$$

3.2.5 Local Search Hybridization

A hybridization of our DE algorithm is proposed by integrating some local search procedures, in order to enhance its performance. Ten local search schemes are proposed and, in each iteration, only one is used according to a probability P. Given a permutation $\Pi = (\pi(1), \pi(2), \ldots, \pi(n))$, these procedures are the following:

- **Random_Exchange_LS**: Generate two random positions $i,j \in [1, n]$ with $i \neq j$ and then exchange $\pi(i)$ and $\pi(j)$.
- **Exchange_All_LS**: Generate a random position $i \in [1, n]$ and exchange $\pi(i)$ with all k positions in the sequence Π with $k \in [1, n]$, and $k \neq i$.
- **Bloc_Exchange_LS**: Decompose the sequence Π in n div 5 blocs and exchange each job $\pi(i)$ with its symmetric position s in the bloc sequence.
- **Symmetric_Exchange_LS**: Exchange each job $\pi(i)$ with its symmetric position $(n - i + 1)$ in the job permutation Π.
- **Intensive_Exchange_LS**: Generate two random positions i and j with $(i < j)$ and exchange each job $\pi(k)$ in the sub-sequence $(\pi(i), \ldots, \pi(j))$ with all the remaining jobs.
- **Insertion_LS**: Remove a randomly chosen job $\pi(i)$ and insert it at a new position j $(i \neq j)$.
- **Circular_Insertion_LS**: Place the job in the first position $\pi(1)$ at the last position n and then shift the remaining jobs. This step is repeated $(n - 1)$ times.

- **Adjacent_Swap_LS**: Starts from a randomly chosen position k and exchange $\pi(k)$ with the job $\pi(k+1)$ and then increment k by 2 ($k = k+2$). These steps are repeated until $k = n-1$.
- **Bloc_Swap_LS**: Decomposing the sequence in n div 5 blocs and then exchange two adjacent jobs in the same bloc $\pi(k)$ with the job $\pi(k+1)$.
- **NEH_Based_LS.**

3.3 Ant Colony Optimization Algorithm

Ant Colony Optimization (ACO) is an evolutionary algorithm based on the behavior of ants. In order to mark some shortest path between ant nest and food that should be followed by other colony ants, these ants deposit a chemical substance called pheromone.

ACO algorithm has been proposed in the early nineties by Dorigo et al. [38]. This meta-heuristic belongs to ant colony algorithm family introduced by Dorigo [39] to solve the Travel Salesman Problem. Several variants of ACO algorithms have since been proposed to solve various optimization problems.

The outline of our ACO algorithm is as follows:

Algorithm 4. ACO

Step 0: Initialize pheromone trails, $t = 0$

Step 1:

Repeat

. Apply **Exchange_All_LS** for all ants

. Apply ants state transition rules

. Apply local updating rule

. Apply **Exchange_All_LS** for all ants

. Apply global updating rule

. Apply **Insertion_LS** for the best ants

. $t = t+1$

Until (the maximum number of iterations is reached)

All the steps of our ACO are described as follows.

3.3.1 Pheromone Trails Initialization

The ACO algorithm starts by initializing the pheromone trails. Let N be the number of ants (population size), so initially N ants are randomly generated. Let X_i^t denotes the i^{th} member in the population at the generation t with $X_i^t = \left[x_{i1}^t, x_{i2}^t, \ldots, x_{in}^t\right]$, n is the number of jobs and x_{ij}^t is a continuous random number $x_{ij}^0 = x_{\min} + (x_{\max} - x_{\min}) \cdot r$ with $r \in [0,1], x_{\min} = -1$ and $x_{\max} = 1$. It is worth noting that for finding permutation the same method as our DE is used.

After that, three matrices are initialized:

- The τ matrix of the pheromone density for each ant, where τ_{ij} is the density of pheromone between two jobs i and j. All $\tau_{ij} = \tau_0$ where τ_0 is the initial rate of pheromone, and $\tau_0 = \dfrac{1}{(1-r_0) \cdot T_{best}}$ where r_0 is a random chosen constant and T_{best} is the best fitness value found.

- The δ matrix of the distance for each ant, where δ_{ij} is the distance between two jobs i and j, and $\delta_{ij} = p_{1j} + p_{2j}$.
- The η matrix of heuristic value for each ant, where $\eta_{ij} = \dfrac{1}{\delta_{ij}}$ is the inverse of the distance between two jobs i and j.

3.3.2 The State Transition Rule

To move from job i to job j, the ants use a decision rule known as *State Transition Rule (STR)* or *Pseudo-random Proportional Rule (PPR)*. The probability P_{ij}^t for an ant to move from job i to job j depends on a random variable q uniformly distributed over $[0, 1]$, and a parameter q_0:

$$j = \begin{cases} \text{argmax}\left(j \in \pi_i^t\right)\left\{\left[\tau_{ij}\right]^{\alpha}\left[\eta_{ij}\right]^{\beta}\right\} \text{ if } q \leq q_0 \\ P_{ij}^t \text{ otherwise} \end{cases} \text{ with } P_{ij}^t = \begin{cases} \dfrac{\left[\tau_{ij}\right]^{\alpha}\left[\eta_{ij}\right]^{\beta}}{\sum_{j \in \pi_i^t}\left[\tau_{ij}\right]^{\alpha}\left[\eta_{ij}\right]^{\beta}} \text{ if } j \in \pi_i^t \\ 0 \text{ otherwise} \end{cases} \tag{5}$$

α and β are parameters controlling the relative importance of the pheromone *vs*. $\eta_{ij} = 1/\delta_{ij}$.

3.3.3 The Local Updating Rule

Local updating rule favors the exploration of other solutions to avoid premature convergence. This rule is performed after each step by all the ants as follows:

$$\tau_{ij} = (1 - \kappa) \cdot \tau_{ij} + \kappa \cdot \tau_0 \tag{6}$$

where $\kappa \in [0, 1]$ and τ_0 are the decay coefficient and the initial value of the pheromone, respectively. Then for each ant *Intensive_Exchange_LS* is applied as local search procedure.

3.3.4 The Global Updating Rule

At the end of each tour, a global updating rule is applied only by the ant finding the shortest way as follows:

$$\tau_{ij} = (1 - \rho)\tau_{ij} + \rho \cdot \Delta\tau_{ij} \tag{7}$$

where $\Delta\tau_{ij} = \dfrac{1}{T_{best}}$ if $(i,j) \in \pi^*$ with π^* is the global best sequence, T_{best} is the best fitness found, and $\rho \in [0, 1]$ is a pheromone evaporating parameter.

3.4 Imperialist Competitive Algorithm

Imperialist Competitive Algorithm (ICA) is an evolutionary algorithm introduced by Atashpaz-Gargari et al. [40]. It's a population-based algorithm inspired by the imperialistic competition. The imperialism consists of expanding the powerful of a country (known as imperialist) by dominating others countries (called colonies) to take control of their resources. So, several imperialists compete for taking possession of colonies of each other. Each individual in the population is called country and can be imperialist or colony. All these colonies are divided among the imperialists according to their power to form empires. Then each colony start moving toward their relevant imperialist country. The total power of an empire depends mainly on the power of the imperialist country and has a negligible effect by the power of the colonies. After that the imperialistic competition between these empires starts and some empires increase their power. However, the powerless ones can't increase their power and will be eliminated from the competition.

Our proposed ICA algorithm follows these steps:

Algorithm 5. ICA

Step 0. Generate the initial empires, $t = 0$

Step 1. Apply local search for all imperialists

Step 2. Assimilation

Step 3. Compute the total cost of each empire

Step 4. Competition

Step 5. The powerless empires elimination

Step 6. Apply local search for all imperialists

Step 7. $t = t + 1$ and Goto **Step 1 Until** all colonies are under the control of the most powerful empire, or the maximum number of iterations is reached.

3.4.1 Initial Empires Generation

The ICA begins by generating the initial empires. Let N be the number of empires (the population size), and N_{imp} be the number of imperialists and N_{col} be the number of colonies with $N = N_{imp} + N_{col}$. So initially, N countries are randomly generated. Let C be a country is the set of N members with $C^i = \left[X_1^i, X_2^i, .., X_N^i\right]$. Let X_g^i denotes the i^{th} member in the population at the generation t with $X_i^t = \left[x_{i1}^t, x_{i2,}^t \ldots, x_{in}^t\right]$, n is the number of jobs and x_{ij}^t is a continuous random number where $x_{ij}^0 = x_{\min} + (x_{\max} - x_{\min}) \cdot r$ with $r \in [0, 1], x_{\min} = -1$ and $x_{\max} = 1$. To find permutation the same approach as in our DE is used.

Then the N_{imp} most powerful countries are selected as imperialists and the remaining N_{col} countries as colonies. To generate the initial empires, the colonies among imperialists are divided based on their power. So, the initial number of colonies of each empire must be proportionate to its power. Then the normalized cost Φ_{imp} of an imperialist *imp* is defined by $\Phi_{imp} = \varphi_{imp} - \max\{\varphi_i\}$ where φ_{imp} is the cost of the imperialist *imp* and Φ_{imp} its normalized cost, respectively.

Therefore, the normalized power Ψ of each imperialist presents the approximate number of colonies that should be possessed by that imperialist and is defined by:

$$\Psi_{imp} = \left| \frac{\Phi_{imp}}{\sum_{i=1}^{N_{imp}} \varphi_i} \right| \tag{8}$$

Also, the initial number of colonies of an empire *imp* is: $N_{imp} = round\left\{\Psi_{imp} \cdot N_{col}\right\}$. So, for each empire N_{imp} colonies are randomly selected.

3.4.2 Local Search Procedures

For each imperialist, one of the three previously defined local search procedures are applied randomly: *Insertion_Suppression_LS, Intensive_LS* and *NEH_Based_LS.*

3.4.3 Assimilation

The aim of this step is to move the colonies of an empire toward the imperialist. A procedure of perturbation (*NEH_Based_L_S*) is applied to the job permutation of the colony. If the colony's cost is less than the imperialist's cost then we exchange the position of this colony with the relative imperialist.

3.4.4 Total Cost Computing

After that the total cost of an empire is computed as follows. Let TC_{imp} be the total cost of the imp^{th} empire and ξ be a positive number $\xi \in [0,1]$, the total cost is computed as follows:

$$TC_{imp} = Cost\left(imperialist_{imp}\right) + \xi \cdot mean\left(Cost\left(Colonies_{imp}\right)\right) \tag{9}$$

So, the value of the total cost depends on the power of imperialist, and the power of the colony can weakly affect this value if ξ have a large amount and can't affect this value if ξ have a small amount.

3.4.5 Competition

In this step, each empire tries to possess and control the colonies of other empires. As a result of this competition, the colonies of powerless empires will be divided among other imperialists and will not necessarily be possessed by the most powerful empires. To model this competition, first each empire's ownership likelihood is calculated according to its total cost.

Let NTC_{imp} be the normalized total cost of the imp^{th} empire, TC_{imp} its total cost, and P_{imp} be the possession probability of each empire, then $NTC_{imp} = TC_{imp} - \max\{TC_i\}, i = \{1,2,\ldots,N_{imp}\}$ and $P_{imp} = \left|\frac{NTC_{imp}}{\sum_{i=1}^{N_{imp}} NTC_i}\right|$.

Then the vector $P = \left[P_1, P_2, \ldots, P_{N_{imp}}\right]$, the vector $R = \left[r_1, r_2, \ldots, r_{N_{imp}}\right]$ with $r_i \in [0,1]$, and the vector $D = P - R\left[P_1 - r_1, P_2 - r_2, \ldots, P_{N_{imp}} - r_{N_{imp}}\right]$ are computed. So, the empire with the maximum value of D_i will take the colonies. If after some iterations, the powerless empires lose all their colonies so they will be eliminated.

3.5 Genetic Algorithm

Genetic Algorithm (GA) is an evolutionary algorithm that have been successfully applied to different complex combinatorial optimization problems such as scheduling problems, knapsack problems, traveling salesman problem, etc. The GAs were introduced by Holland [41] and are inspired from Darwin's theory of evolution. So, GA approach is based on natural evolution techniques, such as selection, crossover and mutation.

GA is a population-based algorithm, so each member in the population is called chromosome or solution. Starting from an initial population, the goal is to create new populations with better solutions after some iterations (called generation). In each generation, the fitness of every chromosome in the population is computed by an evaluation function to select the most fitted individuals from the current population. Then some genetic operators such as crossover and mutation are applied to these selected chromosomes. This process is repeated until the satisfaction of a termination condition.

To solve our problem, a GA hybridized with local search procedures (known as genetic local search GLS) is used. This algorithm has been developed initially for the two-machine flowshop problem where the objective is to minimize the total completion time [42].

We recall here the pseudo-code of the GLS algorithm.

Algorithm 6. GLS

Step 0. Parameters initialization

Step 1. Population generation/initialization

Step 2. Selection

Step 3. Crossover

Step 4. Mutation

Step 5. Local search

Step 6. Goto **Step 2 Until** the maximum number of iterations is reached.

We The above procedure is adapted as follows. The initial population consists of M solutions. $M-1$ solutions are randomly generated while a single one is generated using H_5 developed by Berrais et al. [4]. M is referred to as the population size. The fitness function consists on computing the total tardiness of a given solution.

4 Experimental Results

In order to evaluate the empirical performance of the proposed algorithms, a large set of computational tests have been undertaken. All these procedures were implemented in C and run on an Intel Core i5 PC (3.60 GHz) with 8 GB RAM.

4.1 Test Problems

Our proposed algorithms were tested on 12 different problem sizes $n \in \{10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 200, 300\}$. The processing times and the release dates are uniformly distributed on $[1, 100]$ and $\left[0, \frac{\sum_{j=1}^{n} p_{1j}}{2}\right]$, respectively. The due dates d_j are generated using the scheme of [43] as follows: $d_j = r_j + \lfloor f \cdot \frac{(p_{1j}+p_{2j})}{2} \rfloor$ where the due date slack factor f where generated from a discrete uniform distribution on $\left[1 - T - \frac{R}{2}, 1 - T + \frac{R}{2}\right]$, and T and R are the tardiness factors of jobs $T \in \{1.5, 2.5, 3.5\}$, and the dispersion factor of due dates $R \in \{0.2, 0.4, 0.6\}$, respectively. By varying T and R, 9 problem classes are obtained. For each class, 30 instances were randomly generated for a total of 3240 test problems. Moreover, to consider the non-availability of machines, 5 holes are randomly generated on each machine.

After an experimental study of the different parameters of our proposed algorithms, the following settings have been used to achieve high-quality solutions:

- PSO: $N = 20, c_1 = 2, c_2 = 2, v_{min} = -4, v_{max} = 4, x_{min} = -4, x_{max} = 4, \beta = 0.975, w = 0.9$
- DE: $N = 20, x_{min} = -1, x_{max} = 1, F = 0.5, \lambda = 0.05$
- ACO: $N = 20, x_{min} = -1, x_{max} = 1, \alpha = 0.1, \beta = 0.1, \varphi = 0.1, \rho = 0.1$
- GLS: The same parameters as in [42]
- ICA: $N = 150, N_{imp} = 10, N_{col} = 140, \xi = 0.05$

For all algorithms, the maximum number of iterations is fixed experimentally according to the size of the problem ($n \times 50$).

4.2 Performance of the Proposed Algorithms

To compare our algorithms' performance, the average relative percentage deviation (ARPD) from the best-known solution is used. This percentage deviation is defined as $ARPD = \left[\frac{UB - UB^*}{UB^*}\right] \times 100$, where UB is the solution provided by PSO, DE, ACO, GLS or ICA and $UB^* = min(UB_i); (i = 1, \ldots, 5)$.

Tabs. 1 and 2 summarize the results of the computational experiments.

The analysis with respect to the problem size in Tab. 1 clearly reveals that the best results among all tested algorithms are found by the proposed ICA algorithm. In fact, in comparison to the remaining meta-heuristics this algorithm provides lower values of ARPD, for all problem sizes. Among the 3240 tested problems, ICA gives the best solutions for 1440 instances. It can solve very large instances with up to 300 jobs within a moderate CPU time. The average time in all instances of 300 jobs was 1628.9 s. It can see that the average CPU time grew significantly as the number of jobs increased, especially for the ICA and GLS algorithms. This is due to the fact that the number of jobs directly increases the number of iterations of the algorithms and thereby increasing the computation time.

Table 1: Performance of the proposed evolutionary algorithms

n	*PSO*		*GLS*		*DE*		*ICA*		*ACO*	
	ARPD	*Time*	*ARPD*	*Time*	*ARPD*	*Time*	*ARPD*	*Time*	*ARPD*	*Time*
10	0.000	0.0	0.197	0.1	0.000	0.0	0.000	0.2	0.002	0.0
20	0.025	0.0	0.126	0.3	0.002	0.0	0.000	1.0	0.034	0.1
30	0.060	0.0	0.113	1.0	0.013	0.0	0.000	2.9	0.068	0.4
40	0.070	0.1	0.057	2.4	0.020	0.0	0.000	6.3	0.070	0.7
50	0.076	0.1	0.053	5.3	0.024	0.1	0.000	11.4	0.071	1.5
60	0.091	0.2	0.059	10.6	0.037	0.1	0.000	20.9	0.080	2.6
70	0.081	0.3	0.047	20.1	0.033	0.1	0.000	35.3	0.067	4.2
80	0.073	0.3	0.034	31.5	0.033	0.2	0.000	48.4	0.068	6.2
90	0.068	0.5	0.037	402.9	0.031	0.2	0.000	79.3	0.060	8.6
100	0.073	0.7	0.025	72.4	0.037	0.3	0.000	93.5	0.067	11.3
200	0.046	5.2	0.014	1117.2	0.025	1.5	0.001	542.0	0.045	86.4
300	0.042	17.0	0.011	6733.4	0.028	4.2	0.004	1628.0	0.042	302.2
Average	0.059	2.0	0.064	699.8	0.024	0.6	0.000	205.8	0.056	35.4

Regarding the computational results of the GLS algorithm, we observe that it gives very good results for large size problems ($n > 90$). In fact, the performance of GLS algorithm draws near significantly from the ICA algorithm for $n > 200$. Unfortunately, it shows the worst results for small size problems ($n \leq 30$). For $n = 10$, all the proposed algorithms, except the GLS, present the best performances in term of ARPD and the DE algorithm shows the best CPU time (0.005 s). Furthermore, the DE algorithm shows better results than GLS for small and medium size instances ($n \leq 90$). Also, we observe that in term of ARPD no significant differences between the PSO and ACO algorithms were found, particularly for large size instances ($n \geq 200$).

Table 2: Performance of the proposed algorithms with respect to the due dates' factors

Class	*PSO*		*GLS*		*DE*		*ICA*		*ACO*	
	ARPD	*Time*	*ARPD*	*Time*	*ARPD*	*Time*	*ARPD*	*Time*	*ARPD*	*Time*
T = 1.5, R = 0.2	0.050	2.5	0.058	545.2	0.021	0.5	0.002	227.1	0.047	36.2
T = 1.5, R = 0.4	0.049	1.8	0.057	688.5	0.021	0.5	0.000	214.3	0.046	34.1
T = 1.5, R = 0.6	0.049	1.8	0.057	681.6	0.020	0.5	0.000	189.8	0.046	34.0
T = 2.5, R = 0.2	0.058	2.5	0.063	1213.2	0.024	0.5	0.000	200.3	0.054	34.0
T = 2.5, R = 0.4	0.057	1.8	0.063	554.7	0.023	0.5	0.000	205.1	0.054	34.0
T = 2.5, R = 0.6	0.057	1.8	0.063	847.4	0.022	0.5	0.000	208.9	0.054	34.0
T = 3.5, R = 0.2	0.071	2.0	0.073	681.1	0.027	0.6	0.000	197.1	0.068	36.1
T = 3.5, R = 0.4	0.070	2.0	0.074	528.1	0.027	0.6	0.000	201.1	0.068	36.9
T = 3.5, R = 0.6	0.069	2.0	0.074	558.3	0.026	0.6	0.001	208.1	0.067	39.0
Average	0.059	2.0	0.064	699.8	0.024	0.6	0.000	205.8	0.056	35.4

Tab. 2 shows the ARPD of the proposed algorithms with respect to the combinations for the values of T and R. Our computational experience shows that the performance of the algorithms was affected considerably by the tardiness factor T. In fact, we observe that the ARPD increases when the tardiness factor T increases. The proposed algorithms were able to provide a lower ARPD for the instances which present lower tardiness factor ($T = 1.5$).

Regarding the due date range factor R, the results reveals that the ARPD of the proposed algorithms was slightly affected. Also, we observe the same remark for the CPU time except for the GLS algorithm. In fact, for this meta-heuristic the computation time increases considerably in particular for the ($T = 2.5, R = 0.2$) and ($T = 2.5, R = 0.6$) problem classes.

5 Conclusions

In this paper, we have proposed five evolutionary algorithms developed to minimize the total tardiness in a two-machine flowshop scheduling problem where the machines and jobs are subject to non-availability constraints and unequal release dates, respectively. To avoid the rapid convergence of these algorithms and in order to derive high-quality solutions, we proposed a hybridization of our algorithms with several local search procedures. Computational tests provide strong evidence, on the tested instances, that the Imperialist Competitive Algorithm (ICA) outperforms all the proposed population-based approaches.

Funding Statement: The authors received no specific funding for this study.

Conflicts of Interest: The authors declare that they have no conflicts of interest to report regarding the present study.

References

[1] M. L. Pinedo, *Scheduling: theory, algorithms, and systems*, 4th ed., New York, USA: Springer, 2012.

[2] C. Y. Lee, "Minimizing the makespan in the two-machine flowshop scheduling problem with an availability constraint," *Operations Research Letters*, vol. 20, no. 3, pp. 129–139, 1997.

[3] J. K. Lenstra, A. H. Rinnooy Kan and P. Brucker, "Complexity of machine scheduling problems," *Annals of Discrete Mathematics*, vol. 1, no. C, pp. 343–362, 1977.

[4] Berrais, M. A. Rakrouki and T. Ladhari, "Minimizing total tardiness in two-machine machine permutation flowshop problem with availability constraints and subject to release dates," in *Proc. of the 14th Int. Workshop on Project Management and Scheduling*, Munchen, Germany, pp. 32–35, 2014.

[5] Y. D. Kim, "Minimizing total tardiness in permutation flowshops," *European Journal of Operational Research*, vol. 85, no. 3, pp. 541–555, 1995.

[6] V. A. Armentano and D. P. Ronconi, "Tabu search for total tardiness minimization in flowshop scheduling problems," *Computers and Operations Research*, vol. 26, no. 3, pp. 219–235, 1999.

[7] S. Hasija and C. Rajendran, "Scheduling in flowshops to minimize total tardiness of jobs," *International Journal of Production Research*, vol. 42, no. 11, pp. 2289–2301, 2004.

[8] C. S. Chung, J. Flynn and Ö. Kirca, "A branch and bound algorithm to minimize the total tardiness for m-machine permutation flowshop problems," *European Journal of Operational Research*, vol. 174, no. 1, pp. 1–10, 2006.

[9] F. Ghassemi Tari and L. Olfat, "COVERT based algorithms for solving the generalized tardiness flow shop problems," *Journal of Industrial and Systems Engineering*, vol. 2, no. 3, pp. 197–213, 2008.

[10] F. Ghassemi Tari and L. Olfat, "A set of algorithms for solving the generalized tardiness flowshop problems," *Journal of Industrial and Systems Engineering*, vol. 4, no. 3, pp. 156–166, 2010.

[11] C.-S. Chung, J. Flynn, W. Rom and P. Staliński, "A genetic algorithm to minimize the total tardiness for m-machine permutation flowshop problems," *Journal of Entrepreneurship, Management and Innovation*, vol. 8, no. 2, pp. 26–43, 2012.

[12] G. Liu, S. Song and C. Wu, "Some heuristics for no-wait flowshops with total tardiness criterion," *Computers and Operations Research*, vol. 40, no. 2, pp. 521–525, 2013.

[13] F. Ghassemi Tari and L. Olfat, "Heuristic rules for tardiness problem in flow shop with intermediate due dates," *International Journal of Advanced Manufacturing Technology*, vol. 71, no. 1–4, pp. 381–393, 2014.

[14] V. Fernandez-Viagas, J. M. Valente and J. M. Framinan, "Iterated-greedy-based algorithms with beam search initialization for the permutation flowshop to minimise total tardiness," *Expert Systems with Applications*, vol. 94, no. 3, pp. 58–69, 2018.

[15] K. Karabulut, "A hybrid iterated greedy algorithm for total tardiness minimization in permutation flowshops," *Computers and Industrial Engineering*, vol. 98, no. 4, pp. 300–307, 2016.

[16] T. Sen, P. Dileepan and J. N. Gupia, "The two-machine flowshop scheduling problem with total tardiness," *Computers and Operations Research*, vol. 16, no. 4, pp. 333–340, 1989.

[17] C. Koulamas, "A guaranteed accuracy shifting bottleneck algorithm for the two-machine flowshop total tardiness problem," *Computers and Operations Research*, vol. 25, no. 2, pp. 83–89, 1998.

[18] J. Schaller, "Note on minimizing total tardiness in a two-machine flowshop," *Computers and Operations Research*, vol. 32, no. 12, pp. 3273–3281, 2005.

[19] M. Kharbeche and M. Haouari, "MIP models for minimizing total tardiness in a two-machine flow shop," *Journal of the Operational Research Society*, vol. 64, no. 5, pp. 690–707, 2013.

[20] G. Schmidt, "Scheduling with limited machine availability," *European Journal of Operational Research*, vol. 121, no. 1, pp. 1–15, 2000.

[21] Y. Ma, C. Chu and C. Zuo, "A survey of scheduling with deterministic machine availability constraints," *Computers and Industrial Engineering*, vol. 58, no. 2, pp. 199–211, 2010.

[22] J. Blazewicz, J. Breit, P. Formanowicz, W. Kubiak and G. Schmidt, "Heuristic algorithms for the two-machine flowshop with limited machine availability," *Omega*, vol. 29, no. 6, pp. 599–608, 2001.

[23] F. Ben Chihaoui, I. Kacem, A. B. Hadj-Alouane, N. Dridi and N. Rezg, "No-wait scheduling of a two-machine flow-shop to minimise the makespan under non-availability constraints and different release dates," *International Journal of Production Research*, vol. 49, no. 21, pp. 6273–6286, 2011.

[24] J. Y. Lee and Y. D. Kim, "Minimizing total tardiness in a two-machine flowshop scheduling problem with availability constraint on the first machine," *Computers and Industrial Engineering*, vol. 114, pp. 22–30, 2017.

[25] J. Y. Lee, Y. D. Kim and T. E. Lee, "Minimizing total tardiness on parallel machines subject to flexible maintenance," *International Journal of Industrial Engineering: Theory Applications and Practice*, vol. 25, no. 4, pp. 472–489, 2018.

[26] J. Kennedy and R. C. Eberhart, "Discrete binary version of the particle swarm algorithm," in *Proc. of the IEEE Int. Conf. on Systems, Man and Cybernetics*, Orlando, FL, USA, 5, pp. 4104–4108, 1997.

[27] X. Zhang, W. Zhang, W. Sun, X. Sun and S. K. Jha, "A robust 3-D medical watermarking based on wavelet transform for data protection," *Computer Systems Science and Engineering*, vol. 41, no. 3, pp. 1043–1056, 2022.

[28] H. Moayedi, M. Mehrabi, M. Mosallanezhad, A. S. A. Rashid and B. Pradhan, "Modification of landslide susceptibility mapping using optimized PSO-ANN technique," *Engineering with Computers*, vol. 35, no. 3, pp. 967–984, 2019.

[29] H. Jiang, Z. He, G. Ye and H. Zhang, "Network intrusion detection based on PSO-Xgboost model," *IEEE Access*, vol. 8, pp. 58392–58401, 2020.

[30] M. A. Shaheen, H. M. Hasanien and A. Alkuhayli, "A novel hybrid GWO-PSO optimization technique for optimal reactive power dispatch problem solution," *Ain Shams Engineering Journal*, vol. 12, no. 1, pp. 621–630, 2021.

[31] J. C. Bansal, M. Arya and K. Deep, "A nature inspired adaptive inertia weight in particle swarm optimization," *International Journal of Artificial Intelligence and Soft Computing*, vol. 4, no. 2–3, pp. 228, 2014.

[32] M. Nawaz, E. E. Enscore and I. Ham, "A heuristic algorithm for the m-machine, n-job flow-shop sequencing problem," *Omega*, vol. 11, no. 1, pp. 91–95, 1983.

[33] R. Storn and K. Price, "Differential evolution-a simple and efficient heuristic for global optimization over continuous spaces," *Journal of Global Optimization*, vol. 11, no. 4, pp. 341–359, 1997.

[34] J. Lampinen and I. Zelinka, "Mechanical engineering design optimization by differential evolution," in *New Ideas in Optimization*, D. Corne, M. Dorigo, F. Glover (eds.), UK: McGraw-Hill, pp. 127–146, 1999.

[35] G. Onwubolu and D. Davendra, "Scheduling flow shops using differential evolution algorithm," *European Journal of Operational Research*, vol. 171, no. 2, pp. 674–692, 2006.

[36] M. F. Tasgetiren, Y. C. Liang, M. Sevkli and G. Gencyilmaz, "A particle swarm optimization algorithm for makespan and total flowtime minimization in the permutation flowshop sequencing problem," *European Journal of Operational Research*, vol. 177, no. 3, pp. 1930–1947, 2007.

[37] J. Beans, "Genetics and random keys for sequencing and optimization," *INFORMS Journal on Computing*, vol. 6, no. 2, pp. 154–160, 1993.

[38] M. Dorigo and L. M. Gambardella, "Ant colony system: a cooperative learning approach to the traveling salesman problem," *IEEE Transactions on Evolutionary Computation*, vol. 1, no. 1, pp. 53–66, 1997.

[39] M. Dorigo, "Optimization, learning and natural algorithms," Ph.D. dissertation, Polytechnic University of Milan, Italy, 1992.

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

[41] J. H. Holland, *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. USA: University of Michigan Press, 1992.

[42] T. Ladhari and M. A. Rakrouki, "Heuristics and lower bounds for minimizing the total completion time in a two-machine flowshop," *International Journal of Production Economics*, vol. 122, no. 2, pp. 678–691, 2009.

[43] Y. D. Kim, "A new branch and bound algorithm for minimizing mean tardiness in two-machine flowshops," *Computers and Operations Research*, vol. 20, no. 4, pp. 391–401, 1993.