Contents lists available at ScienceDirect

Expert Systems With Applications

journal homepage: www.elsevier.com/locate/eswa

An iterated greedy algorithm for the parallel blocking flow shop scheduling problem and sequence-dependent setup times

Imma Ribas [a,*], Ramon Companys [b], Xavier Tort-Martorell [c]

[a] *Departament d'Organització d'Empreses, DOE – ETSEIB - Universitat Politècnica de Catalunya. BarcelonaTech, Avda. Diagonal, 647, 7th Floor, 08028 Barcelona, Spain*
[b] *Universitat Politècnica de Catalunya, BarcelonaTech, Spain*
[c] *Departament de Estadística e investigación Operativa- ETSEIB - Universitat Politècnica de Catalunya. BarcelonaTech, Avda. Diagonal, 647, 6th Floor, 08028 Barcelona, Spain*

A R T I C L E I N F O

Keywords:
Parallel Flow Shop
Sequence-dependent Setup times
Makespan
Distributed Flow Shop
Blocking

A B S T R A C T

This paper deals with the problem of scheduling jobs in a parallel flow shop configuration under the blocking constraint, in which the setup time of machines depends not only on the job to be processed but also on the previously processed one, i.e., there are sequence-dependent setup times. The performance analysis of several iterated greedy algorithms with different initial solution procedures and local searches lets us define an efficient algorithm to minimize the maximum job completion time. Moreover, the computational evaluation showed the efficiency of searching in different neighborhood structures and noted the significant influence of the initial solution. However, contrary to other scheduling problems, starting with a high quality solution does not guarantee better performance of the algorithm.

1. Introduction

There are several productive systems in which more than one flow shop (line) is available to process the products, such as in the glass industry (He et al., 1996), the production process of industrial supplies (Jiang and Wan, 2011) or reprocessing lines (Kim et al., 2015). In other environments, parallel lines are used to increase production capacity in many manufacturing systems. In these productive systems, the scheduling of jobs consists of assigning the products (jobs) to a line and to sequence the products allocated to each line. This problem is known as the Parallel Permutation Flow Shop Scheduling Problem (PPFSP). The PPFSP is different from the classical permutation flow shop scheduling problem (PFSP). PFSP focuses in ordering n jobs that have to be processed in m machines in the same order whereas in the PPFSP two decisions have to be made: assignment of jobs to the lines and sequence the allocated jobs to each line. Hence, it is necessary to design specific methods to solve this problem, especially if some additional constraints are considered in order to bring the problem closer to the reality of production environments.

In this paper, we consider the PPFSP with blocking and sequence-dependent setup times, known as the Parallel Blocking Flow Shop Problem (PBFSP). The blocking constraint appears in several manufacturing environments, such as in productive systems without any intermediate space between machines, where an already processed job by one machine must wait until the next machine is free; or in those where robots move the products and the treated parts must wait in the machine until they can be picked up and moved to the next machine (Sethi et al., 1992). Other examples are found in the iron and steel industry (Gong et al., 2010), industrial waste processing and the production of metallic parts (Martinez et al., 2006). Recently, Miyata & Nagano, (2019) published a comprehensive review of the blocking flow shop problem.

On the other hand, setup times appear when the machines have to be prepared, adjusted, or cleaned between two consecutive jobs. Setup times can be sequence-independent or sequence-dependent. In the former case, the setup time depends on the job to be processed; in the latter, the amount of time also depends on the job that was previously processed in that machine. Setup activities mean a loss of available capacity. Trovinger and Bohn (2005) reported direct savings of $1.7 million per year due to reduced setup times in a printed circuit board assembly plant. According to Allahverdi (2015), considering setup times in scheduling decisions can help increase productivity, reduce both waste and non-value-added activity, improve the use of resources and meet deadlines; yet it is dealt with by only about 10% of the existing

* Corresponding author.
E-mail addresses: imma.ribas@upc.edu (I. Ribas), ramon.companys@upc.edu (R. Companys), xavier.tort@upc.edu (X. Tort-Martorell).

https://doi.org/10.1016/j.eswa.2021.115535
Received 7 March 2021; Received in revised form 29 May 2021; Accepted 29 June 2021
Available online 8 July 2021

literature on scheduling. Hence, in this paper we contribute to reduce this gap by proposing an efficient heuristic to solve the problem of scheduling jobs in a parallel flow shop with blocking and sequence dependent setup times.

To design a procedure to for scheduling jobs in a line with blocking and setup times two aspects should be considered: when the machines can be set up and when the jobs processed by one machine can move on to the next one. There are three possible moments for beginning the setup. Let *h* be the job being processed by the considered machine and *i* the next job to be processed in that machine, so the three cases are: (1) when job h has finished, (2) when job h has left the machine, (3) when job *i* has arrived at the machine. Concerning the moment in which a job can move from its current machine to the next one, two possibilities exist: when job *h* has left the machine or when the setup for processing job *i* has finished. In this work, we consider that setup can be performed when job *h* has left the machine (option 2) and job *i* can move on to the next machine once job *h* has left it, even though the setup is not finished.

To the best of our knowledge, the PBFSP with sequence-dependent setup times has only been studied before by Ribas & Companys (2021), who tested several constructive heuristics to solve the problem. Although constructive heuristics allow finding a solution quickly, improvement heuristics allow this solution to be further enhanced. Hence, this paper aims to propose a simple but effective heuristic to find an efficient solution for the problem, by analyzing the performance of different local searches in the neighborhood of the whole solution (product allocation to lines and line sequence).

The remainder of the paper is organized as follows. Section 2 is devoted to the literature review, Section 3 to the problem definition. Section 4 describes the initial solution methods considered. Section 5 describes different improvement methods. The computational results and comparisons are shown in Section 6. Finally, Section 7 concludes and proposes future lines of research.

2. Literature review

The PPFSP has not been extensively studied. The simple case, where lines have only two machines, was studied by He et al. (1996) who proposed mixed integer programming and an efficient heuristic to minimize the maximum completion time of jobs. With the same criterion in mind, Cao & Chen (2003) developed a mathematical model and a tabu search algorithm; Al-Salem (2004) proposed a new polynomial-time heuristic; Zhang & Van De Velde (2012) presented approximation algorithms with worst-case performance guarantees and Dong et al., (2020) proposed a polynomial-time approximation scheme. The general case (i.e., *m*-machine PPFSP for the makespan minimization) was studied by Jiang & Wan (2011), who proposed a quantum algorithm; Ribas et al. (2017) who compared the performance of an iterated local search algorithm and an iterated greedy algorithm in order to minimize the makespan in the presence of blocking constraints; Tong et al., (2018) , who proposed a polynomial-time approximation algorithm and Ribas & Companys (2021) who proposed constructive heuristics to solve the Parallel Blocking Flow Shop problem (PBFSP) with sequence-dependent setup times. Regarding other performance criteria, Ribas et al. (2019) proposed an iterated greedy algorithm for the PBFSP for the total tardiness minimization.

The solution approach for the PPFSP is similar to that for the Distributed Permutation Flow Shop Scheduling Problem (DPFSP) (i.e., in both problems jobs have to be allocated to a line or factory and then have to be sequenced in the assigned lines or factory), therefore, we also review the methods proposed for solving this problem. The DPFSP has mainly been addressed for minimizing total makespan since Naderi & Ruiz (2010) first presented the problem, for which various metaheuristics have been proposed as solutions. A Variable Neighborhood procedure was proposed by Naderi & Ruiz (2010) and Genetic Algorithms by Gao & Chen (2012), Gao et al. (2012) and Xu et al., (2013). A Tabu Search algorithm was proposed by Gao et al. (2013), a scatter search method by Naderi & Ruiz (2014), a discrete electromagnetism algorithm by Liu & Gao (2010). Additionally, Shao et al., (2020) proposed a fruit fly algorithm for the problem under the blocking constraint and Zhao et al., (2020) an ensemble discrete differential evolution algorithm for the same situation. One of the most frequently proposed algorithm to solve scheduling problems is the Iterated Greedy which, despite its simplicity, has been very effective to find good solutions in several productive configurations. Similarly, some authors have suggestid modified versions for the DBFSP, for example, Fernandez-Viagas & Framinan (2014) proposed bounding the local search, Ruiz et al., (2019) suggested different local searches, Lin et al., (2013) proposed removing a variable number of jobs from the solution as well as a new acceptance scheme. Recently, Huang et al., (2020) recommended to discard the acceptance scheme and to restart the search with an scheme with six operators to deal with the DPFSP with sequence-dependent setup times. Regarding other criteria, Fernandez-Viagas et al., (2018) developed an iterative improvement algorithm to minimize the total flow time.

On the other hand, few papers deal with the scheduling problem taking into account sequence-dependent setup times and machine blockage and most of the research done considers the flow shop configuration and the makespan criterion (Costa et al., 2020; Newton et al., 2019; Shao et al., 2018; Takano & Nagano, 2019). However, Aqil & Allali (2021) and Han et al., (2020) took into account more than one performance criterion during the scheduling. Regarding more complex productive configurations, Zandieh & Rashidi (2009) and Moccellin et al. (2018) considered the hybrid flow shop scheduling problem with blocking and setup to minimize the maximum completion time of jobs. Rashidi et al. (2010) considered not only the makespan minimization but also the maximum tardiness of jobs for the hybrid flow shop scheduling problem with unrelated parallel machines. Finally, Ribas & Companys(2021) considered the Parallel Blocking Flow Shop problem with setup times. Notice that most of these papers have been published very recently, which shows an increasing interest of researchers in putting the scheduling problems closer to reality. In this way, this paper proposes an effective iterated greedy algorithm to scheduling jobs in a parallel flow shop without buffers considering sequence-dependent setup times. It is worth mentioning that special attention is required when the set up times are considered, because the machines' idle time and the total completion time can increase considerably without efficient scheduling.

3. Problem description

The parallel blocking flow shop scheduling problem (PBFSP) consists of scheduling a set of *n* jobs in *F* identical flow shops (lines). Each flow shop has *m* machines. Jobs must be allocated to one of the *f* lines. The jobs assigned to a line have to be processed on all machines in the same order. Each job *i*, $i \in \{1, 2, \ldots, n\}$ requires a fixed processing time $p_{j,i}$ and a setup time, which depends on the previous job processed on every machine *j*, $j \in \{1, 2, \ldots, m\}$. Jobs and machines are available from time zero onwards. The objective is to schedule the jobs in order to minimize the maximum makespan (C_{max}) between lines.

Let n_f be the set of jobs assigned to line *f*; σ_f the sequence of the n_f jobs assigned to line *f*; and f_{max} the line with the maximum C_{max}. Hence, a solution Π is composed of the sequence of jobs in each line ($\Pi=(\sigma_1, \sigma_2, \ldots, \sigma_F)$).

Additionally, we denote as [k] the index of the job in the *k*th position in the sequence and $S_{j,i,k}$ as the sequence-dependent setup times (SDST) needed in machine *j* after processing job *i* and before processing job [k]. This setup can be done once job *i* has left the machine. Let $S_{j,0,k}$ be the setup time required to process the first job in the sequence; $\sigma_{k,f}$ is the job [k] in line *f* and $c_{j,k,f}$ is the departure time of job [k] from machine *j* of line *f*. Note that if machine $j + 1$ is available, job *i* can leave machine *j* when it is completed. In this situation, $c_{j,k,f}$ is the completion time of job [k] in machine *j* of line *f*. The outline of the makespan calculation is shown in Fig. 1.

In Fig. 2, for example, it can clearly be observed that it is necessary to design scheduling methods to minimize setup times, blockages and idle times to minimize the total completion time. When the sequence is J_1, J_2, J_3 the completion time is 19, whereas when the sequence is J_2, J_1, J_3 the C_{max} is 21. This is due to the increase in the setup times required to change from one job to the other, but we can also observe a blockage in M2 due to J_1 not being finished in M3, whereas J_3 has already finished in M2.

4. Iterated greedy algorithm for the PBFSP with sequence-dependent setup times

The PPFSP is an NP-Hard problem if ($n > F$). Therefore, the PBFSP with sequence-dependent setup times is also NP-Hard. As a result, it is necessary to design heuristic methods for obtaining good solutions in a reasonable CPU time.

In this paper, an Iterated Greedy (IG) algorithm is proposed because it is a simple local search method that has proved very efficient for different scheduling problems. It was first applied to the permutation flow shop problem to minimize the makespan by (Ruiz & Stützle, 2007). The good performance showed and its simplicity (IG has very few parameters to calibrate) animated the implementation of this algorithm to other scheduling problems. In particular, it was successfully applied to both the DPFSP (Fernandez-Viagas & Framinan, 2014; Ruiz, Pan & Naderi, 2019) and the PBFSP (Ribas et al., 2017, 2019).

IG algorithm starts with a high quality solution which is improved in an iterative procedure until the stopped criterion is met. The iterative method starts by perturbing the incumbent solution with a destruction and construction phase (perturbation phase). Next, the new solution is tried to be improved by a local search and, finally, the new solution is submitted to the acceptance criterion to decide if it replace the incumbent solution.

Fig. 3 shows the outline of the main structure of the Iterated Greedy (IG) algorithm where Π is the set of sequences in each of the parallel line ($\Pi = (\sigma_1, \sigma_2, \ldots, \sigma_F)$) and $C_{max}(\Pi)$ is the maximum makespan value among lines ($C_{max}(\Pi) = \mathrm{Max}_f\{C_{max}(\sigma_f)\}$).

4.1. Initial solution procedures

Five constructive procedures were used to test the influence of the initial solution on the performance of the algorithms base in the results obtained by Ribas & Companys (2021). The procedures were selected considering both the quality of the solution and the CPU time required since, as is stated in Ruiz et al., (2019), the effect of the initial solution is quickly neutralized by the improvement phase. Hence, the chosen candidates were TRA5, LPT5, RCP0, HPF3 and PF23, ordered according to their performance. The description of each method is as follows:

- **LPT5** sequences the jobs in non-increasing order of their total processing time (the sum of the processing time in each machine). Next, for $i = 1$ to n, assign job i to the line that would finish it at the earliest time if located at the end of its sequence. Place job i at the position that minimizes its C_{max}.
- **TRA5** calculates two indexes for each job (S1i and S2i) according to (1) and (2), respectively. Next, Johnson's algorithm (Johnson, 1954) is used to sequence the jobs by considering S1i and S2i as the processing time of job i in the first and second machine, respectively.

$$S1_i = \sum_{j=1}^{m}(m - j + 1)\cdot p_{j,i} \tag{1}$$

$$S2_i = \sum_{j=1}^{m}(j - 1)\cdot p_{j,i} \tag{2}$$

Finally, for $i = 1$ to n, assign job i to the line that would finish it at the earliest time if located at the end of its sequence. Place job i at the position that minimizes its C_{max}.

- **PF23** creates a sequence in each line by assigning a similar load ($\Sigma P_i/F$). This process starts in one of the lines (all of which are identical) by assigning jobs until it reaches the mean load. The process is repeated for each line. The sequence is created by adapting the PF rule (McCormick, Pinedo, Shenker & Wolf, 1989) to the problem at hand. Therefore, the idle time is calculated by considering that a machine is only idle when it is not processing any job but not when it is being set up. Therefore, the total timeout of machines is calculated as in (3), where i denotes the job, k the position, σ the sequence of jobs already sequenced, and σ^*i the partial sequence plus job i:

$$Tm_k(i) = \sum_{j=1}^{m}\left(c_{j,k+1}(\sigma * i) - c_{j,k}(\sigma) - p_{j,i}\right) \tag{3}$$

The job selected is the one that leads to the minimum timeout.

- **HPF3** is quite similar to PF23, the main difference being in the timeout calculation. In this procedure the sequence in each line is created in order to minimize both the timeout of machines and the total flowtime, which is carried out with the index $ind1(i, k)$ calculated according to (4).

$$ind1(i,k) = \mu\cdot\sum_{j=1}^{m}\left(c_{j,k}(\sigma^*i) - c_{j,k-1}(\sigma) - p_{j,i}\right) + (1-\mu)\cdot\left(c_{m,k}(\sigma^*i) - c_{m,k-1}(\sigma)\right) \tag{4}$$

The job selected is the one that has the minimum value of index *ind1* (i, k).

- **RCP0** builds the solution by choosing a job and a line at the same time. At each iteration, the line that has the last machine available soonest is selected. Next, the job that leads to the minimum timeout, which is calculated by equation (3), is chosen. This process is repeated until all jobs have been assigned.

4.2. Local searches

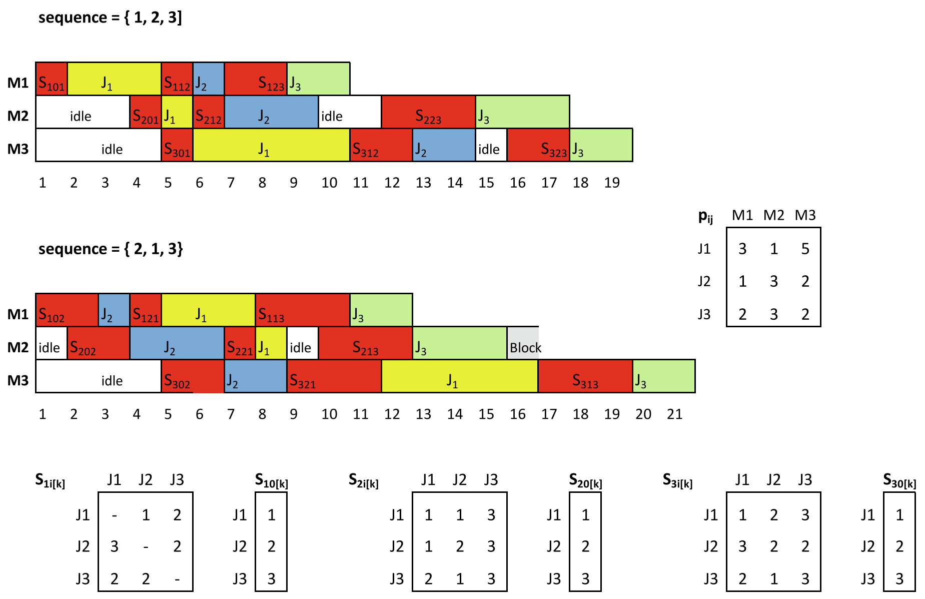
Since this research aims to define an efficient algorithm for the problem, two type of local searches, with different neighborhood structures, were used to analyze their performance.

The first local search, *SSA*, is designed to improve the sequence of each line in order to diminish its C_{max}, whereas the second is designed to reduce the maximum makespan among lines by trying to move jobs from the critical line (f_{max}), the one with the maximum C_{max}, to other lines.

```
$f_{max} = 0$
for $f = 1$ to F
        $i = \sigma_{1,f}$
        $c_{1,1,f} = s_{j,0,k} + p_{1,i}$
        for $j = 2$ to $m$
                $c_{j,1,f} = \max\{s_{j,0,i}\ ;\ c_{j-1,1,f}\} + p_{j,i}$
        next $j$
        for $k = 2$ to $n_f$
                $i = \sigma_{k,f} : h = \sigma_{k-1,f}$
                for $j = 1$ to $m$
                        $c_{j,k,f} = \max\{\max\{c_{j,k-1,f} + s_{j,h,i}\ ;\ c_{j-1,k,f}\} + p_{j,i}\ ;\ c_{j+1,k-1,f}\}$
                next $j$
        next $k$
        if $f_{max} < c_{m,nf,f}$ then $f_{max} = c_{m,nf,f}$
        next f

Being:
        $c_{0,k,f} = c_{m+1,k,f} = 0$        $\forall\ k, f$
```

Fig. 1. Outline of makespan calculation.



p_{ij}	M1	M2	M3
J1	3	1	5
J2	1	3	2
J3	2	3	2

$S_{1i[k]}$	J1	J2	J3	$S_{10[k]}$
J1	-	1	2	1
J2	3	-	2	2
J3	2	2	-	3

$S_{2i[k]}$	J1	J2	J3	$S_{20[k]}$
J1	1	1	3	1
J2	1	2	3	2
J3	2	1	3	3

$S_{3i[k]}$	J1	J2	J3	$S_{30[k]}$
J1	1	2	3	1
J2	3	2	2	2
J3	2	1	3	3

Fig. 2. Makespan comparison between scheduling.

```
IG Algorithm
 Π:=initial solution
 C*=$C_{max}$(Π)
 Π:=SSA (Π)
 Π:=MIL (Π, nlimr, nlimp)
 C*=$C_{max}$(Π)
 Π*:=Π
 while stopping criterion is not met do
     Π':= Perturbation (Π, d)
     Π':=SSA (Π')
     Π':=MIL (Π', nlimr, nlimp)
     if $C_{max}$(Π')<C* then
         Π:=Π'
         Π*:=Π'
         C*:=$C_{max}$(Π')
     else
         Π:=Acceptance Function (Π',T)
     endif
 end while
return C*, Π
end
```

Fig. 3. Outline of the algorithm.

This second local search was called *MIL*. In both searches, the insertion and swap neighborhood were tested, as well as the combination of both.

4.2.1. Line improvement: SSA

In this improvement the local search is performed, in each line, by exploring the neighborhood of the current sequence. An analysis was made of the effectiveness of using the swap procedure, the insertion procedure or a combination of both.

The swap procedure is described as follows. For each job in the sequence, neighbors are generated by swapping a job with all the jobs that follow it in the sequence. If the best neighbor ($\sigma_f{}'$) is better than the current solution (σ_f), it becomes the new current solution σ_f, and the process continues until all jobs have been considered. To prevent the neighborhoods from always being explored in the same order, the jobs are selected randomly. Notice, in Fig. 4, that **swap** (σ, k_1, k_2) indicates that jobs placed in positions k_1 and k_2, in sequence σ, exchange their positions and **shuffle**(σ) creates a randomized sequence with the jobs in σ.

The insertion procedure functions as follows. For each job in the sequence, neighbors are generated by removing the job from its position and inserting it into all the other possible positions. If the best neighbor ($\sigma_f{}'$) is better than the current solution (σ_f), it becomes the new current solution σ_f, and the process continues until all jobs have been considered. As in the swap procedure, jobs are selected randomly. Notice that **reinsert**(σ, k_1, k_2) indicates that job in position k_1, in sequence σ, is placed in position k_2

In the case of using both methods, the neighborhood to be explored first is selected randomly (50% probability each). After exploring the neighboring solutions of current solution σ_f, the local optimum $\sigma_f{}'$ is compared with σ_f. If the solution has improved, $\sigma_f{}'$ replaces σ_f and the search continues in the other neighborhood. This process continues until the current solution is no longer improved.

4.2.2. Redistribution of jobs between lines: MIL

This phase tries to assign the jobs allocated to the critical line f_{max}, the one with a maximum C_{max}, to another line by either reassigning a job to another line or exchanging a job with another job of a different line. We named these procedures *Reassignment* and *Permutation,* respectively.

The *reassignment* procedure consists of swapping two jobs: one from f_{max} and one from another randomly selected line. If the maximum makespan among lines (C_{max}) diminishes, the change is kept and the procedure is repeated with the line that now has the maximum makespan. If the movement does not improve the C_{max}, a new job from f_{max} is selected and the process starts again. The search finishes either when all the jobs in the critical line have been selected or after a limited number of iterations (*nlimir*) are reached. Then the process iterates over each neighborhood to find improvements. The outline of this procedure is shown in Fig. 5.

The *Permutation* procedure involves selecting a job randomly from the line which has the maximum makespan (f_{max}) and then inserting it into the best position of another randomly selected line, i.e., the position that leads to a minimum makespan of this line. If the C_{max} diminishes, the sequence is kept and the procedure is repeated again with the line that now has the maximum makespan. In the same way as the

```
Procedure Swap($\sigma$)
improve =1; $c^*:=C_{max}(\sigma)$; $\sigma^*:=\sigma$; $\sigma_0:=\sigma$; $n_\sigma=|\sigma|$
 while improve=1
  $\sigma_0$:=shuffle($\sigma_0$); improve =0; k=0
  do
        k=k+1
        if k > $n_\sigma$ then exit do
         i=$\sigma_0$(k); $k_1$=position(i, $\sigma$)
         $k_{ins}$ = 0: $c_{aux}$ = infinite; $k_2$ = 0
         do
           $k_2 = k_2 + 1$
           if $k_2 > n_\sigma$ then exit do
              $\sigma'$:=swap($\sigma$,$k_1$,$k_2$)
              If $C_{max}(\sigma') < c_{aux}$ then
                $c_{aux} = C_{max}(\sigma')$; $k_{ins} = k_2$
              end if
          loop
        if $c_{aux} < c^*$ then
              improve = 1;
              $c^*=c_{aux}$;
              $\sigma$:=swap ($\sigma$,$k_1$,$k_{ins}$);
              $\sigma^*:=\sigma e$
        end if
   loop
  end while
 $\sigma:=\sigma^*$
 return $\sigma$
end
```

```
Procedure Insert($\sigma$)
improve =1; $c^*:=C_{max}(\sigma)$; $\sigma^*:=\sigma$; $\sigma_0:=\sigma$; $n_\sigma=|\sigma|$
while improve=1
$\sigma_0$:=shuffle($\sigma_0$); improve =0; k=0
do
        k=k+1
        if k > $n_\sigma$ then exit do
        i=$\sigma_0$(k); $k_1$=position(i, $\sigma$)
        $k_{ins}$ = 0: $c_{aux}$ = infinite; $k_2$ = 0
        do
        $k_2 = k_2 + 1$
        if $k_2 > n_\sigma$ then exit do
                if $k_2 \neq k_1$ then
                $\sigma'$:=reinsert($\sigma$,$k_1$,$k_2$)
                if $C_{max}(\sigma') < c_{aux}$ then
                $c_{aux}= C_{max}(\sigma')$ ;$k_{ins}=k_2$
                end if
        end if
        loop
        if $c_{aux} < c^*$ the
        improve = 1;
        $\sigma$:=reinsert($\sigma$,$k_1$,$k_{ins}$);
        $c^*=c_{aux}$; $\sigma^*:=\sigma_f$
        end if
loop
end while
$\sigma:=\sigma^*$
return $\sigma$
end
```

Fig. 4. Outline of the Swap and Insert procedures.

```
Procedure Reassignment ($\Pi$, nlimr)
  do
     for f=1 to F: q(f)=f: next f
          $c_{aux}:= C_{min}(\Pi)$
          Detect line f* such as $C_{min}(\Pi)=C_{min}(\sigma_{f^*})$
          $\sigma':=\sigma_{f^*}$; $\sigma'$:=shuffle($\sigma'$)
          $nlim_l$=MIN{nlimr, |$\sigma'$|};  $k_b$ = 0
      do
        improve =0; $k_b = k_b + 1$
       if $k_b > nlim_l$ then exit do
          $i_b = \sigma'(k_b)$; f'= 0; q:=suffle(q); $k_q$ = 0
         do
                $k_q=k_q+1$
           if $k_q$ > F then exit do
                f= q($k_q$)
               if f≠f* then
                 $\sigma_1$:= insert $i_b$ on the best position of $\sigma_f$
                        if $C_{max}(\sigma_1) < c_{aux}$ then
                         $c_{aux}= C_{max}(\sigma_1)$; improve =1 ; f'= f
                          exit do
                        end if
              end if
          if improve =1 then exit do
        loop
        if f' > 0 and $c_{aux} < C_{min}(\Pi)$  then
          $\Pi$:=on $\Pi$ remove $i_b$ from $\sigma_{f^*}$ and insert $i_b$ on the best position of $\sigma_{f'}$
         exit do
        end if
      loop
    if improve =0 then exit do
  loop
 return $\Pi$, $C_{max}(\Pi)$
end
```

Fig. 5. Outline of the Reassignment procedure.

```
Procedure Permutation ($\Pi$, nlimp)
  do
     for f=1 to F: q(f)=f: next f
      caux:= Cmin($\Pi$)
      Detect line f* such as Cmin($\Pi$)=Cmin($\sigma$f*)
      $\sigma'$:=$\sigma$f*; $\sigma'$:=shuffle($\sigma'$)
      nlim1=MIN{nlimp, |$\sigma'$|} ;  kb1 = 0
     do
        improve=0; kb1 = kb1 + 1
        if kb1 > nlim1 then exit do
         ip1=$\sigma'$(kb1)
         q:=suffle(q);kq = 0
         do
            kq=kq+1
            if kq > F then exit do
             f= q(kq)
             if f≠f* then
               $\sigma''$=$\sigma$f; $\sigma''$=shuffle($\sigma''$); nlim2=MIN{nlimp, |$\sigma''$|} ;  kb2 = 0
               do
                  kb2 = kb2 + 1
                  if kb2 > nlim2 then exit do
                    ip2 = $\sigma''$(kp2)
                     Remove ip1 from $\sigma$f* and insert in the best position ip2 obtaining $\sigma$2
                     Remove ip2 from $\sigma$f and insert in the best position ip1 obtaining $\sigma$1
                    if Cmax($\sigma$1)<caux and Cmax($\sigma$2)<caux then
                      improve =1
                      exit do
                    end if
                loop
              if improve=1 then exit do
            end if
        loop
       if improve=1 then exit do
     loop
     if improve=1 then
          $\Pi$:=$\sigma$2 replaces $\sigma$f* and $\sigma$2 replaces $\sigma$f on $\Pi$
    end if
   if improve=0 then exit do
  loop
 return $\Pi$, Cmax($\Pi$)
end
```

Fig. 6. Outline of Permutation procedure.

Reassignment procedure, if the movement does not improve the C_{max}, a new job is selected from $f_{\max}$ and the process starts again. This part finishes either when all the jobs in the critical line have been selected or after a limited number of iterations (*nlimip*) are reached. The outline of this procedure is shown in Fig. 6.

4.3. Perturbation mechanism

The perturbation mechanism prevents the solution from being trapped in a local optimum. The implemented method is similar to the one proposed by Fernandez-Viagas & Framinan (2014). It tries to diversify the search by removing d randomly selected jobs from its current line, attempting to assign each job to all the other lines in order to find the factory and position that leads to the minimum global makespan. The outline of this procedure can be seen in Fig. 7.

```
Procedure Perturbation ($\Pi$, $d$)
  Select from $\Pi$, $d$ jobs randomly without repetition and put them in sequence $\tau$
  Remove $d$ jobs from $\Pi$ giving $\Pi'=(\sigma_1', \sigma_2', \ldots, \sigma_F')$
    for k= 1 to $d$
        $i=\tau(k)$; $c_{aux}$=infinite; $f'=0$;
        for f=1 to F
            Insert i in the best position of $\sigma_f'$ giving $\sigma_0$
          if $C_{max}(\sigma_0)<c_{aux}$ then
            $c_{aux}=C_{max}(\sigma_0)$; $f'=f$; $\sigma_1=\sigma_0$
          end if
        next f
        $\sigma_{f'}=\sigma_1$
    next k
  return $\Pi$, $C_{max}(\Pi)$
end
```

Fig. 7. Perturbation mechanism of IGA.

4.4. Acceptance Function

The *Acceptance Function* allows the search to be diversified, as does the perturbation mechanism. The scheme used, see equation (5), is similar to that used in Fernandez-Viagas & Framinan (2014), inspired by the mechanism used in the simulated annealing algorithm. Parameter *T* allows calibrating the interval of the worst solution accepted.

$$Temperature = T\cdot\frac{\sum_{i=1}^{n}\sum_{j=1}^{m}p_{i,j}}{n\cdot m\cdot 10} \tag{5}$$

4.5. Experimental parameter adjustment of heuristic

The proposed algorithm has four parameters to be calibrated: the number of jobs considered in the perturbation mechanism (*d*); the maximum number of iterations conceded to the reassignment process (*nlimr*) and perturbation process (*nlimp*); and the temperature (*T*) used to calibrate the acceptance mechanism. Each parameter was tested with three levels: d={3, 4, 5}, *nlimr*={10, 15, 20}, *nlimp*={10, 15, 20} and T={0.3, 0.4, 0.5}. This results in 3x3x3 = 27 algorithm configurations.

The instances used in this test were combinations of n={25, 50, 75, 100 } and m={5, 10, 15, 20}. There were 16 sets with 5 instances in each that were solved for F={2, 3, 4 ,5}. Therefore, a total of 320 instances were used in this test. The setup time was generated according to a uniform distribution between [1, 50] and the processing time of jobs according to distribution in a range interval [1, 99], as in the scheduling literature. The algorithms were stopped after $20\cdot n^2\cdot m\cdot 10^{-5}$ s. Each one of the 320 calibration instances was run for five different replicates in each algorithm configuration.

The experiments' response variable is the Relative Percentage Deviation (RPD), which calculates the difference between the solution obtained by a specific algorithm configuration and the best solution found, over the same instance, by any of the possible combination of values of parameter. The index RPD is calculated as in (6):

$$RPD = \frac{Sol_{i,j} - Best_i}{Best_i}\bullet 100 \tag{6}$$

where $Sol_{i,j}$ is the makespan value obtained by algorithm *j* (i.e. the algorithm with a determined value of parameters) in calibration instance *i*, and $Best_i$ is the minimum makespan obtained during the evaluation over the same instance.

The algorithm was coded in QB64 and tested on a computer 3.40 GHz Intel Core i7-3770 CPU with 8 GB of RAM. Both instances and solutions are available from the authors on request.

The initial solution used was TRA5, which according to (Ribas & Companys, 2021) showed the best performance in terms of solution quality.

The results were analyzed by an ANOVA and the residual analysis did not show any important violation of the model hypothesis.

As can be seen in Table 1, only parameters *d* and *nlimp* are significant. Note the interaction between *d* and *n*, *m*, *F*, *nlimp*; the interaction between *nlimp* and *n*, *F* is also significant.

Table 1
Anova: ARPD versus *n*, *m*, *F*, *d*, *T*, *nlimr* and *nlimp*.

Source	DF	SS	MS	F	P
n	3	800.36	266.788	4044.13	0.000
m	3	1212.68	404.226	6127.50	0.000
F	3	183.16	61.054	925.49	0.000
d	2	13.43	6.717	101.81	0.000
nlimr	2	0.06	0.032	0.49	0.611
nlimp	2	3.98	1.992	30.19	0.000
T	2	0.01	0.007	0.11	0.900
*n*m*	9	21.81	2.423	36.73	0.000
*n*F*	9	75.89	8.432	127.82	0.000
*n*d*	6	19.47	3.245	49.20	0.000
*n*nlimr*	6	0.20	0.034	0.52	0.796
*n*nlimp*	6	3.82	0.637	9.65	0.000
*n*T*	6	0.19	0.032	0.48	0.825
*m*F*	9	45.14	5.016	76.04	0.000
*m*d*	6	1.80	0.300	4.54	0.000
*m*nlimr*	6	0.12	0.020	0.30	0.937
*m*nlimp*	6	0.66	0.110	1.67	0.124
*m*T*	6	0.11	0.019	0.29	0.943
*F*d*	6	0.85	0.141	2.14	0.045
*F*nlimr*	6	0.40	0.067	1.01	0.417
*F*nlimp*	6	12.92	2.154	32.64	0.000
*d*nlimr*	4	0.01	0.003	0.05	0.996
*d*nlimp*	4	3.24	0.810	12.28	0.000
*d*T*	4	0.07	0.018	0.28	0.892
*nlimr*nlimp*	4	0.23	0.059	0.89	0.469
*nlimr*T*	4	0.14	0.034	0.52	0.721
*nlimp*T*	4	0.08	0.020	0.31	0.874
Error	25,785	1701.02	0.066		
Total	25,919	4101.89			

Fig. 8 shows that the best values are obtained when $d = 4$ and *nlimp* = 15, whereas Figs. 9 and 10 show the two most significant interactions: *n*d* and *F*nlimp*.

Notice that the interaction between *n* and *d* is due to $n = 25$, which has a different behavior to the others. In this case there is no difference between $d = 4$ and $d = 5$, whereas for the other values of *n* it is clear that the best value is $d = 4$. The same happens with the interaction between *F* and *nlimp*, which is due to $F = 2$, since the *nlimp* = 15 is the best value for all values of *F*, except when the number of lines is 2. A slightly better behavior is observed if *nlimp* is 10, although it can be seen that the difference between the Mean RPD values reached when selecting 10 or 15 is very small. Hence, our algorithm is set with $d = 4$, *nlimr* = 10, *nlimp* = 15 and $T = 0.4$.

5. Computational evaluation

In this section, we first evaluate the contribution of each local search and neighborhood structure to the performance of the IG algorithm. Then, we test the algorithm with different initial solution procedures, as well as, against an adaptation of the Iterated Greedy algorithm with a Restart scheme (IGR) proposed in (Huang et al., 2020) for the Distributed Flow shop problem with sequence-dependent setup times. These tests were conducted on a set of *ad-hoc* generated instances. Notice that this set of instances is different than the one used for calibrating the parameters in order to avoid any overfitting.

5.1. Performance evaluation of local searches

This test is performed to define the most effective structure for the local searches. Specifically, we aim to analyze whether it is better to explore one of the defined neighborhoods or both, either in the improvement of the sequence in each line (SSA) or in the improvement of product allocation to lines (MIL). We run an almost full factorial design of experiments with two factors: SSA and MIL with three levels each (the only combination we did not run was not doing any local search). In the case of SSA the levels are: *Swap, Insert, Both*. For MIL the

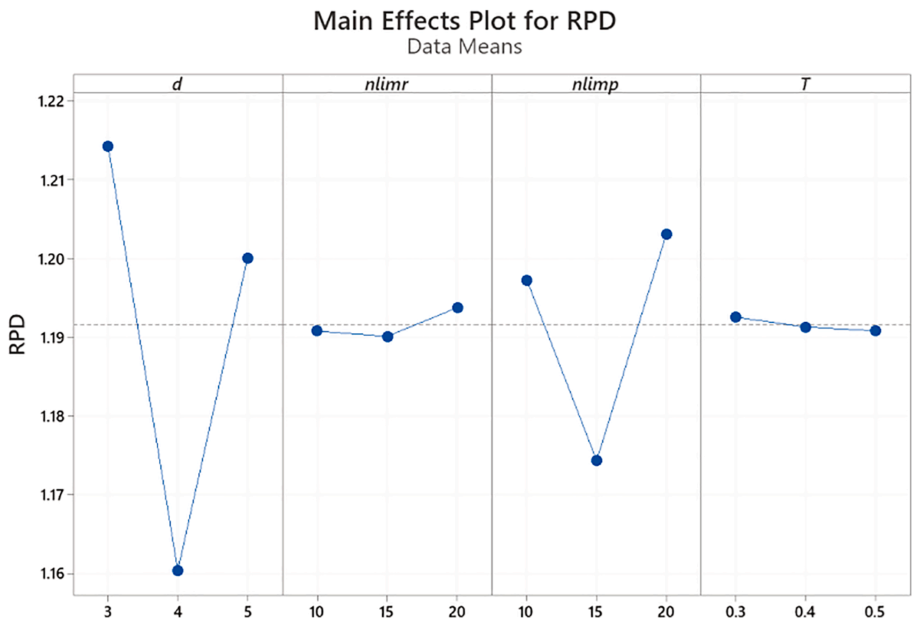


Fig. 8. Main effect plot for *d*, *nlimr*, *nlimp* and *T*.

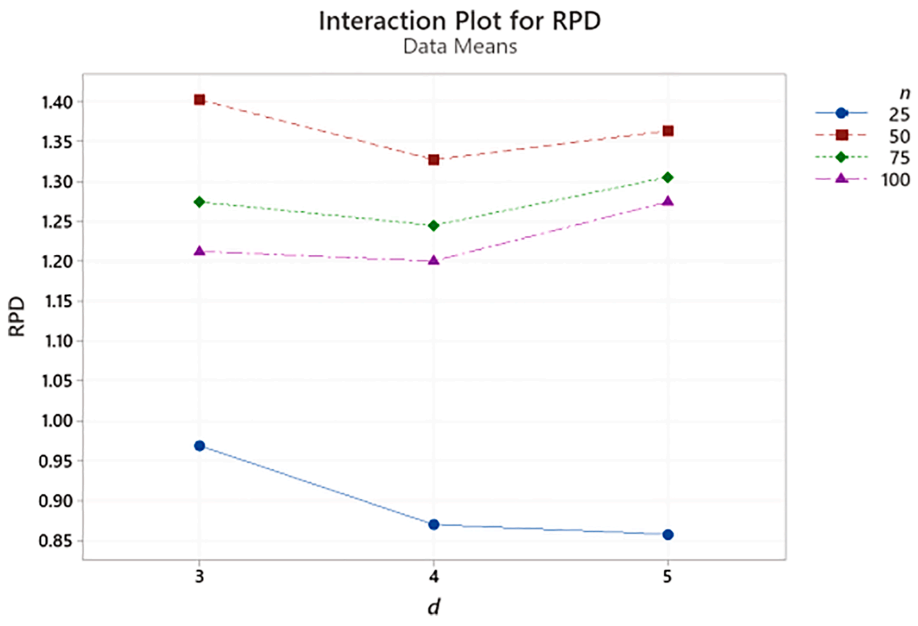


Fig. 9. Interaction plot for *n* and *d*.

levels are: *Reassignment, Permutation, Both*.

For this test we used a set of instances that combined values of $n \times m$, with n={25, 50, 75, 100, 150, 200} and m={5, 10, 15, 20}. There were 24 groups of instances and 10 instances per group, for a total of 240 instances solved for F={2, 3, 4, 5} and three levels of setup times. Hence, the test was conducted on 2880 instances. The processing time was generated as before, using a uniform distribution in the range [1,99]. The three levels of setup were also generated according to a uniform distribution with the range intervals [1, 20] for the low level, [1, 50] for the medium level and [1, 120] for the high level. The algorithms were stopped after $30 \cdot n^2 \cdot m \cdot 10^{-5}$ s.

Algorithms were run 5 times per instance. The response variable was the RPD calculated as equation (6) where $Sol_{i,j}$ is the makespan obtained by heuristic j (i.e. specific combination of SSA and MIL levels) in instance i, and $Best_i$ the minimum makespan obtained by any of the configurations tested over the same instance.

The results were analyzed by an ANOVA. The model hypotheses were tested by a residual analysis that did not show any violation. From the result shown in Table 2 it can be seen that the main factors and their interaction are significant (*p-value* = 0). Moreover, column F shows that MIL has a greater influence than SSA in improving the solution, as was expected. SSA only improves the sequence of each line, whereas MIL moves jobs from a critical line to another one to reduce the total C_{max}. Notice that SSA and MIL interaction is significant but really small (F value). In fact, it confirms that the best combination is when the two neighborhoods, in the two types of local searches, Insert-Swap and Reassignment-Permutation, are used and shows that this solution is a bit better than the addition of the two individual effects.

To better visualize the contribution to each type of local search structure in SSA and MIL, we separate the analysis in two graphs. Fig. 11

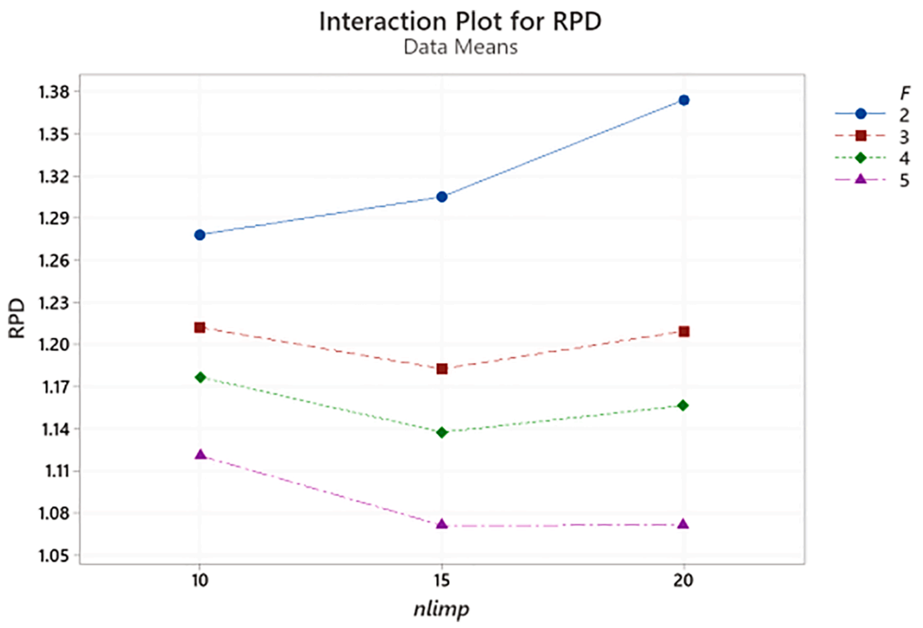


Fig. 10. Interaction plot for *F* and *nlimp*.

Table 2
Anova: ARPD versus *n*, *m*, *setup*, *F*, *SSA* and *MIL*.

Source	DF	SS	MS	F	P
n	5	601.1	120.21	1040.48	0.000
m	3	1681.3	560.44	4850.82	0.000
setup	2	923.9	461.97	3998.49	0.000
F	3	104.6	34.87	301.80	0.000
SSA	2	236.6	118.32	1024.08	0.000
MIL	2	9140.1	4570.06	39555.64	0.000
*n*m*	15	34.8	2.32	20.09	0.000
*n*setup*	10	75.6	7.56	65.43	0.000
*n*F*	15	178.7	11.92	103.14	0.000
*n*SSA*	10	127.6	12.76	110.41	0.000
*n*MIL*	10	271.8	27.18	235.28	0.000
*m*setup*	6	167.2	27.87	241.25	0.000
*m*F*	9	18.1	2.01	17.39	0.000
*m*SSA*	6	29.2	4.86	42.07	0.000
*m*MIL*	6	402.4	67.07	580.53	0.000
*setup*F*	6	10.7	1.79	15.47	0.000
*setup*SSA*	4	106.9	26.74	231.42	0.000
*setup*MIL*	4	81.5	20.38	176.42	0.000
*F*SSA*	6	21.5	3.58	30.98	0.000
*F*MIL*	6	1153.5	192.26	1664.04	0.000
*SSA*MIL*	4	33.1	8.27	71.57	0.000
Error	25,785	2979.1	0.12		
Total	25,919	18379.5			

shows the interval plot of RPD considering the effect of *setup times* and *swap* and *insert* procedures, which are the two tested in SSA. Note that, in Fig. 9, in the *Insert* level, if the insert neighborhood is included in the local search it is denoted by 1, if not it is denoted by −1 and, at the SWAP level, it is denoted as Yes - No if the swap neighborhood is included or not. Hence, in the level SWAP = yes we can find two values of Insert: −1 means that only the swap neighborhood is considered and 1 means that both neighborhoods (*Swap* and *Insert*) are considered. In this figure, it can be observed that the insert procedure is more effective than the swap one, but using both produces better results. It is worth noting that for larger setup times the performance of searching in the swap neighborhood decreases, as can be seen by comparing the interval RPD between level 1 and level 2 or levels 2 and 3 of setup times.

A similar analysis is made for the local Search MIL. In this case the reassignment neighborhood (Reassigment in Fig. 12) is the one with the poorest performance, but, as can be seen in this figure, the combination of both neighborhoods leads to better performance. This confirms our choice for setting *nlimp* with a higher number of iterations than *nlimr*, in order to give more time to search in the best-performing neighborhood. Here again, we can see that the level of setup time is significant. The longer the setup time, the poorer the performance of the reassignment procedure.

In Fig. 12 it is hard to see if the differences between the variants are significant, due to poor performance from using only the reassignment procedure that forces a large scale in the Mean RPD axis. Therefore, it has been excluded from the analysis. In Fig. 13 it is evident that the Reassignment and Permutation procedure is highly recommended for low levels of setup times, whereas when the setup times increase, the difference between using only the permutation procedure or both is reduced.

Finally, Fig. 14 shows the significant but, almost irrelevant, interaction between SSA and MIL. It is clear that the two local searches used, inside and between lines, perform better when the two neighborhood (Insert-Swap and Reassignment-Permutation) are used. It is also clear that Insert and Permutation are more efficient than Swap and Reassignment.

5.2. *Computational evaluation of the IG*

In this second test, we start comparing the 5 simple constructive heuristic proposed for the initial solution procedure: LPT5, TRA5, RCP0, PF23 and HPF3. These are very fast methods that require little CPU time. The test instances used were the same that in the previous test and, as mentioned before, they are different from the calibration instances.

The 5 constructive procedures were coded in QB64 and tested on a computer 3,40 GHz Intel Core i7-3770 CPU with 8 GB of RAM. The response variable was the RPD calculated as in equation (6) where, in this case, $Sol_{i,j}$ is the makespan obtained by constructive heuristic j in instance i, and $Best_i$ is the minimum makespan obtained by any of these constructive methods over the same instance.

Table 3 shows the average RPD (ARPD) obtained by number of lines and level of setup time. The fourth column is the overall average by line and the fifth the overall average by method. From these results one can see that the best performing method is TRA5, as it was already showed in (Ribas & Companys, 2021). PF23 and HPF3 shows a poor performance compared to the others and RCP0, specially designed for this problem, is

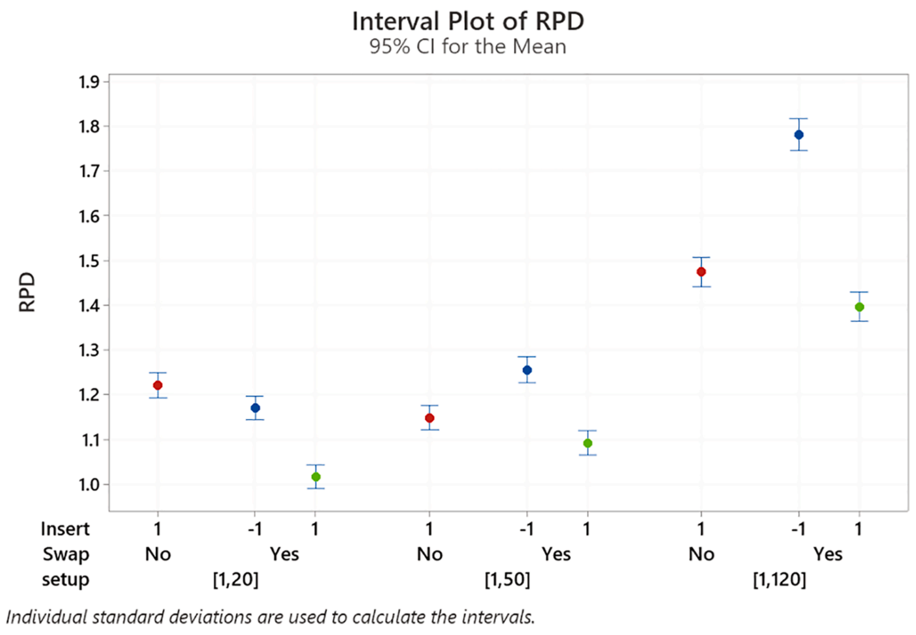


Fig. 11. interval plot of RPD by setup times, swap and insert levels.

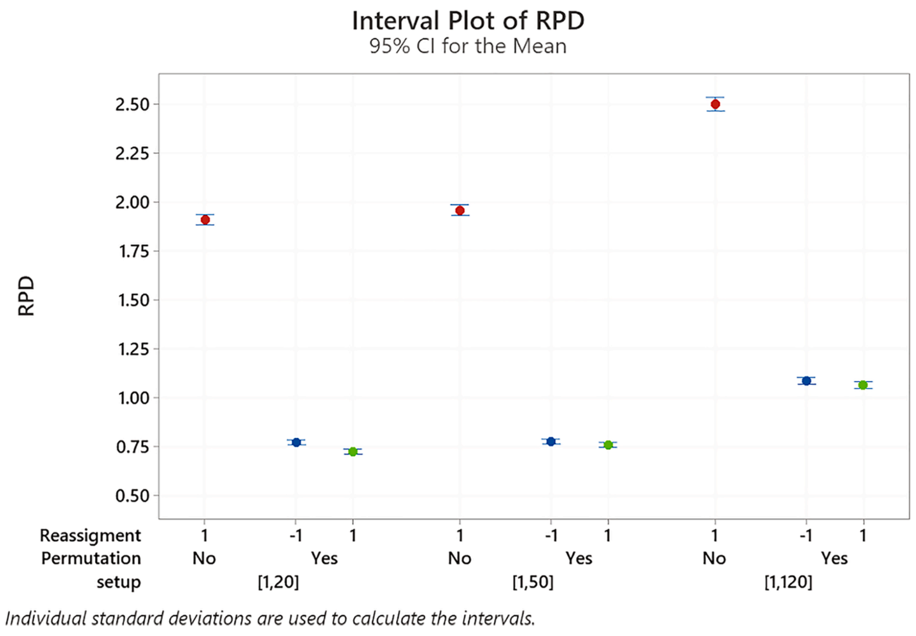


Fig. 12. Interval plot of RPD by *Setup times*, *Reassignment* and *Permutation* levels.

in the third position. It is worth noting that the ARPD of TRA5, LPT5, PF23 and HPF3 increases when setup time increases whereas RCP0 shows the contrary behavior.

Although all these methods take little CPU time, it is interesting to observe the time difference between them. Fig. 15 shows the scatterplot between the average RPD (ARPD) and the average CPU (ACPU) time used by each procedure. In it can be observed, that the best performing methods (TRA5 and LPT5) require about 4 times more than the other three methods. This is so because both methods include an improvement phase embedded in the process. Notice that for each job, once the line is selected, it is tested in all positions in the sequence and placed in the position which results in its minimum C_{max}. This fact leads them to obtain better solutions but with higher CPU time.

RCP0, which was especially designed for the problem dealt with, obtains solutions of certain quality with a little CPU time. Finally, PF23 and HPF3, although as constructive methods do not reach good solutions, they can be also good candidates to help IG to reach better solutions because are very fast methods and both create a sequence considering the minimization of timeout of machines. Therefore, the next step is to analyze the effect of the initial solution procedure in the performance of the IG algorithm and to establish the best method to be implemented in the proposed IG algorithm.

The set of instances used in this test are the same as in the second test. The response variable was the RPD calculated as in equation (6) where, $Sol_{i,j}$ is the makespan obtained by variant j of IG algorithm in instance i, and $Best_i$ is the minimum makespan obtained during this research over the same instance.

The results were analyzed by means of an ANOVA (Table 4) where *Algorithms* are the variants of IG with the different initial solutions. Here, one can see that all the factors and their interactions are significant.

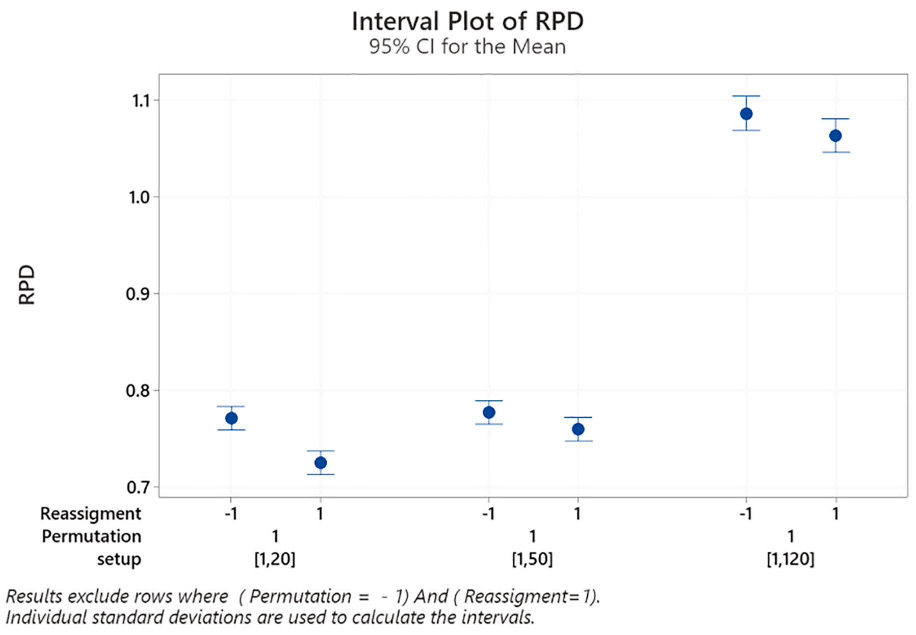


Fig. 13. interval plot of RPD considering only Permutation or combining Reassignment and Permutation.

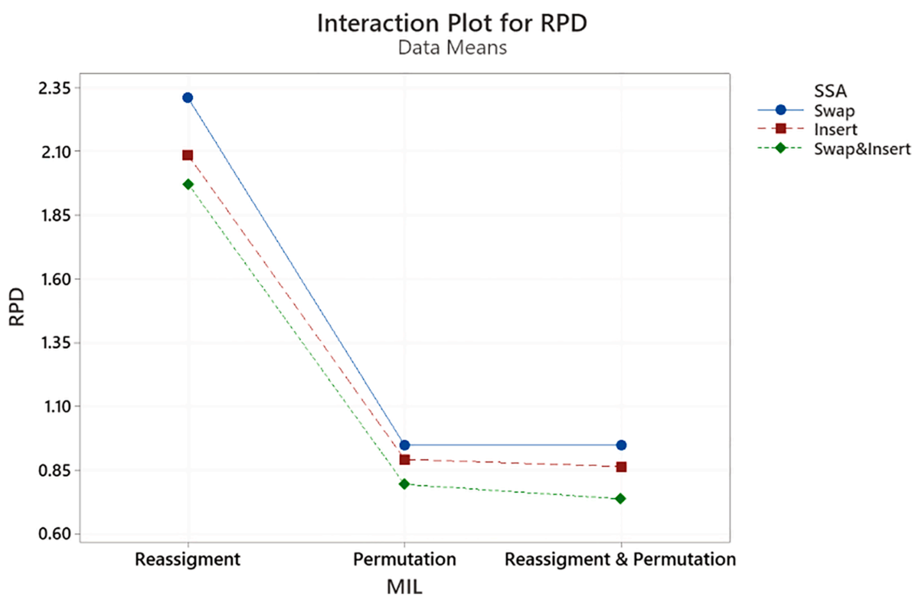


Fig. 14. Interaction plot between SSA and MIL local searches.

The scatterplot (Fig. 16) compares the mean RPD values between constructive methods and the IG algorithm when the same constructive method is used to obtain the initial solution (denoted as initial_solution_IG). It is worth mentioning that the same best value for the RPD calculation was used in both cases, which allowed us to see the effectiveness of the IG algorithm. Note, in Fig. 16, that the improvement heuristic is very effective since the mean RPD shows considerable improvement. For instance, the RPD mean for PF23 is about 26 whereas IG with PF23 (IG_PF23) to obtain the initial solution, is about 0.85.

Another interesting outcome is that the best results of the IG algorithm are obtained when the initial solution is generated by the RCP0 procedure, followed by PF23 and HPF3, whereas these three methods, as constructive heuristics, perform worse than the TRA5 and LPT5.

As mentioned before, PF23, HPF3 are only sequencing rules, although they consider the characteristics of the problem, since the sequence created in each line minimize the timeout of machines due to blockage or idle time. Probably, this fact explain the improvement reached once the improvement phase of the IG acts. On the other hand, RCP0, which was specially designed for this problem, in addition to minimize the timeout of machines, selects at each iteration the job and the line at the same time in order to minimize the maximum makespan among lines. This is what allows RCP0 to reach better solutions than the other two methods not only as a constructive heuristic but also as an initial solution method for the IG.

To better understand the behavior of these algorithms, we analyzed their performance for each level of setup times. Fig. 17 show the Interval Plot by variant of IG and level of setup time. Observe that RCP0_IG and PF23_IG are the best algorithm when the setup times are low; however, when the setup time increases the difference between both methods increases, being the RCP0_IG the best performing variant. Note that all variants show worse performance when the level of setup increases, especially TRA5_IG and LPT5_IG. It is worth notice that RCP0_IG is quite

Table 3
Average RPD by constructive procedure.

	Setup	[1,20]	[1,50]	[1,120]		
	F	ARPD	ARPD	ARPD	All	Overall ARPD
TRA5	2	1.236	1.484	3.615	2.112	
	3	0.977	1.245	3.571	1.931	
	4	0.752	1.161	3.528	1.814	
	5	0.597	0.998	3.086	1.560	1.854
LPT5	2	1.431	1.605	3.772	2.269	
	3	1.213	1.576	3.693	2.161	
	4	1.219	1.404	3.766	2.130	
	5	1.320	1.398	3.534	2.084	2.161
RCP0	2	4.250	3.159	1.080	2.830	
	3	5.597	4.059	1.338	3.665	
	4	6.754	5.036	1.758	4.516	
	5	7.844	5.721	2.468	5.344	4.089
HPF3	2	8.162	8.699	10.910	9.257	
	3	10.528	10.258	11.752	10.846	
	4	12.299	11.771	12.527	12.199	
	5	13.368	12.663	13.110	13.047	11.337
PF23	2	9.055	9.419	11.648	10.041	
	3	11.209	11.295	13.045	11.850	
	4	12.874	12.560	13.806	13.080	
	5	14.128	13.492	14.268	13.963	12.233

robust, its average RPD is quite similar in the three levels of setup times indicating that this variant obtains most of the best solutions. Hence, the combination of RCP0 with IG is recommended to effectively solve the PBFSP with sequence-dependent setup times.

Finally, we compare the IG variants against the Iterated Greedy algorithm with Restart scheme (IGR) proposed by (Huang et al., 2020) for the DPFSP with sequence-dependent setup times, adapted to the problem on hand. Although the main structure of IGR is similar to the one proposed in this manuscript there are three remarkable differences:

1. IGR uses an extension of the NEH heuristic (Nawaz, Enscore Jr, & Ham, 1983) as initial solution procedure. This method orders the jobs according to the LPT rule, then the jobs are taken, one by one, and inserted into the best position of all lines. Hence, this method is more time consuming than LPT5 which tries to insert the jobs in one of the lines.
2. The local search only uses an insertion type of scheme in which a block of b jobs, being $b = rand[1, 3]$, is removed from the critical line and it is tried to be reinserted in all the positions among all lines in order to reduce the C_{max}.
3. Instead of using the simulated annealing-like acceptance criterion, a restart scheme is used to diversify the search.

This algorithm was coded with the same language than the other QB64 and tested in the same computer. The algorithms were stopped after $30 \cdot n^2 \cdot m \cdot 10^{-5}$ s and they were run 5 times per instance. The response variable was the RPD calculated as equation (6) where $Sol_{i,j}$ is the makespan obtained by algorithm j in instance i, and $Best_i$ the minimum makespan obtained during the research over the same instance.

Fig. 18 shows the interval plot of the implemented variants and IGR. It can be seen that either of the implemented variant are better than IGR which allow us to say that our proposal is very competitive. The poor behavior of IGR for the problem on hand can be due to the initial solution procedure but also to the local search used. In fact, the initial solution is similar to LPT5, who showed worse performance than the other variants. Also, we have seen that it is recommended using both the Permutation and Reassignment local searches, and in case of using only one the Permutation local search performs better than the Reassignment, which is the type of local search used in IGR.

Table 4
Analysis of Variance for RPD.

Source	DF	SS	MS	F	P
n	5	171.99	34.3986	363.73	0
m	3	267.57	89.1904	943.09	0
setup	2	105.63	52.8158	558.47	0
Algorithm	4	201.27	50.3174	532.05	0
F	3	45.12	15.0404	159.04	0
*n*m*	15	146.22	9.7479	103.07	0
*n*setup*	10	35.66	3.5664	37.71	0
*n*Algorithm*	20	281.33	14.0665	148.74	0
*n*F*	15	12.83	0.8553	9.04	0
*m*Algorithm*	12	178.73	14.8943	157.49	0
m*F	9	4.04	0.4494	4.75	0
setup*Algorithm	8	31.81	3.976	42.04	0
*setup*F*	6	1.69	0.2813	2.97	0.007
*Algorithm*Lines*	12	40.49	3.3743	35.68	0
Error	14,275	1350.03	0.0946		
Total	14,399	2874.42			

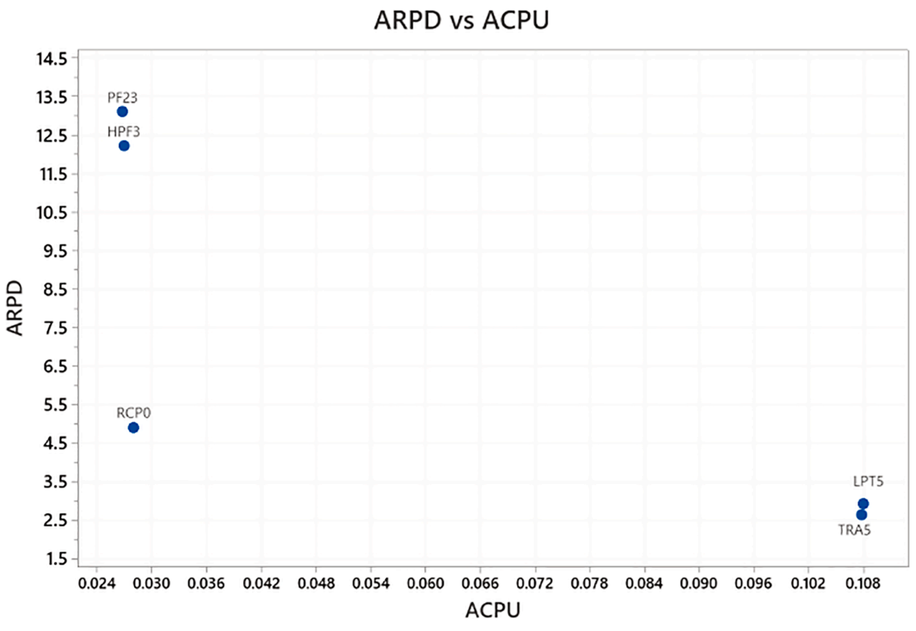


Fig. 15. Average RPD versus Average CPU, in seconds, per constructive heuristic.

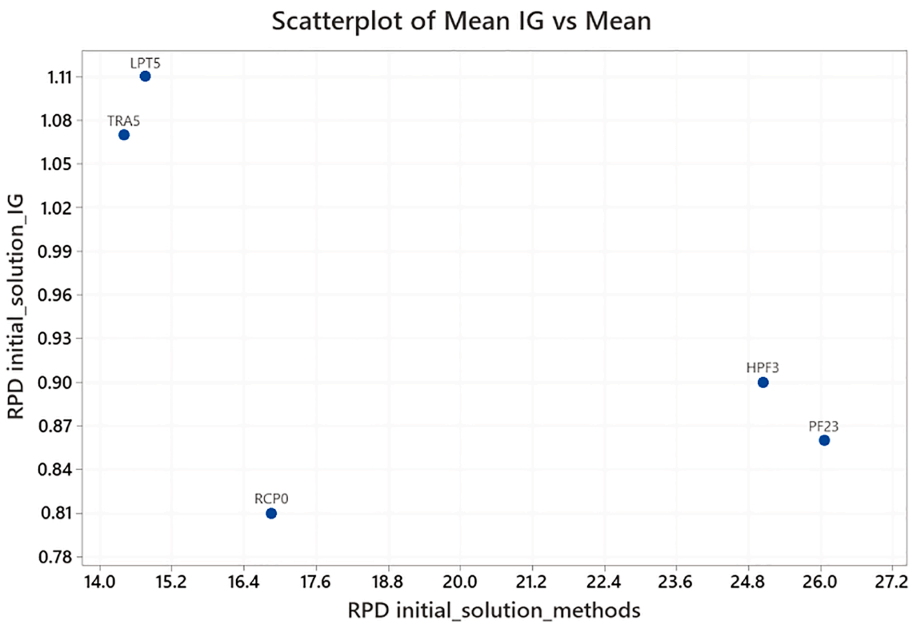


Fig. 16. Interval Plot by initial solution procedure.

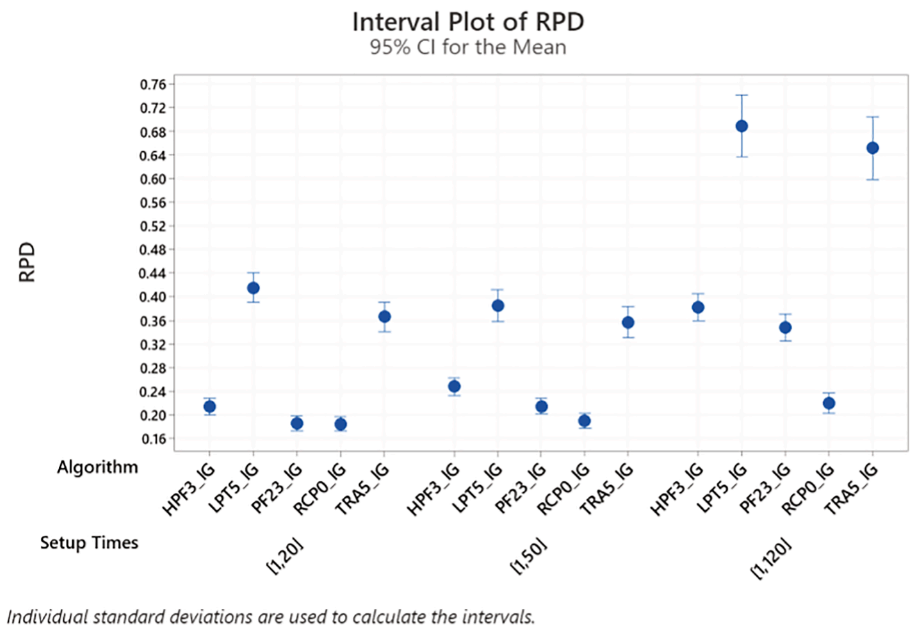


Fig. 17. Interval plot by IG variant and level of setup time.

6. Conclusions

In this paper, different variants of an iterated greedy algorithm were tested in order to provide an efficient algorithm to solve the parallel blocking flow shop problem with sequence-dependent setup times. The variants analyze the suitability of using simple or variable local searches, either to improve the sequence of each line or the distribution of jobs between lines, as well as the effect of the initial solution on the algorithm's performance.

From the computational evaluation of the different variants, it was concluded that the use of variable local searches is recommended. In particular, using Swap and Insert operators to improve the line's sequence and using Reassignment and Permutation procedures to improve the distribution of jobs among lines.

We have also shown that the initial solution has a significant influence on the final performance of the algorithm. It is worth noting that a good initial solution cannot guarantee the best final solution. In consequence, it has been necessary to analyze the combination of constructive procedures with the improvement phase to propose an efficient algorithm for this problem. In this case, the heuristic RCP0 is recommended, which specifically considers the characteristics of the problem (minimization of the idle and blocking taking into account setup times) to obtain the initial solution.

To the best of our knowledge this is the first attempt to study this problem. Therefore, we are sure that further research will propose improved methods to solve this scheduling problem efficiently. Additionally, other lines of research could be devoted to considering additional constraints or other criteria, such us total flow time or tardiness, which can be more suitable in some industrial environments.

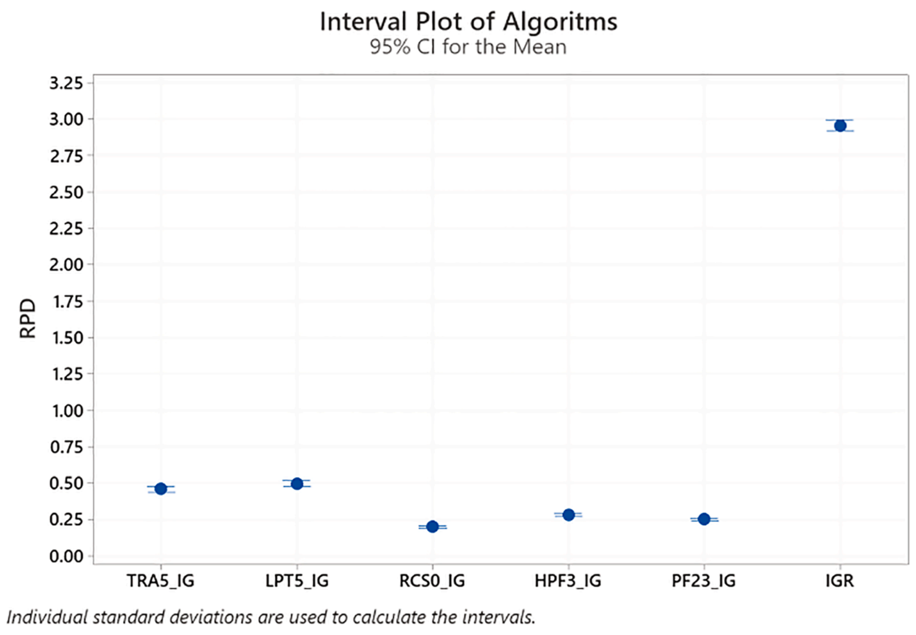


Fig. 18. Interval Plot of IG variants and IGR.

CRediT authorship contribution statement

Imma Ribas: Conceptualization, Methodology, Investigation, Formal analysis, Writing - original draft. **Ramon Companys:** Conceptualization, Investigation, Software, Data curation. **Xavier Tort-Martorell:** Formal analysis.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

Al-Salem, A. (2004). A Heuristic to Minimize Makespan in Proportional Parallel Flow Shops. *International Journal of Computing & Information Sciences, 2*(2), 98.

Allahverdi, A. (2015). The third comprehensive survey on scheduling problems with setup times/costs. *European Journal of Operational Research, 246*(2), 345–378. https://doi.org/10.1016/j.ejor.2015.04.004

Aqil, S., & Allali, K. (2021). On a bi-criteria flow shop scheduling problem under constraints of blocking and sequence dependent setup time. *Annals of Operations Research, 296*(1–2), 615–637. https://doi.org/10.1007/s10479-019-03490-x

Cao, D., & Chen, M. (2003). Parallel flowshop scheduling using Tabu search. *International Journal of Production Research, 41*(13), 3059–3073. https://doi.org/10.1080/0020754031000106443

Costa, A., Cappadonna, F. V., & Fichera, S. (2020). Minimizing makespan in a Flow Shop Sequence Dependent Group Scheduling problem with blocking constraint. Engineering Applications of Artificial Intelligence, 89(December 2019), 103413. https://doi.org/10.1016/j.engappai.2019.103413.

Dong, J., Jin, R., Luo, T., & Tong, W. (2020). A polynomial-time approximation scheme for an arbitrary number of parallel two-stage flow-shops. *European Journal of Operational Research, 281*(1), 16–24. https://doi.org/10.1016/j.ejor.2019.08.019

Fernandez-Viagas, V., & Framinan, J. M. (2014). A bounded-search iterated greedy algorithm for the distributed permutation flowshop scheduling problem. *International Journal of Production Research, 53*(4), 1111–1123. https://doi.org/10.1080/00207543.2014.948578

Fernandez-Viagas, V., Perez-Gonzalez, P., & Framinan, J. M. (2018). The distributed permutation flow shop to minimise the total flowtime. *Computers and Industrial Engineering, 118*, 464–477. https://doi.org/10.1016/j.cie.2018.03.014

Gao, J., & Chen, R. (2012). A hybrid genetic algorithm for the distributed permutation flowshop scheduling problem. Retrieved from http://www.tandfonline.com/doi/abs/10.1080/18756891.2011.9727808#abstract.

Gao, J., Chen, R., & Deng, W. (2013). An efficient tabu search algorithm for the distributed permutation flowshop scheduling problem. *International Journal of Production Research, 51*(3), 641–651. https://doi.org/10.1080/00207543.2011.644819

-, J. G., -, R. C., & -, Y. L. (2012). A Knowledge-based Genetic Algorithm for Permutation Flowshop Scheduling Problems with Multiple Factories. *International Journal of Advancements in Computing Technology, 4*(7), 121–129. https://doi.org/10.4156/ijact10.4156/ijact.vol4.issue710.4156/ijact.vol4.issue7.13

Gong, H., Tang, L., & Duin, C. W. (2010). A two-stage flow shop scheduling problem on a batching machine and a discrete machine with blocking and shared setup times. *Computers and Operations Research, 37*(5), 960–969. https://doi.org/10.1016/j.cor.2009.08.001

Han, Y., Li, J., Sang, H., Liu, Y., Gao, K., & Pan, Q. (2020). Discrete evolutionary multi-objective optimization for energy-efficient blocking flow shop scheduling with setup time. *Applied Soft Computing Journal, 93*, 106343. https://doi.org/10.1016/j.asoc.2020.106343

He, D. W., Kusiak, A., & Artiba, A. (1996). A scheduling problem in glass manufacturing. *IIE Transactions, 28*(2), 129–139. https://doi.org/10.1080/07408179608966258

Huang, J. P., Pan, Q. K., & Gao, L. (2020). An effective iterated greedy method for the distributed permutation flowshop scheduling problem with sequence-dependent setup times. *Swarm and Evolutionary Computation, 59*(June), Article 100742. https://doi.org/10.1016/j.swevo.2020.100742

Jiang, Y., & Wan, S. (2011). Parallel flow shop scheduling problem using quantum algorithm. In International Conference on Applied Informatics and Communication (Vol. 228 CCIS, pp. 269–274). Berlin: Springer Berlin-Heidelberg. https://doi.org/10.1007/978-3-642-23223-7_34.

Kim, M. G., Yu, J. M., & Lee, D. H. (2015). Scheduling algorithms for remanufacturing systems with parallel flow-shop-type reprocessing lines. *International Journal of Production Research, 53*(6), 1819–1831. https://doi.org/10.1080/00207543.2014.962112

Lin, S.-W., Ying, K.-C., & Huang, C.-Y. (2013). Minimising makespan in distributed permutation flowshops using a modified iterated greedy algorithm. *International Journal of Production Research, 51*(16), 5029–5038. https://doi.org/10.1080/00207543.2013.790571

Liu, H., & Gao, L. (2010). A discrete electromagnetism-like mechanism algorithm for solving distributed permutation flowshop scheduling problem. Proceedings - 2010 International Conference on Manufacturing Automation, ICMA 2010, 156–163. https://doi.org/10.1109/ICMA.2010.17.

Martinez, S., Dauzère-Pérès, S., Guéret, C., Mati, Y., & Sauer, N. (2006). Complexity of flowshop scheduling problems with a new blocking constraint. European Journal of Operational Research, 169(3), 855–864. https://doi.org/DOI: 10.1016/j.ejor.2004.08.046.

McCormick, S. T., Pinedo, M. L., Shenker, S., & Wolf, B. (1989). Sequencing in an assembly line with blocking to minimize cycle time. *Operations Research, 37*(6), 925–935.

Miyata, H. H., & Nagano, M. S. (2019). The blocking flow shop scheduling problem: A comprehensive and conceptual review. *Expert Systems with Applications, 137*, 130–156. https://doi.org/10.1016/j.eswa.2019.06.069

Moccellin, J. V., Nagano, M. S., Pitombeira Neto, A. R., & de Athayde Prata, B. (2018). Heuristic algorithms for scheduling hybrid flow shops with machine blocking and setup times. *Journal of the Brazilian Society of Mechanical Sciences and Engineering, 40*(2). https://doi.org/10.1007/s40430-018-0980-4

Naderi, B., & Ruiz, R. (2010). The distributed permutation flowshop scheduling problem. *Computers & Operations Research, 37*(4), 754–768. https://doi.org/10.1016/j.cor.2009.06.019

Naderi, B., & Ruiz, R. (2014). A scatter search algorithm for the distributed permutation flowshop scheduling problem. *European Journal of Operational Research, 239*(2), 323–334. https://doi.org/10.1016/j.ejor.2014.05.024

Newton, M. A. H., Riahi, V., Su, K., & Sattar, A. (2019). Scheduling blocking flowshops with setup times via constraint guided and accelerated local search. *Computers and Operations Research, 109*, 64–76. https://doi.org/10.1016/j.cor.2019.04.024

Rashidi, E., Jahandar, M., & Zandieh, M. (2010). An improved hybrid multi-objective parallel genetic algorithm for hybrid flow shop scheduling with unrelated parallel machines. *International Journal of Advanced Manufacturing Technology, 49*(9–12), 1129–1139. https://doi.org/10.1007/s00170-009-2475-z

Ribas, I., & Companys, R. (2021). A computational evaluation of constructive heuristics for the parallel blocking flow shop problem with sequence-dependent setup times. *International Journal of Industrial Engineering Computations, 12*(3), 321–328. https://doi.org/10.5267/j.ijiec.2021.1.004

Ribas, I., Companys, R., & Tort-Martorell, X. (2017). Efficient heuristics for the parallel blocking flow shop scheduling problem. Expert Systems with Applications, 74, 41–54. Retrieved from https://www.sciencedirect.com/science/article/pii/S0957417417300064.

Ribas, I., Companys, R., & Tort-Martorell, X. (2019). An iterated greedy algorithm for solving the total tardiness parallel blocking flow shop scheduling problem. *Expert Systems with Applications, 121*, 347–361. https://doi.org/10.1016/j.eswa.2018.12.039

Ruiz, R., Pan, Q. K., & Naderi, B. (2019). Iterated Greedy methods for the distributed permutation flowshop scheduling problem. *Omega (United Kingdom), 83*, 213–222. https://doi.org/10.1016/j.omega.2018.03.004

Ruiz, R., & Stützle, T. (2007). A simple and effective iterated greedy algorithm for the permutation flowshop scheduling problem. European Journal of Operational Research, 177(3), 2033–2049. https://doi.org/DOI: 10.1016/j.ejor.2005.12.009.

Sethi, S. P., Sriskandarajah, C., Sorger, G., Blazewicz, J., & Kubiak, W. (1992). Sequencing of parts and robot moves in a robotic cell. International Journal of Flexible Manufacturing Systems, 4, 331–358. https://doi.org/DOI: 10.1007/BF01324886.

Shao, Z., Pi, D., & Shao, W. (2018). A novel discrete water wave optimization algorithm for blocking flow-shop scheduling problem with sequence-dependent setup times. Swarm and Evolutionary Computation, 40(July 2017), 53–75. https://doi.org/10.1016/j.swevo.2017.12.005.

Shao, Z., Pi, D., & Shao, W. (2020). Hybrid enhanced discrete fruit fly optimization algorithm for scheduling blocking flow-shop in distributed environment. *Expert Systems With Applications, 145*, 113147. https://doi.org/10.1016/j.eswa.2019.113147

Takano, M. I., & Nagano, M. S. (2019). Evaluating the performance of constructive heuristics for the blocking flow shop scheduling problem with setup times. *International Journal of Industrial Engineering Computations, 10*(1), 37–50. https://doi.org/10.5267/j.ijiec.2018.5.002

Tong, W., Miyano, E., Goebel, R., & Lin, G. (2018). An approximation scheme for minimizing the makespan of the parallel identical multi-stage flow-shops. *Theoretical Computer Science, 734*, 24–31. https://doi.org/10.1016/j.tcs.2017.09.018

Trovinger, S. C., & Bohn, R. E. (2005). Setup time reduction for electronics assembly: Combining simple (SMED) and IT-based methods. *Production and Operations Management, 14*(2), 205–217. https://doi.org/10.1111/j.1937-5956.2005.tb00019.x

Xu, Y., Wang, L., Wang, S., & Liu, M. (2013). An effective hybrid immune algorithm for solving the distributed permutation flow-shop scheduling problem. *Engineering Optimization, 46*(9), 1269–1283. https://doi.org/10.1080/0305215X.2013.827673

Zandieh, M., & Rashidi, E. (2009). An Effective Hybrid Genetic Algorithm for Hybrid Flow Shops with Sequence Dependent Setup Times and Processor Blocking. Journal of Industrial Engineering (Vol. 4). QIAU. Retrieved from http://www.qjie.ir/article_32.html.

Zhang, X., & Van De Velde, S. (2012). Approximation algorithms for the parallel flow shop problem. *European Journal of Operational Research, 216*(3), 544–552. https://doi.org/10.1016/j.ejor.2011.08.007

Zhao, F., Zhao, L., Wang, L., & Song, H. (2020). An ensemble discrete differential evolution for the distributed blocking flowshop scheduling with minimizing makespan criterion. *Expert Systems with Applications, 160*, 113678. https://doi.org/10.1016/j.eswa.2020.113678