Research Inventy: International Journal of Engineering And Science
Vol.10, Issue 9 (September 2020), PP 06-17
Issn (e): 2278-4721, Issn (p):2319-6483, www.researchinventy.com

The Permutation Flowshop Scheduling Problem: An Efficient Genetic Algorithm

José Lassance C. Silva[1], Gerardo Valdisio R. Viana[2], Bruno Castro H. Silva[3]
[1] *Federal University of Ceará, Fortaleza-Ceará, Brazil*
[2] *State University of Ceará, Fortaleza-Ceará, Brazi*
[3] *Federal University of Ceará, Campus Crateús, Crateús-Ceará, Brazil*
Corresponding Author: lassance@lia.ufc.br

Abstract: *This paper proposes a Genetic Algorithm (GA) to solve the permutation flowshop scheduling problem with the makespan criterion. The problem has important applications in industrial systems. The main contribution of this study is due to the fact that new genetic operators were used such as mutation operator which intensifies the search to find good solutions and a new intensification criteria to escape from local minima. In addition, The GA is not hybridized as does most of the resolution methods used recently to solve the problem. The GA took into account the diversification and intensification of the search to solve the problem. Computational experiments are reported for the literature instances and the obtained results are compared with other techniques.*

Keywords: *Combinatorial Optimization, Evolutionary Computation, Metaheuristic, Heuristic, Sscheduling.*

Date of Submission: 04-09-2020 Date of Acceptance: 19-09-2020

I. INTRODUCTION

Flowshop scheduling problems focus on processing a given set of jobs, where all jobs have to be processed in an identical order on a given number of machines. Among all types of scheduling problems, flowshop scheduling has important applications in different industries. The objective of the permutation flowshop scheduling problem (PFSP) is to find a job sequence to minimize *the makespan* or *the total flow time* (TFT). References [1-5] show a statistical review of flowshop scheduling research, in the last decades. In this paper, we worked the PFSP with the makespan criterion, where it was extensively addressed in the literature [6-8]. Ceberio *et al.* [9] contains the state-of-the-art for the PFSP with respect to the TFT criterion.

The PFSP input is given by a matrix P with m x n non-negative elements, where P_{ik} is associated with job J_k processing time in the machine M_i. Following the four parameters $A/B/C/D$ [1] notation, the problem is classified as $n/m/P/F_{max}$. The problem is $F/\text{prmu}/C_{max}$ within [3], suggested a classification $\alpha/\beta/\gamma$. The PFSP is also known to be NP-Complete in the strong sense for $m \geq 3$ [2]. However, for $m = 2$ it can be exactly solved in polynomial time. There is an extensive literature survey – in past decades – to the problem and the authors indicate the existence of more than 1,200 Operations Research papers on it, also containing a large diversity of aspects and applications, references [4-5].

The *Permutation Flowshop Scheduling Problem* is defined as given a set of n jobs, $J_1, J_2, \ldots, J_n$, to be processed by m machines $M_1, M_2, \ldots, M_m$. Each job demands m operations and every job has to obey the same operation flow, *i.e.* job J_k for $k = 1,2,\ldots, n$ is processed first in machine M_1, then in machine M_2 and so forth up to M_m. If job J_k does not use all the machines its processing flow continues to be the same but with time zero whenever that happens. A machine processes just a single job and once it is started it cannot be interrupted up to its completion. It is worth of mentioning that the total search space of possible sequences to be considered is very large and it is bounded by above by $O(n!)$. A solution to the problem consists in processing all the n jobs in the least possible time. Suppose that the permutation $\pi=\{\pi_1, \pi_2, \ldots, \pi_n\}$ represents the schedule of jobs to be processed. The finishing time of every operation can be calculated from the following expression:

$$\begin{aligned} &c(1,\pi_1) = P(1,\pi_1); \\ &c(1,\pi_j) = c(1,\pi_{j-1}) + P(1,\pi_j); \\ &c(i,\pi_1) = c(i-1,\pi_1) + P(i,\pi_1); \\ &c(i,\pi_j) = \max\{c(i-1,\pi_j), c(i,\pi_{j-1})\} + P(i,\pi_j); \\ &\text{where } i = 2,3,\ldots,m \text{ and } j = 2,3,\ldots,n. \end{aligned} \tag{1}$$

Then the makespan of permutation π can be defined as

$$C_{\max}(\pi) = c(m, \pi_n) \quad (2)$$

The permutation flowshop scheduling problem with the makespan criterion is to find a permutation π^* in the set of all permutations Π such that $C_{\max}(\pi^*) \leq C_{\max}(\pi)$, $\forall\ \pi \in \Pi$.

In the practice, the exact solution methods for the problem are still limited to small instances, n ≤ 20 and even to them the running time continues to be large. The heuristics are the primary way to solve the problem. Simple heuristic strategies may be based on applying priority based dispatching rules. More effective heuristics represent specific algorithms which are developed for some special type of problem. This problem might be partly solved by applying metaheuristics, which are widely generic with respect to the type of problem. In past decades, most research focused on developing heuristic and metaheuristics algorithms. These solution techniques can be broadly classified into two groups referred as constructive heuristics and improvement method (metaheuristics). In the first group, heuristics have been developed for the makespan criterion by Palmer [10], Campbel *et al.* [11], Newaz, Enscore, and Ham (*NEH*) [12], Ruiz and Maroto [13], among others. These algorithms can obtain the near-optimum solutions in a short period, but the qualities of solutions are not satisfactory. NEH is one of most effective constructive heuristic algorithms, which first assigned priorities on each job based on the total processing time and then inserted jobs successively into the sequence to obtain a complete scheduling. The second group has grown quickly with the advance of modern computers. Since 1995, many metaheuristics have been developed and appropriate for solving the problem, among which we highlight those that had the best performance: genetic algorithm of Chen *et al.* (GAC) [14], of Reeves (GARe) [15], of Murata *et al.* (GAM) [16], of Ruiz *et al.* (GARu) [17], hybrid differential evolution algorithm (HDE) [18], estimation of distribution algorithm (EDA) [19], discrete differential evolution algorithm (DDE) [20], hybrid differential evolution algorithm of Liu *et al.* (LHDE) [21], and self-guided differential evolution with neighborhood search (SGDE) [8]. These metaheuristics also had better performance than the heuristics of the first group.

GAC (1995), GARe (1995), GAM (1996), and GARu (2006) are traditional genetic algorithms, each characterized by their genetic operators (population initialization, crossover, mutation, selection strategy, and stopping criteria). GARu had the best performance among them when applied in the Taillard's problems [22]. Qian *et al.* (2008) [18] proposed HDE algorithm which employed a largest-order-value rule to convert the continuous value to job permutations and used a problem-dependent local search to enhance exploitation. The convergence property of the HDE was analyzed based on Markov chains. Jarboui *et al.* (2009) [19] presented an EDA with a probabilistic model to determine the probability of the jobs to be in determined positions. Pan *et al.* (2007) [20] combined the optimization mechanism of the differential evolution algorithm with the feature of the PFSP, and proposed DDE algorithm which introduced mutation and crossover operators based on permutations. Liu *et al.* (2014) [21] proposed a hybrid differential evolution named LHDE which combines the differential evolution with the individual improving scheme and greedy-based local search. Shao and Pi (2016) [8] presented a series of studies and proposed SGDE algorithm which combines differential evolution with a self-guided mechanisms and some local searching strategy. They used two neighborhood searches based on the variable neighborhood search (VNS) are designed to enhance the local searching ability, which includes Insert_VNS and Swap_VNS. Their computational results showed to be superior to several of the best heuristics and metaheuristics reported in the literature, in terms of quality of the search, robustness and efficiency. Their computational experiments confirmed that the SGDE algorithm was more effective than all other algorithms. SGDE is by far the best metaheuristic algorithm for solving the PFSP, with makespan criterion.

The PFSP has been extensively studied over the last decades, it is still a challenge to find optimal solutions to large instances of the problem in a reasonable amount of time. This paper presents a different genetic algorithm (DGA) without the use of the following strategies: (i) initialization effectively followed by (ii) a hybridization step with a search technique known. These strategies are common features of the genetic algorithms used for solving the PFSP. The computational results show that the DGA algorithm generated better results than those metaheuristics reported previously. All computational results were based on the same benchmark instances taken from Reeves [15] and Taillard [22] with the makespan criterion. The DGA algorithm decreases by 51% the average deviation of the SGDE algorithm in the 21 instances of Reeves.

The remaining contents of this paper are organized as follows. Section 2 gives the details of the proposed DGA algorithm and design of experiments for parameter setting. The computational results over benchmark problems are discussed in Section 3. Finally, Section 4 summarizes the concluding remarks.

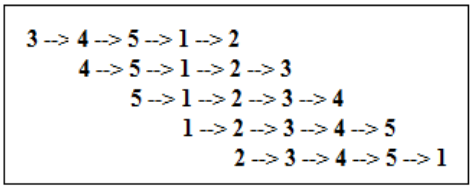
II. THE GENETIC ALGORITHM

The first step in applying genetic Algorithm (GA) to a particular problem is to convert the feasible solutions of that problem into a string type structure called chromosome. In order to find the optimal solution of a problem, a standard GA starts from a set of assumed or randomly generated solutions (chromosomes) called initial population and evolves different but better sets of solution (chromosomes) over a sequence of generations (iterations). In each generation the objective function (fitness measuring criterion) determines the suitability of

each chromosome and, based on these values, some of them are selected for reproduction. For the no-wait flowshop scheduling problem we take the fitness value of each chromosome to be the reciprocal of the makespan, using (2). The number of copies reproduced by an individual parent is expected to be directly proportional to its fitness value, thereby embodying the natural selection procedure, to some extent. The procedure thus selects better (highly fitted) chromosomes and the worse can be eliminated. Genetic operators such as crossover and mutation are applied to these (reproduced) chromosomes and new chromosomes (offspring) are generated. These new chromosomes constitute the next generation. These iterations continue still some termination criterion is satisfied. The best chromosome evaluated is presented as the optimal solution of the problem. A good reference for understanding how GA work is Goldberg [18].

Two principles were used to guide the construction of DGA: diversification and intensification. Michell [24] described the evolution of the solutions depends of the variation in the abilities of the individuals in the population (i.e. diversification of solutions). Already intensification in the search process tends to improve the quality of the final solutions, in [25]-[26]. Three different procedures were developed for the DGA based on these two principles. The first procedure allows regenerate individual with worse fitness through mutation operator. The second procedure is the creation of a new type of crossover operator. Finally, the third procedure allows modifying the population to concentrate the search in new regions. The components of the DGA are given below.

Figure 1. Solutions obtained of s_3=<3 4 5 2 1>.



2.1 Solution Representation

The chromosome (an individual of the population) is defined as a permutation of the *n* jobs. The representation used for a solution of the problem is a permutation of the set J ={J_1, J_2, ..., J_n}, where the relative order (from left to the right) of the jobs indicates the processing order of the jobs on the machines.

2.2 Population Initialization

The generation of the initial population is the main criterion to deal with the diversification. If the initial population is not well diversified, a premature convergence can occur for GA. A set of *Npop* initial individuals or chromosomes form an initial population, where *NPop* represents the population size. The DGA has an initial population generated by the *PopInic* procedure. The *NPop* best solutions of this procedure are inserted in the initial population in ascending order (makespan). Thus, the best individuals in the population are I_1, I_2, ..., I_{NPop}, in this order. The *PopInic* procedure produces $n \times n$ feasible solutions to the problem, where $NPop < n^2$. It is equivalent to *nearest neighbor heuristic* fairly applied to the Traveling Salesman Problem, where if the best neighbor of job *j* is the job *k*, then processing the job *k* immediately after the job *j*, leaves the lower idle in the last machine (M_m). Initially, *PopInic* produces *n* distinct solutions (s_1, s_2, ..., s_n), where each solution is created taking into consideration this criterion, and begin by job k=1, 2, …, n. After this, (n-1) solutions are generated from each solution s_k as follow: s_k^1=< s_{k2}, s_{k3}, s_{k4}, …, s_{kn}, s_{k1}>, s_k^2=< s_{k3}, s_{k4}, …, s_{kn}, s_{k1}, s_{k2}>, …, s_k^{n-1} =< s_{kn}, s_{k1}, s_{k2}, …, s_{kn-1}>. For example, if n=5 e s_3=<3 4 5 1 2>, the four solutions are <4 5 1 2 3>, <5 1 2 3 4>, <1 2 3 4 5>, e <2 3 4 5 1>. Fig. 1 shows how the four solutions were obtained.

2.3 The Fitness Function

The individuals are all evaluated by using (2).

2.4 Selection Strategy

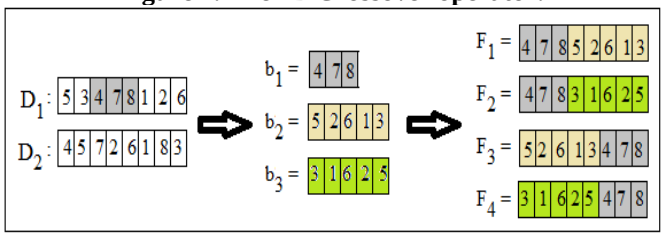
The selection strategy for crossover and mutation is made with all individuals of the population.

2.5 Crossover and Mutation

Crossover is a genetic operation to generate a new sequence (i.e., child) from its parent strings. It has a great influence on the performance of genetic algorithm. The crossover operator exchanges the information of the selected parents to generate promising offspring or sequences. It can be used to generate a set of new solutions or offspring between two solutions from the set. The idea behind crossover is that the new chromosome may be better than both of the parents if it takes the best characteristics from each of the parents. The crossover operator is applied to all individuals in the population, making (NPop × (NPop-1))/2 applications

for each type of crossover operator. Four crossover operators were used in DGA: order crossover with one-point (1P), order crossover with two-point (2P), partially mapped crossover (PM), and order crossover with two blocks (2B). The operators 1P, 2P and PM are widely used in evolutionary computation and can be found in [15], [23] and [27]. We developed the 2B operator. The two cutting points c1 and c2 used in 2P, PM and 2B are given as $c_1=\lfloor n/3 \rfloor+1$ and $c_2=2\times\lfloor n/3 \rfloor + 1$. The cutting point c_1 used in 1P is given as $c_1=\lfloor n/2 \rfloor$. We did not use the random generation of these points for the algorithm proposed because we want to split the chromosome in approximately three equal parts when we used 2P, PM and 2B, and in approximately two equal parts, when used 1P.

Figure 2. The 2B Crossover operator.



We developed the 2B operator on the idea of replicating the good built blocks. In addition, it increases the number of solutions generated and evaluated by DGA, diversifying the search to find the optimal solution of the problem. The Fig. 2 illustrates this procedure, where each parent D_1 and D_2 is divided into three blocks. The cutting points that generate the blocks are c_1 and c_2, specified above. Taking D_1 as the base, the blocks b_1, b_2 and b_3 generate four offspring (children): F_1=<b_1 b_2>, F_2= <b_1 b_3>, F_3= <b_2 b_1>, and F_4= <b_3 b_1>. The block b_1 is the part between the two cutting points (central block D_1 – gray color). The block b_2 is formed by elements that are not in b_1 and they are placed in the order of their appearance in D_2. The block b_3 consists of the elements of block b_2, with their order reversed. The same procedure is repeated for D_2 being the base, generating more four children. The traditional genetic algorithms generally use the *insertion* or *swapping* operators as the mutation operator. In this work, a new mutation operator is proposed in order to intensify the search, regenerating the solutions considered worse quality. The purpose of this operator is to build good solutions from a particular solution combined with the best solution found up to that moment of the search. The mutation operator used in the DGA works as follows.

Figure 3. Application of the mutation operator.

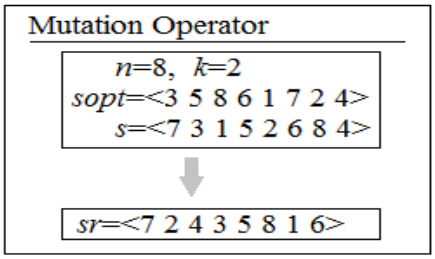


Fig. 3. shows an application of the mutation operator with n=8, k=2, *sopt*=<3 5 8 6 1 7 2 4> (solution defined as the *regeneration solution*), and *s*=<7 3 1 5 2 6 8 4> (a solution of the current population to be regenerated), it produces the solution *sr*=<7 2 4 3 5 8 1 6> (*solution regenerated* by the application of the mutation operator). Initially, k adjacent positions of the regeneration solution are copied for the solution regenerated. Thereafter, the copies can be alternate positions of the regeneration solution, if certain jobs of k adjacent positions have already been copied to the solution regenerated. Furthermore, the copy can also be made with v positions of the regeneration solution ($v<k$). This happens when the number of jobs, from a certain position of the regeneration solution, yet not copied from the regeneration solution is less than k. In the example, the regenerate solution was obtained as follows. Let s_1 be the first job of the solution s (s_1=7). Insert s_1 into first position of solution *sr* (*sr* = <7>). Find the position j of job s_1 in solution *sopt* (j=6). Let r=(r_{j+1}, r_{j+2}, ..., r_{j+k}) be a partial sequence of *sopt* with k adjacent positions, after position j (r=(2, 4)). Insert r into *sr* (*sr*=<7 2 4>). This process is repeated for other jobs of the solution s, checking which of them are not in *sr*. Thus, the next iterations show the following results {s_2=3, j=1, r=(5, 8), *sr*=<7 2 4 3 5 8>}, {s_3=1, j=5, r=(), *sr*=<7 2 4 3 5 8 1>}, {s_6=6, j=4, r=(), *sr*=<7 2 4 3 5 8 1 6>}. Another important feature of this technique is that the regeneration solution can be updated whenever a better solution is found. After several tests, the DGA is running with k=1, 2, ..., $\lfloor n/2 \rfloor$. The Fig. 4 shows the algorithm of the mutation operator.

2.6 Replacement Strategy

The replacement strategy is responsible for controlling the replacement of individuals from one generation to the next in the population. The size of the population is constant (*NPop*). The proposed strategy for our AG is fully replacing all individuals of the population by the best individuals found in the application of the crossover and mutation operators. Acting this way, the DGA intends to diversify and intensify further the search to find the optimal solution of the problem.

Figure 4. The mutation operator.

```
Procedure Mutation
Input: n, k, sopt, s;
Output: sr (solution regenerated);

for i= 1 to n do
    set(i)=0; sr(i)=0;
endfor
k1=0;
for i= 1 to n do
    if (set(s(i))==0) then
        j=1; k1=k1+1;
        while (j<=n) do
            if (s(i)==sopt(j)) then
                a=j; j=n;
            endif
            j=j+1;
        endwhile
        sr(k1)=s(i); set(s(i))=1;
        j=a+1; k2=1;
        while ((j<=n) and (k2<=kk)) do
            if (set(sopt(j))==0) then
                k1=k1+1; k2=k2+1;
                sr(k1)=sopt(j); set(sopt(j))=1;
            endif
            j=j+1;
        endwhile
    endif
endfor
```

2.7 Stopping Criteria

Many stopping criteria based on the evolution of a population may be used. Some of them use the following conditions to determine when to stop: *generations* (when the number of generations reaches the value of generations), *time limit* (after running for an amount of time in seconds equal to time limit), *fitness limit* (when the value of the fitness function for the best point in the current population is less than or equal to fitness limit), *stall generations* (when the average relative change in the fitness function value over stall generations is less than function tolerance), *function tolerance* (The algorithm runs until the average relative change in the fitness function value over stall generations is less than function tolerance), among other conditions. The algorithm stops when any one of these conditions is met.

Initially, the DGA used the following criteria: *generations*, *time limit* and *stall generations*. Several analyzes were performed with the execution of the algorithm applied to different instances of the problem, where it was observed that the algorithm never stopped for the values of the first two variables (*generations*=*n*) and (*time limit*=7200 seconds, when n<100). The average value of the makespan (fitness) of individuals of the current population was used for the third variable. It was compared with the average value of the immediately preceding population. When these values are equal to at least 2 consecutive iterations or *time limit*>7200, the algorithm stops and presents the best solution found to the problem. This feature prevents the evaluation of solutions that can be distinct from those already generated and analyzed, but with the same performance. This greatly reduced the algorithm runtime. With these three criteria, the DGA was converging very fast, mainly for problems with n<100, and failing to appreciate other regions of the search space, compromising the process of diversification. Thus, we decided to adopt a radical change in the individuals of the last population and restart the search, applying the steps of the genetic algorithm again. The individuals obtained in the last population may not be a good enough solutions, since there can be better solutions in the neighborhood of each individual. Therefore, a local search method was adopted to further improve the current solutions.

In particular, two local search methods were applied to find more promising solutions in the solution space through changing neighborhood structures during the search process. A neighborhood is usually defined based on moves of jobs, the search process can benefit much from suitably selected moves. Among various types of moves considered in the literature, *insert* and *swap* moves are most commonly used for the PFSP. The neighborhood based on insert moves is defined by enumerating all possible pairs of positions $i, j \in \{1, \ldots, n\}$ in sequence s ($i \neq j$), where job s_i is removed and then reinserted at position j of s. The neighborhood based on swap moves is defined analogously, which considers exchanging the positions of the two jobs i e j in the sequence. In this work, we modified these two moves to k consecutive jobs ($1 \leq k \leq \lfloor n/2 \rfloor$). The *k-insert* moves considers removing k consecutive jobs from positions i, i+1, …, i+k-1 (i=1, …, n+1-2k) and re-inserting them together into positions j, j+1, …, j+k-1 (j=1, …, n+1-k) in the same order.

The *k-swap* moves is defined in a similar manner, which considers exchanging the positions of k consecutive jobs in the sequence. The makespan of each new individual is performed and compared to the best makespan found and individuals of the population *I*, i.e., after the formation of the new solution, the evaluation is made using (2) and comparing your fitness with the fitness of the individual of population *I*, being able to former a new population. Other significant change was made in the insert and swap moves, when one of the moves finds a better solution than current optimal solution, then this solution becomes the new solution *s*. The variables *dij*, *e1* and *e2* were used in k-insert moves algorithm to prevent the generation e evaluation of repeated solutions. The procedures of the two moves are described in the Fig. 5.a and 5.b.

Figure 5. Pseudocode of the *k-swap* (a) and *k-insert* (b) moves.

(a)

```
Procedure SwapK
Input: NPop, n, I, s_opt, cust_opt;
Output: I (New population);

for v=1 to NPop do
  s=I(v);
  for k=1 to ⌊n/2⌋ do
    for i=1 to (n+1-2*k) do
      for j=i+k to (n+1-k) do
        b=0;
        while (b<k) do
          a=s(i+b); s(i+b)=s(j+b); s(j+b)=a; b=b+1;
        endwhile
        s(0)=M(s); // Calculate the makespan of s
        if (s(0)<cust_opt) then
          Update I, s_opt, and cust_opt;
        else
          b=0;
          while (b<k) do
            a=s(j+b); s(j+b)=s(i+b); s(i+b)=a; b=b+1;
          endwhile
        endifelse
      endfor
    endfor
  endfor
endfor
```

(b)

```
Procedure InsertK
Input: NPop, n, I, s_opt, cust_opt;
Output: I (New population);

for v=1 to NPop do
  s=I(v);
  for k=1 to ⌊n/2⌋ do
    e1=k-1; e2=(-1)*e1;
    for i=1 to (n-(k-1)) do
      for j=1 to (n-(k-1)) do
        dij=i-j;
        if (dij!=0 and dij!=k and (dij<e2 or dij>e1)) then
          for b=1 to n do sc(b)=s(b) endfor
          if (i<j) then
            for b=i+k to (j+k-1) do sc(b-k)=s(b) endfor
          else
            for b=j+k to (i+k-1) do sc(b)=s(b-k) endfor
          endifelse
          for b=0 to k-1 do sc(j+b)=s(i+b);
          sc(0)=M(sc); // Calculate the makespan of sc
          if (sc(0)<cust_opt) then
            s=sc; Update I, s_opt, and cust_opt;
          endif
        endif
      endfor
    endfor
  endfor
endfor
```

The Fig. 6 illustrates the pseudocode of our genetic algorithm. The best solution (*s_opt*) is evaluated within the *initial population*, *crossover*, *mutation* and *local search* procedures. The crossover and mutation population is allocated to array *ICM*. The makespan of the individual *v* is stored in its first component (I_{v0} and ICM_{v0}). The components of *avg1* and *avg2* contain the average makespan of the populations *I* and *ICM*. These values are calculated for individuals of the current population and immediately preceding population. Likewise, the components of *avgP* store these values for the populations used in the local search method. The DGA algorithm was designed to run in four different ways based on local search methods: k-insert (*GAI*), k-swap (*GAS*), k-swap with k-insert in this order (*GASI*), and k-insert with k-swap (*GAIS*). The crossover operator 2P

only starts when the crossover operator 1P is finalized for all individuals of the population *I*. Likewise, 2P, PM, and 2B. For each instance, the GAI, GAS, GASI, and GAIS algorithms were evaluated only once, and the best result was attributed to the DGA. The CPU time of DGA is equal to the average time of GAI, GAS, GASI, and GAIS algorithms. Thus, we use the same rules as the non-deterministic algorithms practice.

Figure 6. Pseudocode of the DGA algorithm.

```
Procedure DGA
Input: n, NPop, P;
Output: s_opt (The best solution found);

avg=1; stop=0;
avg1(1)=1; avg1(2)=2;
avg2(1)=2; avg2(2)=1;
avgP(1)=1; avgP(2)=2;
InitialPopulation();
while (stop==0) do
   Crossover_1P();
   Crossover_2P();
   Crossover_PM();
   Crossover_2B();
   Mutation();
   avg1(1)=avg1(2);
   avg2(1)=avg2(2);
   avg1(2)= ∑_{v=1}^{NPop} I(v,0)/NPop;
   avg2(2)= ∑_{v=1}^{NPop} ICM(v,0)/NPop;
   I=ICM;
   if (avg1==avg2) then avg=avg+1;
   if (avg==2) then
      avg=1;
      LocalSearch();
      avgP(1)= avgP(2); avgP(2)=avg1(2);
      if (avgP(1)==avgP(2)) then stop=1;
   endif
endwhile
```

2.8 Calibration of the proposed DGA

We conducted a study with simple experiments to determine an appropriate *Npop* value and select the best crossover operator (*1P*, *2P*, *PM* and/or *2B*). We used the test set of instances with $n \in \{20, 30, 50, 75\}$ e $m \in \{5, 10, 15, 20\}$. They are 21 benchmark instances provide by Reeves [15]. For the makespan criterion, the solution quality was evaluated according to the makespan generated from Shao and Pi [8]. One run were carried out for each problem instance to report the performance based on the percentage relative deviations (*d*), computed as $d=100*(z - r)/r$, where *z* is the makespan generated by DGA and *r* is the optimal solution describe from Shao and Pi [8]. In these tests, the DGA did not use the local search methods. In this moment, the results of the experiments conducted in instances of PFSP were to determine these two parameters specifically.

The Fig. 7 shows the performance of DGA with $NPop \in \{30, 40, ..., 200\}$. It is possible to see that the best result was obtained for *NPop*=200, with *d*= 2.41. The performance of the algorithm was weak to *NPop* values between 30 and 110, with percentage relative deviation greater than 3.0%, and to *NPop* values between 120 and 200, the percentage relative deviation was greater than 2.41% and less than 3.00%. Therefore, we proposed DGA with 200 individuals in the population.

The Table 1 shows the result of DGA when the object of analysis is to determine what type of crossover should be used, alone or combined. It presents in its columns: the instance of the problem; values for *m* and *n*; deviations (*d*) of each crossover operator for each instance; the average, minimum and maximum deviations for each operator; and the number of problems (*BP*) that the operator has better performance. The bold values indicate the best results at each line.

Figure 7. Performance of DGA with 30≤NPop≤200.

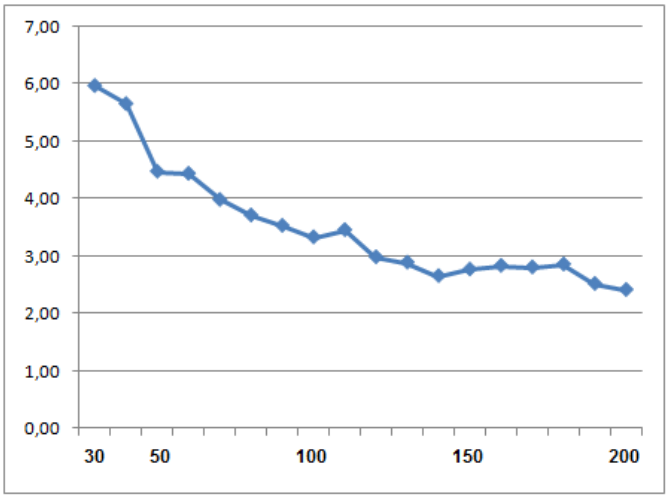


Table 1 – Performance of the Crossover Operators.

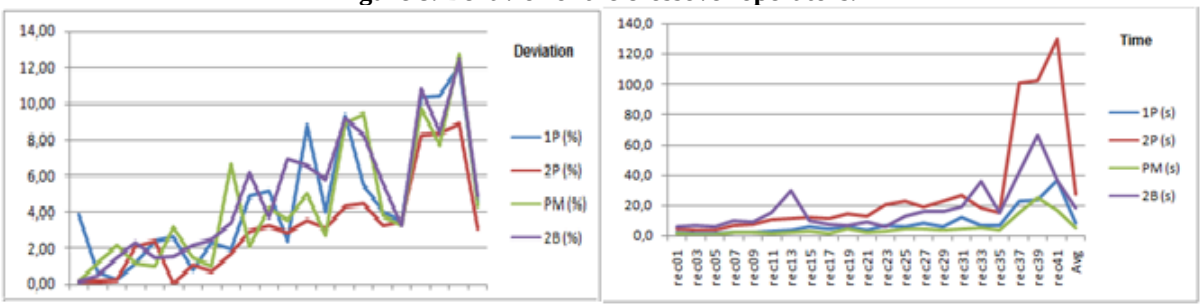
Instance	n x m	1P (%)	2P (%)	PM (%)	2B (%)
rec01	20x5	3.85	**0.16**	**0.16**	**0.16**
rec03	20x5	0.72	**0.18**	1.17	0.45
rec05	20x5	**0.24**	**0.24**	2.17	1.45
rec07	20x10	**1.15**	2.11	**1.15**	2.30
rec09	20x10	2.41	2.41	**0.98**	1.43
rec11	20x10	2.66	**0.00**	3.14	1.54
rec13	20x15	**0.78**	1.09	1.45	2.18
rec15	20x15	2.31	**0.72**	1.03	2.46
rec17	20x15	1.95	**1.68**	6.62	3.42
rec19	30x10	4.83	2.96	**2.10**	6.16
rec21	30x10	5.16	**3.22**	4.26	3.67
rec23	30x10	**2.39**	2.88	3.53	6.96
rec25	30x15	8.79	**3.54**	5.01	6.53
rec27	30x15	4.00	3.08	**2.70**	5.77
rec29	30x15	9.31	**4.33**	8.92	9.14
rec31	50x10	5.48	**4.47**	9.43	8.24
rec33	50x10	4.01	**3.21**	3.79	5.52
rec35	50x10	3.51	3.48	**3.23**	3.27
rec37	75x20	10.32	**8.24**	9.74	10.75
rec39	75x20	10.42	8.32	**7.73**	8.39
rec41	75x20	12.10	**8.85**	12.76	12.40
Average		4.59	**3.10**	4.34	4.87
Minimum		0.24	**0.00**	0.16	0.16
Maximu		12.10	**8.85**	12.76	12.40
BP		4	**13**	7	1

Fig. 8 shows the graphs for the deviation (%) and execution time (seconds) of each operator. It is possible to note the good performance of *2P* and *PM* operators. These two operators obtained the best performers with an average deviation of 3.10% (*2P*) and 4.34% (*PM*). The *2P* operator had the best performance in 13 of the 21 instances tested. The *2P* operator had the best deviation range, with the best minimum (0.00%) and best maximum (8.85%). The *2P* and *2B* operators had the lowest runtimes. These two operators are good to diversify the search, given that they do not compromise the runtime. For these reasons, we decided to adopt the four crossover operators *1P*, *2P*, *PM* and *2B*, in this order of execution, as the crossover operators of the DGA. Other combinations of these four operators were tested and none of them was better than this.

The mutation operator had performance better than other genetic operators in the following instances: *rec15*, *rec27*, *rec31*, and *rec35* (DGA with *1P*); *rec35* (DGA with *2P*); *rec13*, *rec15*, *rec27*, *rec29*, *rec31*, *rec35* and *rec39* (DGA with *PM*); *rec15*, *rec17*, *rec19*, *rec21*, *rec23*, *rec25*, *rec27*, *rec29*, *rec33*, *rec35*, *rec37*, *rec39*, and *rec41* (DGA with *2B*). The mutation and crossover operators have achieved the same results in other

instances.

Figure 8. Behavior of the crossover operators.



III. COMPUTATIONAL EXPERIMENTS

The computational experiments carried out to observe the performance of DGA was executed in a PC Dell with a clock of 3.0 GHz and 4 Gbytes of RAM and the source program is in ANSI C. Table 2 presents the average deviation of the algorithms GAI, GAS, GASI, GAIS, DGA, HDE, LHDE, DDE, and SGDE, in the 21 problem instances of Reeves [15]. The average deviation (d) is the same as given in Section II (item 2.8). A description of the algorithms HDE, LHDE, DDE, and SGDE was given in the fifth paragraph of Section I, remembering that SGDE algorithm is considered the best metaheuristic applied in solving the PFSP. The bold values indicate the best results at each line. Table 3 presents CPU time of GAI, GAS, GASI, GAIS, and DGA. All algorithms were run on different machines, except GAI, GAS, GASI, GAIS and DGA.

Table 2 – Comparison of algorithms with respect to instances of Reeves.

Inst.	HDE	LHDE	DDE	SGDE	GAI	GAS	GASI	GAIS	DGA
rec01	0.16	**0.00**	0.16	**0.00**	**0.00**	0.16	**0.00**	**0.00**	**0.00**
rec03	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	0.18	**0.00**	**0.00**	**0.00**
rec05	0.24	0.24	0.24	0.24	0.24	0.24	**0.00**	0.24	**0.00**
rec07	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	1.15	1.15	**0.00**	**0.00**
rec09	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**
rec11	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**
rec13	0.10	**0.00**	**0.00**	**0.00**	0.26	0.26	0.10	0.16	0.10
rec15	**0.00**	**0.00**	0.05	**0.00**	**0.00**	0.05	**0.00**	**0.00**	**0.00**
rec17	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**
rec19	0.86	0.57	**0.29**	**0.29**	0.81	**0.29**	**0.29**	**0.29**	**0.29**
rec21	1.44	1.44	1.44	1.44	**1.39**	1.44	1.44	1.44	**1.39**
rec23	0.50	0.50	0.55	0.50	0.70	**0.15**	**0.15**	0.45	**0.15**
rec25	1.15	0.96	0.80	0.68	0.36	1.15	**0.24**	0.48	**0.24**
rec27	1.10	0.84	1.10	**0.80**	0.97	1.10	**0.80**	0.97	**0.80**
rec29	1.09	0.57	1.36	0.57	**0.13**	1.14	0.35	1.44	**0.13**
rec31	2.59	1.94	1.35	1.25	**0.26**	0.69	**0.26**	**0.26**	**0.26**
rec33	0.83	0.83	0.83	0.45	0.51	0.96	**0.00**	0.83	**0.00**
rec35	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	0.61	**0.00**	**0.00**	**0.00**
rec37	4.48	3.43	3.27	2.89	2.36	2.81	1.72	**1.51**	**1.51**
rec39	3.11	2.83	1.65	1.69	1.00	1.95	1.45	**0.71**	**0.71**
rec41	4.33	3.43	3.15	2.84	1.88	2.44	1.73	**1.09**	**1.09**
Avg	1.047	0.837	0.773	0.650	0.518	0.798	0.461	0.470	**0.318**
Min.	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**
Max.	4.48	3.43	3.27	2.89	2.36	2.81	1.73	1.51	**1.51**
Effic.	7	9	7	9	8	3	9	8	**10**

The Fig. 9.a and Tables 2 and 3 give some observations about the computational experiments:

a) All algorithms had the lowest minimum value (0.00%). The DGA had the best maximum value (1.51%) and the best performance (Average = 0.318%). DGA had Efficient = 10, i.e., they found the optimal solution in 10 of the 21 problem instances;

b) The DGA obtained 20 (95.2%) satisfactory results whereas the SGDE had 11 (52.4%) satisfactory results, of 21 instances used. Also in this aspect, GASI and GAIS were more efficient than SGDE, with 14 (66.7%) and 13 (61.9%), respectively. GAI obtained 11 (52.4%) satisfactory results. In the average deviation,

DGA, GASI, GAIS, and GAI were better than SGDE, DDE, LHDE, HDE, and GAS. SGDE had one better results than DGA in the instances rec13 whereas DGA was better than SGDE in all other instances;

c) The HDE and LHDE had the worst results, with Avg=1.047% and Avg=0.837%, respectively. They also had the worst results with the maximum value, i.e., their solutions have a greater range (HDE with [0.0, 4.48] and LHDE with [0.0, 3.43]) than the other algorithms for the average deviation values;

d) The DGA, GASI, and GAIS were much better than the other algorithms;

e) The average running time of GAI, GAS, GASI, GAIS and DGA was between 355.1 and 1071.3 seconds;

f) The DGA performance was better than all algorithms, as shown in Fig. 9.a, where a 95% Confidence Interval plot (CI) for the d value under different algorithms is provided.

Figure 9. CI – for instances of *Reeves* (a) and *Taillard* (b).

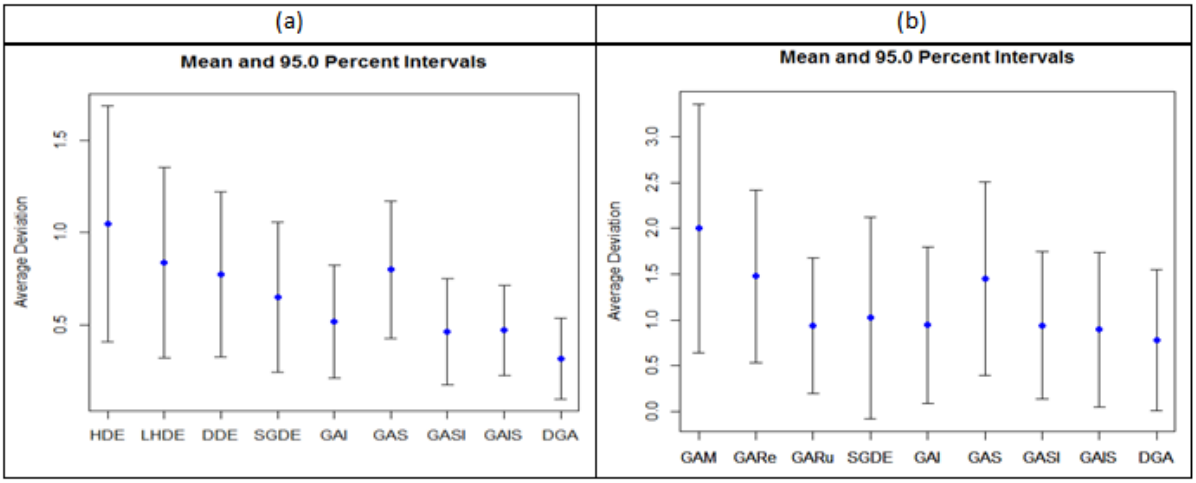


Table 3 – Comparison of algorithms with respect to CPU time.

Instance	GAI	GAS	GASI	GAIS	DGA
rec01	6.3	5.8	6.0	6.0	6.0
rec03	6.8	6.1	6.6	6.0	6.4
rec05	10.1	5.5	5.7	6.6	6.9
rec07	17.6	12.6	8.7	16.4	13.8
rec09	12.0	12.1	13.5	14.5	13.0
rec11	20.6	16.7	21.3	18.7	19.3
rec13	25.1	25.1	33.5	25.0	27.2
rec15	31.7	39.9	29.7	28.4	32.4
rec17	25.2	31.5	22.6	27.8	26.8
rec19	35.4	46.7	77.8	77.0	59.2
rec21	33.6	20.4	34.6	44.3	33.2
rec23	65.4	53.4	40.3	61.3	55.1
rec25	136.3	85.9	141.3	117.3	120.2
rec27	99.0	63.3	79.6	105.0	86.7
rec29	101.6	77.3	96.1	51.4	81.6
rec31	402.4	227.8	307.3	426.5	341.0
rec33	161.9	49.8	217.5	191.2	155.1
rec35	78.4	47.3	112.8	114.1	88.2
rec37	3459.5	2993.5	3745.7	6576.0	4193.7
rec39	3976.3	1426.4	2625.2	7372.0	3850.0
rec41	5083.4	2210.9	4537.5	7211.1	4760.7
Average	656.6	355.1	579.2	1071.3	665.5

The instances from the Reeves benchmark set are of relatively small scale and to further demonstrate the good efficiency of the our algorithms, numerical comparisons were conducted on the Taillard benchmark set [17], which contains a good number of large-scale instances formed by 90 problem instances, divided in 9 classes ($n \times m$): *C1* (20×5); *C2* (20×10); *C3* (20×20); *C4* (50×5); *C5* (50×10); *C6* (50×20); *C7* (100×5); *C8* (100×10); *C9* (100×20). Each class contains 10 problems. We compared the results of GAI, GAS, GASI, GAIS, and DGA with GAM, GARe, GARu, and SGDE. A description of the algorithms GAM, GARe, GARu, and SGDE was given in the fifth paragraph of Section I. Reeves [15], Murata *et al.* [16], and Ruiz *et al.* [17] did not describe the individual solutions found by GARe, GAM, and GARu algorithms, respectively. They presented the results by class. Thus, we could not compare them with the individual solutions found by SGDE, GAI, GAS, GASI, GAIS, and DGA algorithms. The Table IV and Fig. 11 show the computational results for the 9 algorithms, whereas the Table V presents the CPU time of GAI, GAS, GASI, GAIS, and DGA algorithms in these instances.

Table 4 – Comparison of algorithms with respect to Taillard's Instances.

Class	GAM	GARe	GARu	SGDE	GAI	GAS	GASI	GAIS	DGA
C1	0.53	0.62	0.25	**0.00**	0.10	0.53	0.10	0.10	0.10
C2	1.61	1.71	0.64	0.08	0.20	0.50	0.07	0.08	**0.00**
C3	1.36	1.31	0.40	0.07	0.07	0.28	0.09	0.07	**0.05**
C4	0.23	0.16	0.06	**0.03**	0.25	0.46	0.34	0.22	0.15
C5	3.27	2.00	1.46	1.32	1.08	1.94	1.28	1.07	**0.83**
C6	4.75	3.58	**2.47**	3.91	2.94	4.07	2.76	2.74	2.57
C7	0.22	0.11	**0.06**	0.21	0.33	0.59	0.44	0.31	0.31
C8	1.34	0.67	**0.52**	0.58	0.76	1.30	0.76	0.61	0.53
C9	4.68	3.12	2.54	3.01	2.75	3.37	2.62	2.84	**2.47**
Avg	1.999	1.476	0.933	1.025	0.942	1.449	0.941	0.893	**0.778**
Min.				**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**
Max.				4.79	4.33	6.42	4.22	4.21	**4.03**
Effic.				26	20	12	25	27	**32**

Table 5 – The CPU time with respect to Taillard's Instances.

Class	GAI	GAS	GASI	GAIS	DGA
C1	9.0	10.7	10.0	10.1	9.9
C2	27.0	23.8	27.1	28.3	26.5
C3	49.7	41.6	49.9	50.1	47.9
C4	49.5	32.9	74.1	69.9	56.6
C5	229.9	169.4	266.9	263.9	232.5
C6	902.9	658.5	1197.3	1132.5	972.8
C7	548.2	281.2	810.4	806.6	611.6
C8	1679.3	916.4	2124.3	2555.1	1818.8
C9	7281.1	5805.8	7467.3	7486.5	7010.2
Average	1197.4	882.3	1336.4	1378.1	1198.5

The Fig. 9.b and Tables 4 and 5 give some observations about the computational experiments:

a) The SGDE, GAI, GAS, GASI, GAIS, and DGA algorithms had the lowest minimum value (0.00%). The DGA had the best maximum value (4.03%), the best performance (*Average* = 0.778%), and best *Efficient* (32). The DGA found the optimal solution in 32 of the 90 problem instances;
b) The DGA obtained 4 (44.0%) satisfactory results whereas the GARu and SGDE had 3 (33.3%) and 2 (22.2%), respectively. In the average deviation, DGA and GAIS were better than the others algorithms. GARu had three better results in the classes *C6*, *C7*, and *C8*, whereas DGA had four better results in the classes *C2*, *C3*, *C5*, and *C9*. SGDE had two better results in the classes *C1*and *C4*;
c) The GAM (*avg*=1.99%), GARe (*avg*=1.47%) and GAS (*avg*=1.44%) had the worst results. GAS and SGDE had the worst results with the maximum value, i.e., their solutions have a greater range (GAS with [0.0, 6.42] and SGDE with [0.0, 4.79]) than GAI, GASI, GAS, and DGA algorithms for the average deviation values;
d) The average running time of GAI (1197.4), GAS (882.3), GASI (1336.4), GAIS (1378.1), and DGA (1198.5) was less than 23 minutes. The CPU time of the algorithms was less than 51 seconds in the classes

C1 to *C3*, between 170 and 1198 seconds in the classes *C4* to *C6*, and between 282 and 7487 seconds in the classes *C7* to *C9*, these times were quit significant;

e) The DGA performance was better than all algorithms, as shown in Fig. 9.b. We consider very efficient DGA algorithm, because the deviation of the 58 instances which it did not find the optimal solution was less than or equal to 1.00% for 33 instances; between 1.01% and 2.00% for 11 instances; between 2.01% and 3.00% for 10 instances; and between 3.01% and 4.03% for 4 instances.

IV. CONCLUSION

In this paper, a genetic algorithm (DGA) is proposed and applied to the permutation flowshop scheduling problem with the makespan criterion. The genetic algorithm has been widely used in a wide range of applications. The DGA employs a permutation representation and uses the new procedure as the mutation operator. This procedure can regenerate some solutions of low quality. This technique had the good performance to improve the quality of solutions. It is presented here as one scientific contribution. In addition, a new crossover operator (2B) and radical changes in the individuals of the population (to restart the search) have been proposed to DGA.

The computational experiments from the standard benchmarks of the area show that the DGA algorithm is an enough competitive metaheuristic if compared to other metaheuristic. The DGA algorithm has obtained better results than SGDE and GARu (considered the best algorithms applied to the specific problem). The DGA is not hybridized, whereas the other algorithms are hybridized.

The DGA algorithm was applied successfully to the PFSP and it seems reasonable to suppose, at least in a first moment, that it can solve other permutation combinatorial optimization problems. This hypothesis, however, needs further studies on them. An attempt to improve the CPU time of DGA algorithm would be the use of parallel or distributed processing, since it seems that the running time still lies within practical acceptable intervals.

Our next job will be to use the best solution from the DGA algorithm and apply it to the TG algorithm, in [28], as an initial solution. Thus, TG accelerates the tree search to determine the optimal solution for the specific problem. In addition, we will also use the SH algorithm, in [29], to populate the initial DGA population and speed up the search for the optimal solution. The DGA was successfully applied to the No-Wait Flowshop Scheduling Problem, *cf.* [30]. This fact was also verified here when applied to the PFSP.

Acknowledgment

The authors thank the Federal University of Ceará (UFC), the State University of Ceará (UECE) and Cearense Foundation of Research Support (FUNCAP).

References

[1] R. W. Conway, W. L. Maxwell, and L. W. Miller, "*Theory of scheduling*," Addison-Wesley, Reading, Mass., 1967.

[2] M.R. Garey, D.S. Johnson, and R. Sethi, "The complexity of flowshop and jobshop scheduling," *Mathematics of Operations Research*, vol. 1(2), pp. 117–29, 1976.

[3] R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. H. G. R. Kan, "Optimization and approximation in deterministic sequencing and scheduling: a survey", *Annals of Discrete Mathematics*, vol. 5, pp. 287–326, 1979.

[4] A. Reisman, A. Kumar, and J. Motwani, "Flowshop scheduling /sequencing research: A statistical review of the literature, 1952–1994," *IEEE Trans. on Engineering Management*, vol. 44, pp. 316–329, 1997.

[5] J. N. D. Gupta and E. F. Stafford Jr, "Flowshop scheduling research after five decades," *European Journal of Operational Research*, vol. 169, pp. 699–711, 2006.

[6] S. R. Hejazi and S. Saghafianz, "Flowshop scheduling problems with makespan criterion: A review," *Int. J. Prod. Res.*, vol. 43 (14), pp. 2895–2929, 2005.

[7] L. Sheng and X. Gu, "A Genetic Algorithm with Combined Operators for Permutation Flowshop Scheduling Problems," *Proceeding of the IEEE International Conference on Information and Automation*, pp. 65-70, Hailar-China, 2014.

[8] W. Shao and D. Pi, "A self-guided differential evolution with neighborhood search for permutation flowshop scheduling," *Expert Systems with Applications*, vol. 51, pp. 161-176, 2016.

[9] J. Ceberio, E. Irurozki, and A. Mendiburu, "A Distance-Based Ranking Model Estimation of Distribution Algorithm for the Flowshop Scheduling Problem," *IEEE Transactions on Evolutionary Computation*, vol. 18 (2), pp. 286-300, 2014.

[10] D. S. Palmer, "Sequencing jobs through a multistage process in the minimum total time - a quick method of obtaining a near optimum," *Operational Research Quartetly*, vol. 16, pp. 101-107, 1965.

[11] H. G. Campbell, R. A. Dudek, and M. L. Smith, "A heuristic algorithm for the n-job, m-machine sequencing problem," *Management Science*, vol. 16, pp. B630–B637, 1970.

[12] M. Nawaz, E. E. Enscore, and I. Ham, "A heuristic algorithm for the m-machine, n-job flow-shop sequencing problem, " *Omega*, vol. 11 (1), pp. 91–95, 1983.

[13] R. Ruiz and C. Maroto, "A comprehensive review and evaluation of permutation flowshop heuristics," *European Journal of Operational Research*, vol. 165, pp.479-494, 2005

[14] C. L. Chen, V. S. Vempati, and N. Aljaber. "An application of genetic algorithms for flow shop problems," *European Journal of Operational Research*, vol. 80, pp. 389-396, 1995.

[15] C. R. Reeves, "A genetic algorithm for flow shop sequencing," *Computers and Operations Research*, vol. 22, pp. 5-13, 1995.

[16] T. Murata, H. Ishibuchi, and H. Tanaka, "Genetic algorithms for flowshop scheduling problems," *Computers & Industrial Engineering*, vol. 30, pp. 1061-1071, 1996.

[17] R. Ruiz, C. Maroto, and J. Alcaraz, "Two newrobust genetic algorithms for the flowshop scheduling problem," *The International Journal the Management Science* (*Omega*), vol. 34, pp. 461–476, 2006.
[18] B. Qian, L. Wang, R. Hu, W.-L. Wang, D.-X. Huang, and X. Wang, "A hybrid differential evolution method for permutation flowshop scheduling," *The International Journal of Advanced Manufacturing Technology*, vol. 38 (7–8), pp. 757–777, 2008
[19] Y.-M. Chen, M.-C. Chen, P.-C. Chang, and S.-H. Chen, "Extended artificial chromosomes genetic algorithm for permutation flowshop scheduling problems," *Computers & Industrial Engineering*, vol. 62(2), pp. 536–545, 2012.
[20] Q.-K. Pan, M. F. Tasgetiren, and Y.-C. Liang, "A discrete differential evolution algorithm for the permutation flowshop scheduling problem," *In Paper presented at the 9th annual genetic and evolutionary computation conference* (*GECCO2007*), 2007.
[21] Y. Liu, M. Yin, and W. Gu, "An effective differential evolution algorithm for permutation flowshop scheduling problem," *Applied Mathematics and Computation*, vol. 248, pp. 143–159, 2014.
[22] E. Taillard, "Benchmarks for basic scheduling problems," *European Journal Operational Research*, vol. 64(2), pp. 278–285, 1993.
[23] D. E. Goldberg, *GAs in search, optimization and machine learning*, Reading, MA, Addison-Wesley, 1989.
[24] M. Mitchell, *An introduction to Genetic Algorithms*, MIT Press, Cambridge, 1998.
[25] J. Grabowski and J. Pempera, "Some local search algorithms for no-wait flow-shop problem with makespan criterion," *Computers and Operations Research*, vol. 32, pp. 2197-2212, 2005.
[26] J. Dréo, A. Pétrowski, P. Siarry, and E. Taillard, "*Metaheuristics for Hard Optimization: Methods and Case Studies*," Springer, New York, 2006.
[27] E. G. Talbi, "*Metaheuristics from design to implementation*," Wiley, New Jersey, 2009.
[28] J. L. C Silva, L. S. Rocha, and B. C. H. Silva, "A New Algorithm for Finding all Tours and Hamiltonian Circuits in Graphs," *IEEE Latin America Transactions*, vol. 14, pp. 831-836, 2016.
[29] J. L. C Silva, G. V. R. Viana, and B. C. H. Silva, "An efficient algorithm based on metaheuristic for the no-wait flowshop scheduling problem," *Proceedings of the 12th Metaheuristics International Conference* (MIC 2017), Barcelona, Universitat Pompeu Fabra, v. 1. pp. 414-423, 2017.
[30] José Lassance C. Silva, G. V. R. Viana, and B. C. H. Silva, "An Efficient Genetic Algorithm for the No-wait Flowshop Scheduling Problem." *International Journal of Engineering and Science*, vol. 10, no. 08, pp. 32-43, 2020.

José Lassance C. Silva, et. al. "The Permutation Flowshop Scheduling Problem: An Efficient Genetic Algorithm." *International Journal of Engineering and Science*, vol. 10, no. 09, 2020, pp. 06-18.