Research Inventy: International Journal of Engineering And Science
Vol.10, Issue 8 (August 2020), PP 32-43
Issn (e): 2278-4721, Issn (p):2319-6483, www.researchinventy.com

An Efficient Genetic Algorithm for the No-wait Flowshop Scheduling Problem

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 no-wait 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: 30-07-2020 Date of Acceptance: 14-08-2020

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, no-wait flowshop has important applications in different industries such as chemical processing, food processing, concrete ware production, and pharmaceutical processing, [1]-[4]. The no-wait flow shop scheduling problem (NFSP) has the additional restriction that the processing of each job has to be continuous, i.e., once the processing of a job begins, there must not be any waiting times between the processing of any consecutive tasks of this job. The objective of the no-wait flowshop scheduling problem is to find a job sequence to minimize the makespan or the total flow time. References [5-7] show a statistical review of flowshop scheduling research.

Garey and Johnson [8] proved that the computational complexity of the no-wait flowshop scheduling problem is NP-hard. 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 algorithms. These solution techniques can be broadly classified into two groups referred as constructive heuristics and improvement method (metaheu- ristics). In the first group, heuristics have been developed for the makespan criterion by Bonney and Gundry [9], King and Spachis [10], Gangadharan and Rajendran [11], Rajendran [1], Li *et al.* [12]. 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: DPSO [13], IIGA [14], GAVNS [15], and TMIIG [6]. These metaheuristics also had better performance than the heuristics of the first group.

Pan *et al.* presented a series of studies and proposed discrete particle swarm optimization (DPSO) algorithm [13] and an improved iterated greedy algorithm (IIG) [14]. Their computational showed to be superior to several of the best heuristics reported in the literature, in terms of quality of the search, robustness and efficiency. Jarboui *et al.* presented a hybrid genetic algorithm with a variable neighborhood search (GA-VNS) [15] as a procedure to improve the final step of this method. Their computational results showed that GA-VNS algorithm provide competitive results and better performance than DPSO algorithm. Recently, Ding *et al.* proposed a tabu mechanism improved iterated greedy (TMIIG) algorithm [6] for solving the NFSP. Their computational results confirmed that the TMIIG algorithm was more effective than all other algorithms. TMIIG is by far the best metaheuristic algorithm for solving the NFSP, as it is written in [7].

The no-wait flowshop scheduling problem can be described as follows: each of n jobs from set J={1, 2, ..., n} will be sequenced through m machines. Job $j \in J$ has a sequence of m operations (O_{j1}, O_{j2}, ..., O_{jm}). Operation O_{jk} corresponds to the processing of job j on machine k (k=1, 2, …, m) during an uninterrupted processing time $P(j,k)$. At any time, each machine can process at most one job and each job can be processed on at most one machine. To follow the no-wait restriction, the completion time of the operation O_{jk} must be equal to the earliest start time of the operation $O_{j,k+1}$ for k=1,2, ..., m−1. In other words, there must be no waiting time between the processing for any consecutive operations of each of n jobs. The problem is then to find a schedule such that the processing order of jobs is the same on each machine and the maximum completion time is minimized.

Suppose that the job permutation $\pi=\{\pi_1, \pi_2, \ldots, \pi_n\}$ represents the schedule of jobs to be processed. Let $d(\pi_{j-1}, \pi_j)$ be the minimum delay on the first machine between the start of job π_j and π_{j-1} restricted by the no-wait constraint when the job π_j is directly processed after the job π_{j-1}. The minimum delay can be computed from the following expression:

$$d(\pi_{j-1},\pi_j) = P(\pi_{j-1},1) + \max\left[0, \max_{2\le k\le m}\left\{\sum_{h=2}^{k} P(\pi_{j-1},h) - \sum_{h=1}^{k-1} P(\pi_j,h)\right\}\right]. \quad (1)$$

Then the makespan can be defined as

$$\mathrm{M}(\pi) = \sum_{j=2}^{n} d(\pi_{j-1},\pi_j) + \sum_{k=1}^{m} P(\pi_n,\mathrm{k}). \quad (2)$$

The no-wait flowshop scheduling problem with the makespan criterion is to find a permutation π^* in the set of all permutations Π such that $\mathrm{M}(\pi^*) \le \mathrm{M}(\pi)$, $\forall\ \pi \in \Pi$. The NFSP 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 NFSP. 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 [16] and Taillard [17] with the makespan criterion. We highlight that the DGA algorithm did not find the optimal solution in only 1 of the 21 instances of Reeves. We used the optimal solutions found by the exact algorithm of Lin and Ying [7] to compare with the results obtained in our computational experiments.

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 [19] 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 [20]-[21]. 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>.

```
3 --> 4 --> 5 --> 1 --> 2
      4 --> 5 --> 1 --> 2 --> 3
            5 --> 1 --> 2 --> 3 --> 4
                  1 --> 2 --> 3 --> 4 --> 5
                        2 --> 3 --> 4 --> 5 --> 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).

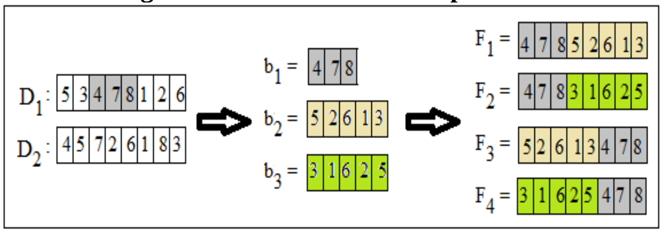
2.4 Selection Strategy

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 [16], [18] and [22]. We developed the 2B operator. The two cutting points c_1 and c_2 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.

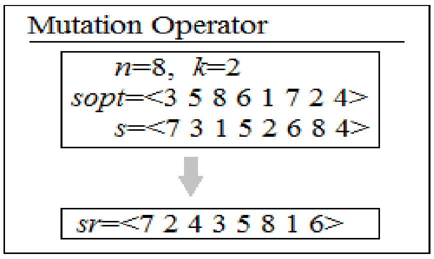


Fig. 3. shows an application of the mutation operator with n=8, k=2, *sopt*=<3 5 8 6 1 7 2 4> (solution defined as the *regeneration solution*), and s=<7 3 1 5 2 6 8 4> (a solution of the current population to be regenerated), it produces the solution *sr*=<7 2 4 3 5 8 1 6> (*solution regenerated* by 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.

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). 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, 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 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.

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
```

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 NWFSP. 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 $\lfloor n/2 \rfloor$ 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 $\lfloor n/2 \rfloor$ 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.

2.8 Calibration of the proposed DGA

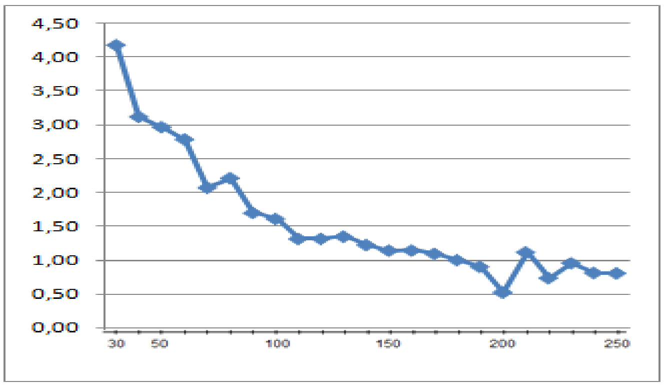
We conducted a study with simple experiments to determine an appropriate *Npop* value and select the best crossover operator (*1P*, *2P*, *PM* and/or *2B*). We used the test set of instances with $n \in \{20, 30, 50, 75\}$ e $m \in \{5, 10, 15, 20\}$. They are 21 benchmark instances provide by Reeves [16]. For the makespan criterion, the solution quality was evaluated according to the reference makespan generated by from Lin and Ying [7]. 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 from Lin and Ying [7]. In these tests, the DGA did not use the local search methods. In this moment, the results of the experiments conducted in instances of NWFSP were to determine these two parameters specifically.

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)= $\sum_{v=1}^{NPop} I(v,0)/NPop$;
    avg2(2)= $\sum_{v=1}^{NPop} ICM(v,0)/NPop$;
    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
```

Figure 7. Performance of DGA with 30≤NPop≤250.



The Fig. 7 shows the performance of DGA with *NPop* ∈ {30, 40, ..., 250}. It is possible to see that the best result was obtained for *NPop*=200, with *d*= 0,52. The performance of the algorithm was weak to *NPop* values between 30 and 170, with percentage relative deviation greater than 1.0%, and to *NPop* values between 210 and 250, the percentage relative deviation was greater than 0.73%. Therefore, we proposed DGA with 200 individuals in the population.

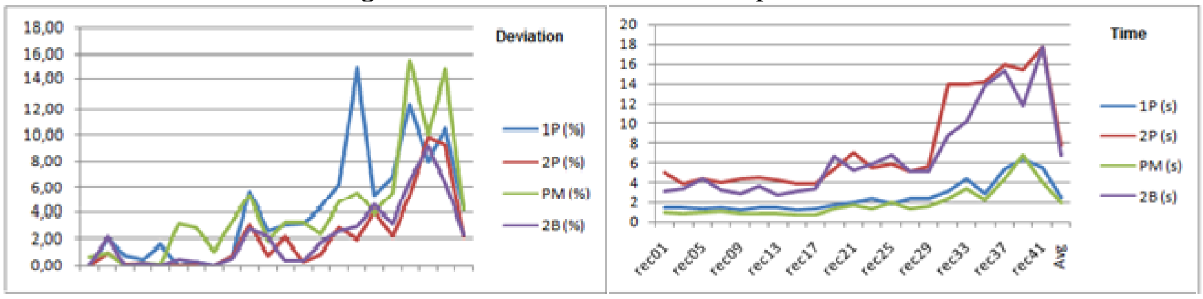
The Table 1 shows the result of DGA when the object of analysis is to determine what type of crossover should be used, alone or combined. It presents in its columns: the instance of the problem; values for *m* and *n*; deviations (*d*) of each crossover operator for each instance; the average, minimum 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.

Table 1 – Performance of the Crossover Operators.

Instance	n x m	1P (%)	2P (%)	PM (%)	2B (%)
rec01	20x5	**0.00**	**0.00**	0.72	0.07
rec03	20x5	2.06	**1.03**	**1.03**	2.13
rec05	20x5	0.86	0.07	**0.00**	**0.00**
rec07	20x10	0.54	0.24	0.24	**0.20**
rec09	20x10	1.62	**0.00**	0.15	**0.00**
rec11	20x10	**0.00**	**0.00**	3.14	0.48
rec13	20x15	0.31	**0.00**	2.87	0.28
rec15	20x15	**0.00**	**0.00**	1.11	0.04
rec17	20x15	0.77	0.77	3.36	**0.62**
rec19	30x10	5.58	3.02	5.30	**2.74**
rec21	30x10	2.59	**0.82**	1.81	2.09
rec23	30x10	3.04	2.15	3.22	**0.41**
rec25	30x15	3.15	**0.31**	3.26	0.45
rec27	30x15	4.26	**0.87**	2.39	1.72
rec29	30x15	6.26	2.86	4.65	**2.55**
rec31	50x10	14.77	**1.90**	5.55	2.90
rec33	50x10	5.29	4.07	**3.91**	4.50
rec35	50x10	6.80	**2.16**	5.48	3.16
rec37	75x20	12.33	**5.26**	15.52	6.52
rec39	75x20	**7.90**	9.79	10.05	9.05
rec41	75x20	10.41	9.13	14.73	**6.26**
Average		4.21	**2.12**	4.21	2.20
Minimum		**0.00**	**0.00**	**0.00**	**0.00**
Maximu		14.77	9.79	15.52	**9.05**
BP		4	**12**	3	8

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 *2B* operators. These two operators obtained the best performers with an average deviation of 2.12% (*2P*) and 2.20% (*2B*). The *2P* operator had the best performance in 12 of the 21 instances tested. The *2B* operator had the best deviation range, with the best minimum (0.00%) and best maximum (9.05%). The *1P* and *PM* 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.

Figure 8. Behavior of the crossover operators.



The mutation operator had performance better than other genetic operators in the following instances: *rec21*, *rec27*, *rec31*, and *rec41* (DGA with *1P*); *rec13*, *rec25*, *rec31*, *rec33* and *rec39* (DGA with *PM*); *rec03*, *rec09*, *rec13*, *rec15*, *rec19*, *rec29*, *rec31*, *rec37*, and *rec39* (DGA with *2B*). The mutation and crossover operators have achieved the same results in other instances.

III. COMPUTATIONAL EXPERIMENTS

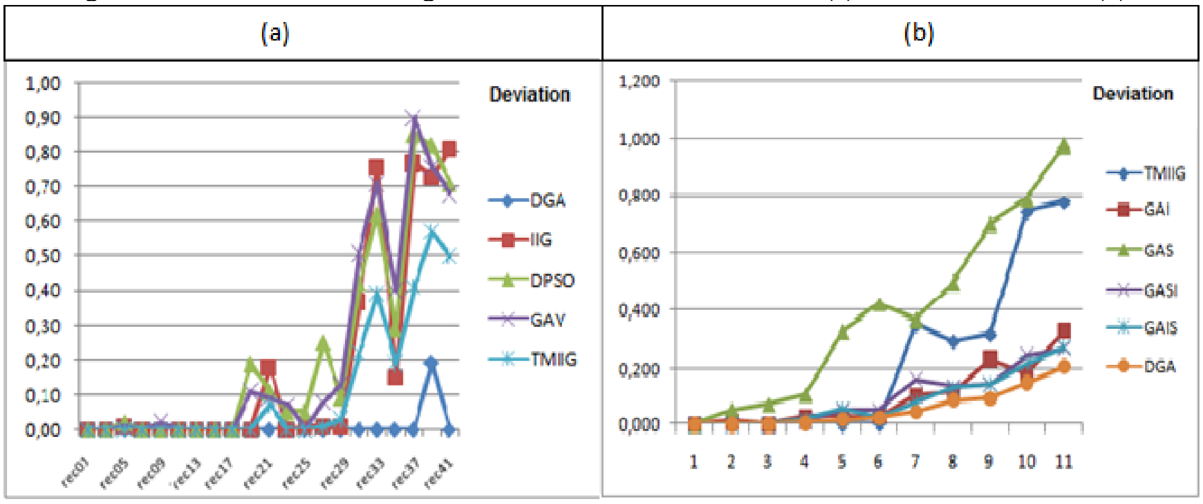
The computational experiments carried out to observe the performance of DGA was executed in a PC Dell with a clock of 3.0 GHz and 4 Gbytes of RAM and the source program is in ANSI C. Table 2 presents the average deviation of the algorithms GAI, GAS, GASI, GAIS, DGA, IIG, DPSO, GAV (GA-VNS) and TMIIG, in the 21 problem instances of Reeves [16]. The average deviation (d) is the same as given in Section II, item 2.8. A description of the algorithms IIG, DPSO, GA-VNS and TMIIG was given in the third paragraph of

Section I, remembering that TMIIG algorithm is considered the best metaheuristic applied in solving the NWFSP. 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. Note that GA-VNS and DPSO are non-deterministic algorithms whereas IIG, TMIIG, GAI, GAS, GASI, GAIS and DGA are deterministic algorithm.

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

Inst.	**GAI**	**GAS**	**GASI**	**GAIS**	**DGA**	**IIG**	**DPSO**	**GAV**	**TMIIG**
rec01	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**
rec03	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**
rec05	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	0.01	0.02	0.01	**0.00**
rec07	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**
rec09	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	0.02	**0.00**
rec11	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**
rec13	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**
rec15	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**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	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	0.19	0.11	**0.00**
rec21	**0.00**	0.25	**0.00**	**0.00**	**0.00**	0.18	0.11	0.09	0.07
rec23	**0.00**	0.07	**0.00**	**0.00**	**0.00**	**0.00**	0.05	0.07	**0.00**
rec25	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	0.01	0.05	**0.00**	**0.00**
rec27	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	0.01	0.25	0.08	0.01
rec29	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	0.01	0.09	0.13	0.03
rec31	**0.00**	0.12	**0.00**	**0.00**	**0.00**	0.37	0.42	0.51	0.22
rec33	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	0.76	0.62	0.71	0.39
rec35	**0.00**	0.02	**0.00**	**0.00**	**0.00**	0.15	0.29	0.40	0.19
rec37	0.25	0.07	**0.00**	0.11	**0.00**	0.77	0.85	0.90	0.41
rec39	0.37	0.30	0.25	0.19	0.19	0.73	0.82	0.76	0.57
rec41	0.06	0.83	**0.00**	0.32	**0.00**	0.81	0.71	0.68	0.50
Avg	0.03	0.08	0.01	0.03	**0.009**	0.18	0.21	0.21	0.11
Min.	0.00	0.00	0.00	0.00	**0.00**	0.00	0.00	0.00	0.00
Max.	0.37	0.83	0.25	0.32	**0.19**	0.81	0.85	0.90	0.57
Effic.	18	14	**20**	18	**20**	10	8	8	12

Figure 9. Performance of the Algorithms in the instances of *Reeves* (a) and *Taillard Classes* (b).



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 (0.19%) and the best performance (*Average* = 0.009%). DGA and GASI had *Efficient* = 20, i.e., they found the optimal solution in 20 of the 21 problem instances, except the instance *rec39*;

b) The DGA obtained 20 (95.2% *efficient*) satisfactory results whereas the TMIIG had 12 (57.1% *efficient*) satisfactory results, of 21 instances used. Also in this aspect, GASI, GAIS, GAI and GAS were more efficient than TMIIG, with efficient of 20 (95.2%), 18 (85.7%), 18 (85.7%) and 14 (66.7%). In the average deviation, DGA, GASI, GAIS, GAI and GAS were better than TMIIG, IIG, DPSO and GA-VNS. TMIIG had two better results than GAS in the instances *rec21* and *rec41*;

c) The DPSO and GA-VNS had the worst results (A*vg*=0.21% and *Efficient*=8). They also had the worst results with the maximum value, i.e., their solutions have a greater range (DPSO with [0.0, 0.85] and GA-VNS with [0.0, 0.90]) than the other algorithms for the average deviation values;
d) The DGA, GASI, GAIS, GAI and GAS were much better than the other algorithms;
e) The average running time of GAI, GAS, GASI, GAIS and DGA was between 29.33 and 41.07 seconds, i.e., less than 1 minute;
f) The DGA performance was better than all algorithms, as shown in Fig. 9.a.

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

Instance	GAI	GAS	GASI	GAIS	DGA
rec01	5.3	4.7	5.5	5.6	5.3
rec03	5.2	4.9	5.3	5.4	5.2
rec05	6.1	6.7	5.7	6.6	6.3
rec07	4.6	4.4	4.7	6.2	5.0
rec09	5.7	6.1	6.0	6.4	6.0
rec11	4.7	4.5	4.9	5.2	4.8
rec13	6.3	6.0	6.5	6.7	6.4
rec15	4.5	3.7	4.6	4.8	4.4
rec17	8.1	7.1	6.8	6.9	7.2
rec19	12.5	13.6	12.5	15.2	13.5
rec21	11.4	11.1	11.6	10.8	11.2
rec23	13.7	13.4	14.6	14.9	14.1
rec25	8.9	8.7	9.7	9.9	9.3
rec27	12.3	14.2	12.6	12.5	12.9
rec29	13.6	12.1	13.2	14.8	13.4
rec31	43.3	39.5	51.5	48.2	45.6
rec33	35.4	30.5	42.7	40.8	37.4
rec35	33.0	30.6	43.1	49.1	38.9
rec37	167.8	101.9	174.8	199.7	161.1
rec39	114.6	175.9	210.7	253.7	188.7
rec41	224.7	116.4	215.6	135.2	173.0
Average	35.3	29.3	41.1	40.9	36.7

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 110 problem instances, divided in 11 classes. Each class contains 10 problems. We compared the results of GAI, GAS, GASI, GAIS, and DGA with the TMIIG algorithm. As Ding *et al.* [6] described the solutions found by TMIIG algorithm for the Taillard benchmark set, we could compare them with the optimal solutions described in the work of Lin and Ying [7]. The Table 4 and Fig. 9.b show the computational results for the 6 algorithms, whereas the Table 5 presents the CPU time of GAI, GAS, GASI, GAIS, and DGA algorithms in these instances.

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

g) All algorithms had the lowest minimum value (0.00%). The DGA had the best maximum value (0.32%) , the best performance (*Average* = 0.057%), and best *Efficient* (59). The DGA found the optimal solution in 59 of the 110 problem instances;
h) The DGA obtained 59 (53.6% *efficient*) satisfactory results whereas the TMIIG, GASI and GAIS had 54 (49.1% *efficient*) satisfactory results. Also in this aspect, GAS was less efficient than all other algorithms. In the average deviation, DGA, GAIS, GASI, and GAI were better than TMIIG and GAS. TMIIG had two better results than DGA in the classes *C5* and *C6*, whereas DGA had 6 better results than TMIIG in the classes *C4*, *C7*, …, *C11*;
i) The TMIIG (*avg*=0.228%) and GAS (*avg*=0.393%) had the worst results. They also had the worst results with the maximum value, i.e., their solutions have a greater range (TMIIG with [0.0, 0.93] and GAS with [0.0, 1.49]) than the other algorithms for the average deviation values;
j) The DGA, GAIS, GASI, and GAI were much better than the other algorithms;
k) The average running time of GAI (1022.6), GAS (574.6), GASI (1629.8), GAIS (1384.1), and DGA (1152.8) was less than 28 minutes. The CPU time of the algorithms was less than 52 seconds in the classes *C1* to *C6*, whereas in the classes *C10* and *C11*, these times were quit significant;

l) 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 51 instances which it did not find the optimal solution was less than or equal to 0.32% for each one of these instances.

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

Class (n x m)	TMIIG	GAI	GAS	GASI	GAIS	DGA
C1 (20 x 05)	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**
C2 (20 x 10)	**0.000**	0.010	0.052	**0.000**	**0.000**	**0.000**
C3 (20 x 20)	**0.000**	**0.000**	0.070	**0.000**	**0.000**	**0.000**
C4 (50 x 05)	0.019	0.028	0.107	0.013	0.016	**0.007**
C5 (50 x 10)	**0.000**	0.031	0.328	0.050	0.055	0.014
C6 (50 x 20)	**0.002**	0.025	0.423	0.050	0.022	0.022
C7 (100 x 5)	0.353	0.104	0.368	0.161	0.079	**0.043**
C8 (100 x 10)	0.296	0.115	0.501	0.137	0.132	**0.087**
C9 (100 x 20)	0.320	0.235	0.701	0.144	0.145	**0.095**
C10 (200 x 10)	0.743	0.186	0.787	0.247	0.219	**0.154**
C11 (200 x 20)	0.778	0.332	0.984	0.268	0.275	**0.211**
Average	0.228	0.097	0.393	0.097	0.086	**0.057**
Minimum	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**
Maximum	0.93	0.48	1.49	0.50	0.57	**0.32**
Efficient	54	49	30	54	54	**59**

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

Class	GAI	GAS	GASI	GAIS	DGA
C1	5.3	5.3	5.5	5.5	5.4
C2	5.5	5.4	6.0	5.8	5.7
C3	5.9	6.1	6.2	5.9	6.0
C4	43.8	34.8	51.1	51.0	45.2
C5	43.4	36.1	45.9	51.1	44.1
C6	41.7	34.6	47.7	49.5	43.4
C7	342.3	205.0	402.4	406.2	339.0
C8	426.4	230.8	416.9	537.0	402.8
C9	363.8	239.2	560.4	532.3	423.9
C10	4396.7	2318.4	8416.4	5478.0	5152.4
C11	5573.4	3205.0	7969.0	8102.5	6212.5
Average	1022.6	574.6	1629.8	1384.1	1152.8

IV. CONCLUSION

In this paper, a genetic algorithm (DGA) is proposed and applied to the no-wait 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 a new mutation operator. This operator has been adapted to evolutionary computation in order 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 TMIIG (considered the best algorithm applied to the specific problem). The DGA is considered for us as a non-traditional genetic algorithm and is not hybridized, whereas the other algorithms are hybridized.

The DGA algorithm was applied successfully to the NWFSP 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 [23], 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 [24], to populate the initial DGA population and speed up the search for the optimal solution.

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] C. Rajendran, “A no-wait flowshop scheduling heuristic to minimize makespan,” *J. Oper. Res. Society*, vol. 45, pp. 472–478, 1994.

[2] N. G. Hall and C. Sriskandarayah, “A survey of machine scheduling problems with blocking and no-wait in process,” *Operations Research*, pp. 44510–44525, 1996.

[3] J. Grabowski and J. Pempera, “Sequencing of jobs in some production system,” *European J. of Oper. Research*, vol. 125, pp. 535–350, 2000.

[4] W. Raaymakers and J. Hoogeveen, “Scheduling multipurpose batch process industries with no-wait restrictions by simulated annealing,” *European J. Oper. Research*, vol. 126, pp. 131–151, 2000.

[5] 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.

[6] J. Y. Ding, S. Song, J. N. D. Gupta, R. Zhang, R. Chiong, and C. Wu, "An improved iterated greedy algorithm with a Tabu-based reconstruction strategy for the no-wait flowshop scheduling problem," *Applied Soft Computing*, vol. 30(5), pp. 604–613, 2015.

[7] S. W. Lin and K. C. Ying, “Optimization of makespan for no-wait flowshop scheduling problems using efficient matheuristics”, *Omega*, 2015, http://dx.doi.org/10.1016/j.omega.2015.12.002.

[8] M. R. Garey and D. S. Johnson, *Computers and intractability: a guide to the theory of NP-completeness*, San Francisco, Freeman, 1979.

[9] M. C. Bonney and S. W. Gundry, “Solutions to the constrained flowshop sequencing problem,” *Operational Research Quarterly*, vol. 24, pp. 869–883, 1976.

[10] J. R. King and A. S. Spachis, “Heuristics for flowshop scheduling,” *Intern. J. of Prouction. Research*, vol. 18, pp. 343–357, 1980.

[11] R. Gangadharan and C. Rajendran, “Heuristic algorithms for scheduling in no-wait flowshop,” *International Journal of Production Economics*, vol. 32, pp. 285–290, 1993.

[12] X. Li, Q. Wang, and C. Wu, “Heuristics for no-wait flow shops with makespan minimization”, *International Journal of Production Research*, vol. 46 (9), pp. 2519-2530, 2008.

[13] Q. K. Pan, M. F. Tasgetiren, and Y. C. Liang, “A discrete particle swarm optimization algorithm for the no-wait flowshop scheduling problem,” *Comp. and Oper. Research*, vol. 35, pp. 2807-2839, 2008.

[14] Q. K. Pan, L. Wang, and B. H. Zhao, “An improved iterated greedy algorithm for the no- wait flowshop scheduling problem with makespan criterion,” *International Journal of Advanced Manufacturing Technology*, vol. 38(7–8), pp. 778–786, 2008.

[15] B. Jarboui, M. Eddaly, and P. Siarry, “A hybrid genetic algorithm for solving no-wait flowshop scheduling problems,” *International Journal of Advanced Manufacturing Technology*, vol. 54(9–12), pp. 1129–114, 2011.

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

[17] E. Taillard, “Benchmarks for basic scheduling problems,” European Journal Operational Research, vol. 64(2), pp. 278–285, 1993.

[18] D. E. Goldberg, *GAs in search, optimization and machine learning*, Reading, MA, Addison-Wesley, 1989.

[19] M. Mitchell, *An introduction to Genetic Algorithms*, MIT Press, Cambridge, 1998.

[20] 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.

[21] J. Dréo, A. Pétrowski, P. Siarry, and E. Taillard, “*Metaheuristics for Hard Optimization: Methods and Case Studies*,” Springer, New York, 2006.

[22] E. G. Talbi, “*Metaheuristics from design to implementation*,” Wiley, New Jersey, 2009.

[23] 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.

[24] 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.

José Lassance C. Silva, et. al. " An Efficient Genetic Algorithm for the No-wait Flowshop Scheduling Problem." *International Journal of Engineering and Science*, vol. 10, no. 08, 2020, pp. 32-43.