# Single machine interfering jobs problem with flowtime objective[*]

Paz Perez-Gonzalez and Jose M. Framinan
Industrial Management, School of Engineering, University of Seville
pazperez,framinan@us.es

September 26, 2015

**Abstract**

Interfering jobs problems (or multi agents scheduling problems) are an emergent topic in the scheduling literature. In these decision problems, two or more sets of jobs have to be scheduled, each one with its own criteria. More specifically, we focus on a problem in which jobs belonging to two sets have to be scheduled in a single machine in order to minimize the total flowtime of the jobs in one set, while the total flowtime of the jobs in the other set should not exceed a given constant $\epsilon$. This problem is known to be weakly NP-hard, and a Dynamic Programming (DP) algorithm has been proposed to find optimal solutions.

In this paper, we first analyse the distribution of solutions of the problem in order to establish its empirical hardness. Next, a novel encoding scheme and a set of properties associated to the neighbourhood of this scheme are presented. These properties are used to develop both exact and approximate methods, i.e. a branch and bound (B&B) method, several constructive heuristics, and different versions of a genetic algorithm (GA). The computational experience carried out shows that the proposed B&B is more efficient than the existing DP algorithm. The results also show the advantages of the proposed encoding scheme, as the approximate methods yield close-to-optimum solutions for big-sized instances where exact methods are not feasible.

Scheduling interfering jobs two-agent scheduling problem total flowtime single machine

## 1 Introduction

Interfering jobs problems, or multi-agent scheduling problems, address the decision problem of scheduling jobs from different sets, each one with its own objective(s), and competing for the same machines (Agnetis et al., 2004). Multi-agent scheduling problems model many realistic situations in productive environments. In a recent review on the topic, Perez-Gonzalez and Framinan (2014) cite a number of real applications in different domains, such as supply chain scheduling (Fan, 2010), rescheduling (Unal et al., 1997; Perez-Gonzalez and Framinan, 2010; Hall and Potts, 2004; Yuan and Mu, 2007; Yuan et al., 2007; Mu and Gu, 2010), telecommunications (Peha and Tobagi,

[*]This is an Accepted Manuscript of an article published by Springer in the Journal of Intelligent Manufacturing, http://dx.doi.org/10.1007/s10845-015-1141-6

1990; Peha, 1995; Meiners and Torng, 2007; Arbib et al., 2004), and maintenance scheduling (Khelifati and Bouzid-Sitayeb, 2011; Wan et al., 2010; Kellerer and Strusevich, 2010).

In this class of problems, the inherent difficulty of multiobjective scheduling problems is emphasized by the need to consider different sets of jobs. For this reason, most literature addresses the single machine layout (e.g. Ni and Zhao, 2014; Yuan et al., 2013; Agnetis et al., 2013; Wu et al., 2012), which represents a case base for other multi-agent scheduling problems.

In this paper, we consider a single machine scheduling problem with two disjoint sets of jobs $\mathcal{J}^A$ and $\mathcal{J}^B$ where the objective is to minimize the total flowtime of jobs in set $\mathcal{J}^A$, $C^A_{sum}$, subject to that the total flowtime of jobs in set $\mathcal{J}^B$ is lower or equal than a given constant $\epsilon > 0$, i.e. $C^B_{sum} \leq \epsilon$. Adapting the notation from T'kindt and Billaut (2002) for multicriteria scheduling problems, our problem can be denoted as $1||\epsilon(C^A_{sum}/C^B_{sum})$. In manufacturing scheduling, this problem arises when the scheduling decision is taken on a rolling horizon basis, i.e.: when a set of incoming jobs (set $\mathcal{J}^A$) has to be scheduled for a given decision interval, but there are jobs already scheduled in the previous decision interval (set $\mathcal{J}^B$) whose schedule should not change greatly due to the fact that the required resources and raw materials have been already planned. In such case, it may be natural to minimize the flowtime of the jobs in set $\mathcal{J}^A$ (thus aiming at the efficiency in scheduling the new jobs) while bounding the disruption of the existing schedule (that of jobs in set $\mathcal{J}^B$).

Although the problem $1||C_{sum}$ is polynomially solvable by the SPT (Shortest Processing Time) rule, our problem was shown to be binary (or weakly) NP-hard by Agnetis et al. (2004). The same authors developed a Dynamic Programming algorithm with running time $\mathcal{O}(n^A n^B \epsilon)$, although they do not carry out a computational study on its efficiency. We are not aware of additional authors addressing this problem.

As we will discuss later, the nature of the problem $1|\epsilon(C^A_{sum}/C^B_{sum})$ may vary greatly depending on the specific values of $\epsilon$, the number of jobs in the different sets and their processing times. Therefore it is of interest to analyse the distribution of the solutions depending on these factors in order to provide efficient solutions for the different cases. In this paper, we tackle both issues, i.e.: we analyse the structure of solutions of the problem, and propose new exact and approximate methods for the problem. To do so, we define a new encoding scheme of the solutions (denoted *binary codification* in the following), using a property of the problem by Agnetis et al. (2004) based on the SPT rule. This codification allows us to reduce the solution space, and it is used to analyse empirically the distribution of the solutions of the problem depending on the instance size and on the value of $\epsilon$. The objective of this analysis is to identify the range of values of these factors that produce instances for which finding feasible/good solutions is easier or harder. Taking into account the conclusions of the analysis, we propose some properties of the neighbourhood structure that we use to develop different algorithms for the problem. More specifically, we develop a Branch and Bound (B&B), seven fast constructive, and three versions of a Genetic Algorithm (GA). Computational experiments to test the efficiency of the proposals are carried out in several test beds designed according to the findings gained when analysing the distribution of solutions.

The results show that the proposed B&B is more efficient than the DP for small sized problems. Moreover, the fast constructive heuristics proposed are able to provide approximate high-quality solutions within negligible computation times. Finally, the results show that the version of the GA embedding the binary codification performs better than the classical permutation encoding.

The paper is structured as follows: in Section 2 the problem and the proposed binary codification are formally stated. Section 3 presents the distribution of solutions depending on the parameters of the problem. In Section 4 some properties of the neighbourhood structure according to the new codification are proposed. In Section 5, the description of the B&B, the constructive heuristics and the three versions of GA are given. Based on the analysis conducted in Section 3, test beds of different sizes and empirical hardness are proposed in Section 6 together with the computational experiments. Finally, conclusions and future research lines are presented in Section 7.

## 2 Problem statement and codification

This paper considers two sets of jobs $\mathcal{J}^A$ and $\mathcal{J}^B$ with $n^A$ and $n^B$ jobs respectively, verifying that $\mathcal{J}^A \cap \mathcal{J}^B =$, and $\mathcal{J} = \mathcal{J}^A \cup \mathcal{J}^B$ with $n = n^A + n^B$ jobs. The processing times of jobs in $\mathcal{J}^X$, $X = A$ or $B$, are denoted $p_j^X$, $p^X = \sum_{j \in \mathcal{J}^X} p_j^X$. Let $\sigma$ be a complete schedule formed by all jobs in the system, represented by a vector formed by the $n$ elements of $\mathcal{J}$: $\sigma = [\sigma_1, \ldots, \sigma_n]$ where each $\sigma_i$ is in some $\mathcal{J}^X$, $i = 1, \ldots, n$. A partial schedule of jobs in $\mathcal{J}^X$ is denoted $\sigma^X$, $X = A$ or $B$.

The completion time of job $\sigma_j \in \sigma$ is denoted as $C_j(\sigma)$. Then, $C_{sum}^X(\sigma) = \sum_{j \in \mathcal{J}^X} C_j(\sigma)$ is the total flowtime or total completion time of a complete schedule $\sigma$ for jobs in $\mathcal{J}^X$. In the following, we omit $\sigma$ in the objective function unless it may lead to confusion. As objective we consider the minimization of $C_{sum}^A$ subject to $C_{sum}^B \leq \epsilon$, with $\epsilon$ a given constant. The problem is denoted $1|\epsilon(C_{sum}^A/C_{sum}^B)$. Finally, let $\sigma_{SPT}^X$ be the sequence formed by jobs in $X = A$ or $B$ according to the SPT order.

Note that the decision problem can be infeasible or feasible depending on the values of $\epsilon$. More specifically, we consider the following cases:

- $\epsilon_{\min} = C_{sum}^B(\sigma_{SPT}^B)$.

  If $\epsilon = \epsilon_{\min}$ the problem is trivial, and an optimal schedule is $\sigma^* = \sigma_{SPT}^B \cup \sigma_{SPT}^A$ with $C_{sum}^A(\sigma^*) = n^A p^B + C_{sum}^A(\sigma_{SPT}^A)$. Note that $\epsilon_{\min} = C_{sum}^B(\sigma^*)$.

  If $\epsilon < \epsilon_{\min}$ the problem is infeasible.

- $\epsilon_{\max} = C_{sum}^B(\sigma_{SPT}^B) + n^B p^A$.

  If $\epsilon \geq \epsilon_{\max}$ the problem is trivial, and the optimal schedule is $\sigma^* = \sigma_{SPT}^A \cup \sigma_{SPT}^B$ with $C_{sum}^A(\sigma^*) = C_{sum}^A(\sigma_{SPT}^A)$. Note that $\epsilon_{\max} = C_{sum}^B(\sigma^*)$.

For $\epsilon$ between $\epsilon_{\min}$ and $\epsilon_{\max}$ the problem is known to be weakly NP-hard (Agnetis et al., 2004), so it is not possible to find the optimal solution (sequence) in polynomial time. It is clear that

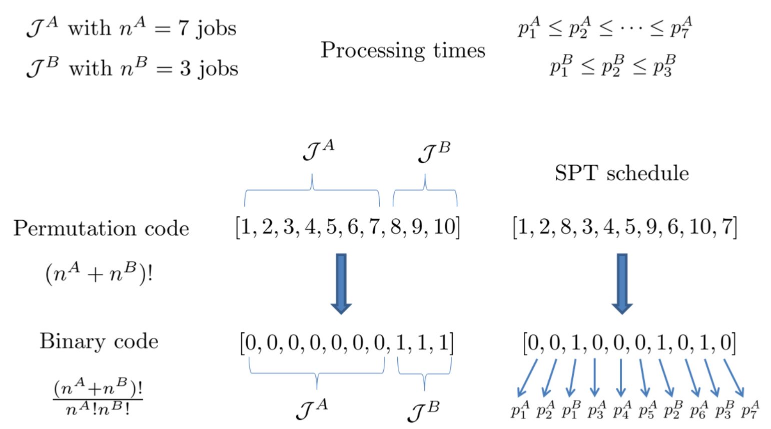

Figure 1: Binary codification

the number of different sequences for this problem is $n!$ (i.e. the number of possible permutations for $n$ jobs). However, the following property allows reducing this number of solutions:

(Agnetis et al., 2004) In an optimal sequence, jobs in $\mathcal{J}^A$ and jobs in $\mathcal{J}^B$ follow the Shortest Processing Time first (SPT) rule.

In view of this property, we define a *SPT sequence* as a sequence in which jobs in set $\mathcal{J}^A$ are ordered according to the SPT rule, whereas jobs in set $\mathcal{J}^B$ are ordered according to the SPT rule. Note that, in a SPT sequence, jobs belonging to different sets may not follow the SPT rule. In the following, without loss of generality we will assume that the processing times of jobs in $\mathcal{J}^A$ and $\mathcal{J}^B$ are given in SPT order respectively. Property 2 states that the optimal solution is one out of all possible SPT sequences, therefore permutation sequences that do not belong to the class of SPT sequences can be discarded. However, it is difficult to check if a permutation solution verifies Property 2, since in SPT sequences, jobs in $\mathcal{J}^A$ always maintain the same partial order, and jobs in $\mathcal{J}^B$ too.

A more efficient way to use Property 2 is to define another representation of the solutions, that we label *binary encoding*. In this representation, jobs in $\mathcal{J}^A$ are coded as zero whereas jobs in $\mathcal{J}^B$ are coded as one. Thus, a sequence of $n_A$ zeros and $n_B$ ones represents univocally a given SPT sequence, since we can determine the job that corresponds to each zero or each one in the schedule (see Figure 1). By using this codification, the size of the solution space is smaller than that when using the permutation codification. More specifically, we only consider $\frac{(n^A+n^B)!}{n^A!n^B!}$ sequences out of a total of $(n_A + n_B)!$ sequences. The proposed encoding scheme may allow the development of methods based on the systematic exploration of solutions. More specifically, in view of the need to determine the factors influencing the feasibility/difficulty of the problem mentioned in Section 1, we will conduct an analysis of the distribution of solutions in Section 3. From the insights gained in this analysis, in Section 4 we will derive further properties which will serve us to develop efficient exact and approximate methods (Section 5).

## 3 Distribution of solutions

The nature of the problem $1|\epsilon(C^A_{sum}/C^B_{sum})$ may vary greatly depending on the number of jobs in the different sets and on the specific values of $\epsilon$ related to the processing times of the jobs in both sets. For instance, it is clear that for very loose $\epsilon$ values (as compared to the processing times of the jobs), many different schedules are feasible, and the problem is basically to minimize the flowtime of the jobs in set $\mathcal{J}^A$ since there may be great differences among the $C^A_{sum}$ of the feasible schedules. In contrast, for tight $\epsilon$ values, perhaps only a few feasible schedules exist, maybe with similar values of $C^A_{sum}$. In such case, the decision problem turns to be more similar to a feasibility problem rather than to a minimisation problem, as the few feasible solutions would consists on first scheduling all/most jobs in set $\mathcal{J}^B$. Therefore, it is important to investigate the different nature of the problems depending on the combination of the number of jobs in $\mathcal{J}^A$ and $\mathcal{J}^B$, and $\epsilon$ (expressed as a function of the processing times of the jobs). Such analysis can provide the basis for designing and testing new solution methods, both exact and approximate.

As mentioned before, the problem under consideration is weakly NP-hard. However, in practice, simple methods may find good or even optimal solutions for some NP-hard scheduling problems whereas in other problems, sophisticated methods are required to even find acceptable solutions. Such 'empirical hardness' is determined by the distribution of the space of solutions of the problem, as it may turn out that there are many solutions with objective values close to the optimum. In such case, acceptable or even good values of the objective function can be found by random schedules. Therefore, analysing the empirical hardness of the scheduling problem is of great practical interest, as it allows us know whether is worth developing sophisticated algorithms for a problem, or not, as the effort may not pay off. In addition, such analysis may provide the basis for designing appropriate test beds for the problem, thus determining the parameters that make the problem to be 'easy' or 'hard'. The usual way to conduct these analysis is to establish the empirical frequencies of finding sequences that are under a given percentage of the optimal value. Note that, in order to perform such analysis, not only the optimal value has to be determined, but all possible sequences in the space of solutions have to be evaluated in order to measure their distance to the optimum.

Despite the usefulness of such analysis, these studies are often neglected in the literature (Framinan et al., 2014). However, there are some contributions on the topic. For instance, Perez-Gonzalez and Framinan (2009) study a strongly NP-hard scheduling problem for which, in some cases, almost all solutions (99.6%) have an approximation percentage to the optimal value of less than 2%. So, finding a good solution in such cases is 'easy'. Taillard (1990), Armentano and Ronconi (1999) and Framinan et al. (2001) conduct similar analysis on other scheduling problems concluding that the corresponding problems are 'empirically hard'. To the best of our knowledge, this kind of study have been not carried out for interfering jobs problems. The only related reference is Khowala et al. (2014), who study the structure of two interfering jobs problems (under the Pareto approach), but their methodology is quite different as their multi-objective approach is not the same.

The main drawback of this analysis is the fact that it must be confined to small problem sizes, although, in our case, the binary codification introduced in the previous section may help to address bigger instances, since it requires evaluating $n_T = \frac{(n^A+n^B)!}{n^A!n^B!}$ sequences for each instance instead of the $(n^A + n^B)!$ sequences with the standard permutation codification. Note that, since not all sequences are feasible, two indicators are considered when analysing the empirical hardness of our problem:

- The relative number of feasible solutions $rf$, i.e. the ratio between the number of feasible solutions $n_f$ of the problem instances with respect to the total number of solutions $n_T = \frac{(n^A+n^B)!}{n^A!n^B!}$. More specifically:
$$rf = \frac{n_f}{n_T} \cdot 100$$
For our problem, this ratio clearly depends on the value of $\epsilon$ and on the processing times of the jobs belonging to each set. After evaluating all solutions of each instance, $n_f$ is obtained so $rf$ can be derived, and then it can be averaged for several instances. According to this indicator, we define the empirical *hardness with respect to feasibility* (HF) of a problem as the (empirical) difficulty to find feasible schedules.

- The relative distance of the feasible solutions to the optimal solution $rd$, i.e.: If $\sigma^*$ is the optimal solution ($C^A_{sum}(\sigma^*) \leq C^A_{sum}(\sigma)$, $\forall\sigma$ feasible), then $rd$ can be computed for each solution as in Taillard (1990):
$$rd = \frac{C^A_{sum}(\sigma) - C^A_{sum}(\sigma^*)}{C^A_{sum}(\sigma^*)} \cdot 100$$
By counting the number of times that the solutions are within a given relative distance to the optimal solution with respect to the number of feasible solutions, we can construct the empirical cumulative distribution function (cdf) yielding the probability of a random solution to be within a given relative distance to the optimal solution. This cdf also gives an indication of the variability of the objective function. According to this indicator, we define the empirical *hardness with respect to quality* (HQ) of a problem as the difficulty to find (feasible) schedules close to the optimal. Clearly, the higher the variability, the more difficult the problem is, since the probability of a random solution to be close to the optimal is lower. Note that the number of times that the solutions are within a given relative distance to the optimum can be normalised with respect either to the total number of possible solutions ($n_T$), or to the number of feasible sequences ($n_f$). In our analysis we have adopted the second approach in order to distinguish the feasibility aspects of the problem (i.e. how likely is to find a feasible solution) from the optimality aspects (i.e. how likely is to find a good solution).

A series of preliminary experiments shows that, as foreseeable, instances of the $1||\epsilon(C^A_{sum}/C^B_{sum})$ problem could have different structure of solutions depending on certain values of $n^A$, $n^B$, and

Table 1: Percentage of feasible solutions $rf$

| | | $\alpha$ | | | | |
|---|---|---|---|---|---|---|
| $n^A$ | $n^B$ | 0.2 | 0.4 | 0.6 | 0.8 | Aver. |
| 5 | 20 | 12.1031 | 56.6029 | 90.5590 | 99.3303 | 64.6488 |
| | 15 | 13.1127 | 56.4880 | 89.8013 | 99.1338 | 64.6340 |
| | 10 | 14.9950 | 56.2970 | 88.3350 | 98.6547 | 64.5704 |
| | 5 | 19.2063 | 55.3968 | 84.2857 | 96.6270 | 63.8790 |
| 10 | 20 | 9.7848 | 67.5351 | 97.3084 | 99.9668 | 68.6488 |
| | 15 | 11.2345 | 66.8304 | 96.6568 | 99.9368 | 68.6646 |
| | 10 | 13.8711 | 65.6915 | 95.3331 | 99.8345 | 68.6826 |
| | 5 | 19.8968 | 63.4266 | 91.5018 | 99.1275 | 68.4882 |
| 15 | 20 | 7.1052 | 69.3507 | 98.5913 | 99.9884 | 68.7589 |
| | 15 | 8.6749 | 68.3683 | 98.0317 | 99.9868 | 68.7654 |
| | 10 | 11.5637 | 66.8426 | 96.6568 | 99.9464 | 68.7524 |
| | 5 | 18.2598 | 64.0473 | 92.9870 | 99.5130 | 68.7018 |
| 20 | 15 | 7.8457 | 71.2204 | 98.8205 | 99.9964 | 69.4708 |
| | 10 | 10.9965 | 69.2182 | 97.7723 | 99.9783 | 69.4913 |
| | 5 | 18.1993 | 65.7271 | 94.1585 | 99.6868 | 69.4429 |
| | Aver. | 13.1233 | 64.2029 | 94.0533 | 99.4472 | 67.7067 |
| | St. Dv | 4.1890 | 5.3833 | 4.3492 | 0.8837 | 2.0045 |

$\epsilon$. Based on these preliminary results, a test bed is designed to analyse these parameters in depth. More specifically, we have generated ten small instances for each problem size combining different values of $n^A \in \{5, 10, 15, 20\}$, $n^B \in \{5, 10, 15, 20\}$ (except the case $n^A \times n^B = 20 \times 20$ since the computational requirements are too high). The processing times of each job is randomly generated from a $[1, 99]$ uniform distribution. Then, for each problem instance, the value of $\epsilon$ is computed as in Agnetis et al. (2009), i.e.: $\epsilon \in [\epsilon_{min}, \epsilon_{max}]$, for a given $\alpha \in (0, 1)$:

$$\epsilon = \epsilon_{min} + \alpha(\epsilon_{max} - \epsilon_{min}) \tag{1}$$

with $\epsilon_{\min} = C^B_{sum}(\sigma^B_{SPT} \cup \sigma^A_{SPT})$, and $\epsilon_{\max} = C^B_{sum}(\sigma^A_{SPT} \cup \sigma^B_{SPT})$. In our experiments, we consider different levels of $\alpha$, i.e. $\alpha \in \{0.2, 0.4, 0.6, 0.8\}$. Once the test bed has been generated, all possible solutions for each instance have been obtained by complete enumeration, and the corresponding average values of $rf$ and the empirical cdf of $rd$ have been obtained for each problem size and $\alpha$.

Regarding HF, in Table 1, the average results of $rf$ for each size is shown. In average, the percentage is around 65-70% of feasible solutions. According to $\epsilon$, and as expected, the percentage of feasible solutions increases with $\alpha$ (i.e. with $\epsilon$). In particular, we can observe the following aspects: For the tighter values of $\epsilon$ ($\alpha = 0.2$), $rf$ decreases with $n^A$ and $n^B$, so HF increases with the problem size. However, the performance is different for $\alpha \in \{0.4, 0.6, 0.8\}$, since HF decreases with the problem size. The values of the standard deviation of $rf$ show that there is not a high variability among instances.

Regarding HQ, the results can be observed in Figure 2, which shows the empirical cdf of $rd$ depending on $\alpha$, and in Figure 3, which shows the distribution of feasible solutions depending

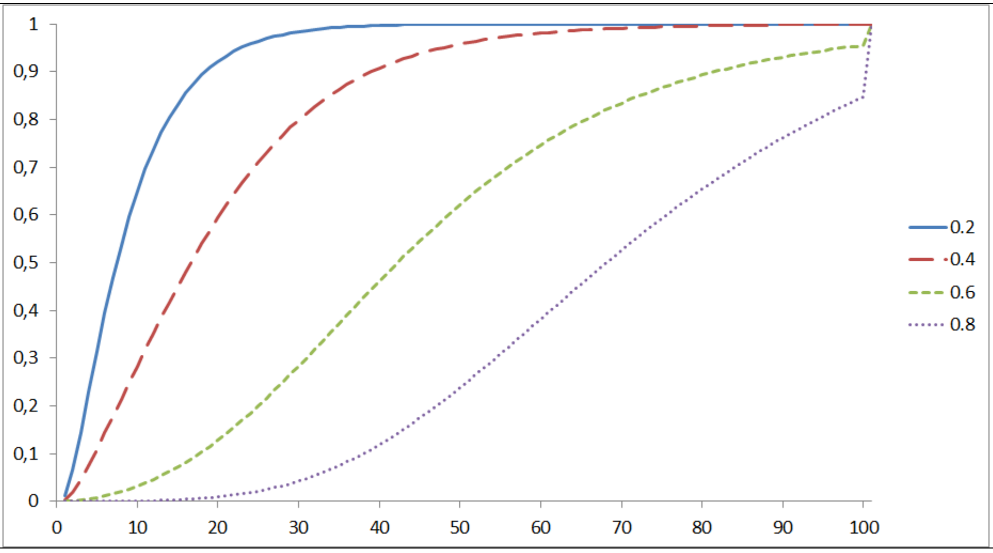

Figure 2: Distribution of feasible solutions for different values of $\alpha$

on the size of the problem according to the ratio $r_n = \frac{n^A}{n^B}$. These figures have been developed by aggregating the values of $rd$ for all instances in the test bed for a given value of $\alpha$ and $r_n$ respectively. They must be interpreted in the following manner: the value in the $y$-axis indicates the empirical probability of finding a solution with a lower or equal relative distance to the optimum given by the $x$-axis. Therefore, the lowest the curve, the more difficult is to randomly find a good solution for the problem.

On one hand, the curves in Figure 2 reach lower values as $\alpha$ increases, which means that HQ is higher as $\epsilon$ increases. Note that a high percentage of solutions are very far to the optimum, in some cases farther than the 100% of distance, taking into account the vertical line which appears in the 100% for the highest values of $\alpha$.

On the other hand, Figure 3 shows that, when $r_n$ is lower than 1 (i.e. $n_A < n_B$) the feasible solutions are more distant to the optimum than in the opposite case. $n^A = n^B$ is a medium term case. Finally, for values of $r_n < 1$, a high percentage of solutions are very far to the optimum, farther than the 100% of distance. Therefore, HQ increases when $r_n$ decreases, that is, while $n^A < n^B$ and the difference among the sizes of each set is greater.

Summarizing, we can observe that:

- Regarding HF (see Table 1):
  - As expected, it decreases when $\alpha$ increases (i.e. when $\epsilon$ increases).
  - When $n_A$ and $n_B$ increases, it decreases for the smallest values of $\epsilon$ ($\alpha = 0.2$). However, it increases for medium and high values of $\epsilon$ ($\alpha \in \{0.4, 0.6, 0.8\}$). Moreover, when $n_A < n_B$ HF decreases for the smallest values of $\epsilon$, and increases for medium and high values of $\epsilon$.
- Regarding HQ:

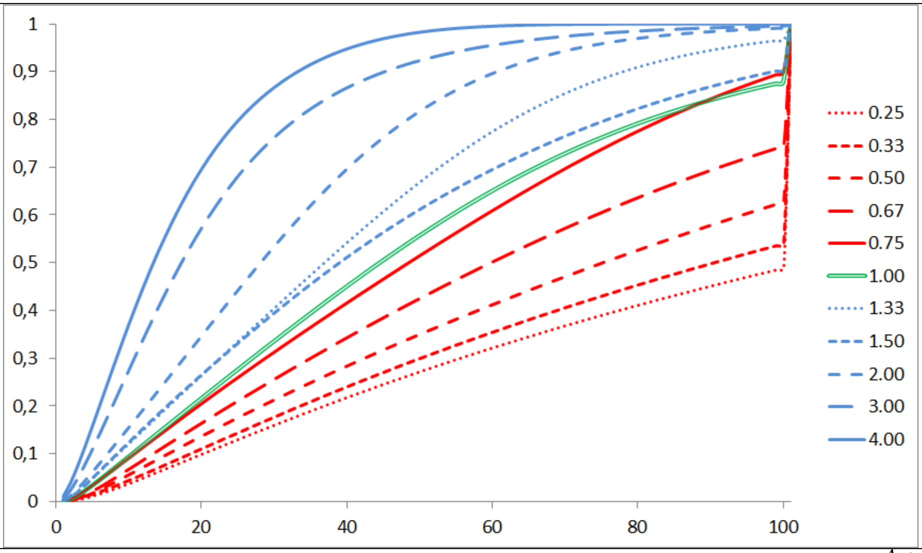

Figure 3: Distribution of feasible solutions for different values of $r_n = n^A/n^B$

- It increases when $\alpha$ increases, i.e. when $\epsilon$ increases (see Figure 2).
- It increases when $r_n < 1$, i.e. when $n_A < n_B$ (see Figure 3).

Finally, the analysis carried out in this section allows us to derive the following conclusions:

1. For the smallest values of $\epsilon$ ($\alpha = 0.2$), the number of feasible solutions is low and the problem is difficult in terms of feasibility. However, these solutions are close to the optimal solution, so it is relatively likely that a random feasible solution is a good solution for the problem.

2. For medium/high values of $\epsilon$, ($\alpha \in \{0.4, 0.6, 0.8\}$), the number of feasible solutions is high, so finding a feasible solution is easy. However, the distance to the optimal is large, therefore is it likely that a random feasible solution is far from the optimum.

3. When $n_A < n_B$ the problem is difficult regarding both HF and HQ.

The observations above allow us to generate instances with different hardness degrees depending on the parameters of the instance. We will use these inputs to generate three test beds in Section 6. In addition, the results about HQ show that heuristics focused on properties of the neighbourhood structure can be useful for this problem (see Section 5), since random solutions have a high probability to be very far to the optimal solution and these properties may help in discarding bad solutions and in restricting the search space. Finally, the results about HF show the importance to design methods to repair the infeasibility of schedules, since finding feasible schedules is not trivial (see Section 4).

With the above results in mind, a number of properties focused on several aspects of the neighbourhood (including infeasibility repairing) have been developed. These properties are presented in the next section.

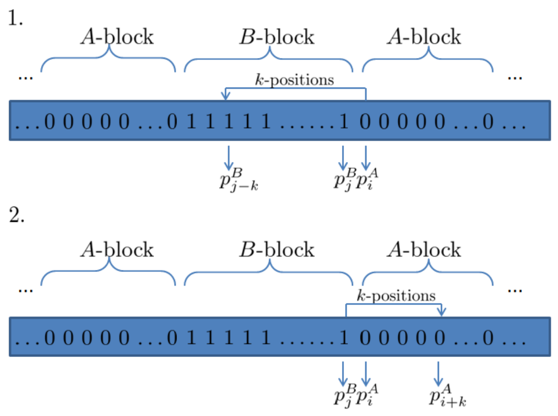

Figure 4: Backward and Forward Insertion Improvement

## 4 Properties of the neighbourhood structure

As mentioned before, the binary codification of Section 2 serves to reduce the space of solutions from $n!$ to $\frac{(n^A+n^B)!}{n^A!n^B!}$ possible sequences. However, this reduction implies that the usual neighbourhood operators defined for permutation sequences cannot be directly applied, as they may provide redundant sequences, i.e. in the binary encoding the interchange of positions between two jobs does not necessarily lead to different solutions. The following theorem formalises this statement (note that, in the following, all sequences are assumed to be binary-encoded).

Let $\sigma'$ be a sequence obtained by swapping two positions of a sequence $\sigma$. If the jobs in these positions belong to the same set, then $\sigma' = \sigma$.

Trivial.

In order to provide non redundant schedules by using the usual neighbourhood operators, we propose several results using the following definitions:

Let $\sigma$ be a sequence and $j$ a job belonging to a set $X = A$ or $B$ in the position $l$ in $\sigma$. We define a forward (backward) insertion of step $k$ as the insertion of the job $j$ in position $l - k$ $(l + k)$, obtaining a new sequence $\sigma'$.

A $X$-block is defined as a subsequence of consecutive jobs of $\mathcal{J}^X$ in $\sigma$, with $X = A$ or $B$.

A forward (backward) insertion of a job belonging to a set $X = A$ or $B$ does not provide a redundant schedule if and only if the jobs between the initial deletion position and the final insertion position include elements of both sets. Therefore, in a valid forward (backward) insertion, the selected job must be the first (last) job of its $X$-block, and it has to be inserted into another block formed by jobs in the other set (see Figures 4 and 5).

Based on these results, it is possible to improve the total flowtime of jobs in $\mathcal{J}^A$ of a given a feasible sequence while maintaining its feasibility by the following insertion methods (see Figure 4):

**(Backward Insertion Improvement)** Let $\sigma$ be a SPT sequence and $\sigma'$ the sequence obtained by a backward insertion of step $k$ of the first job of an $A$-block, with processing time $p^A_i$, in the adjacent $B$-block of size $b$ (see Figure 4 1.).

Then, $C^A_{sum}(\sigma') \leq C^A_{sum}(\sigma)$. Moreover, if $\sigma$ is feasible, then $\sigma'$ is feasible if and only if

$$k = \min\left\{b, Int\left[\frac{\epsilon - C^B_{sum}(\sigma)}{p^A_i}\right]\right\}$$

The total flowtime of $\mathcal{J}^A$ and $\mathcal{J}^B$ are:

$$C^A_{sum}(\sigma') = C^A_{sum}(\sigma) - \sum_{l=j-k}^{j} p^B_l$$

$$C^B_{sum}(\sigma') = C^B_{sum}(\sigma) + k \cdot p^A_i$$

First, it is trivial to proof that $C^A_{sum}(\sigma') \leq C^A_{sum}(\sigma)$. If $\sigma$ is feasible then $C^B_{sum}(\sigma) \leq \epsilon \Rightarrow \epsilon - C^B_{sum}(\sigma) \geq 0$. Let $\sigma'$ be the sequence obtained by inserting backward the first job of an $A$-block in the adjacent $B$-block. If is moved $k \leq b$ positions, then

$$C^B_{sum}(\sigma') = C^B_{sum}(\sigma) + kp^A_i$$

$\sigma'$ is feasible $\Leftrightarrow C^B_{sum}(\sigma') \leq \epsilon \Leftrightarrow C^B_{sum}(\sigma) + kp^A_i \leq \epsilon \Leftrightarrow kp^A_i \leq \epsilon - C^B_{sum}(\sigma) \Leftrightarrow k \leq \frac{\epsilon - C^B_{sum}(\sigma)}{p^A_i}$

**(Forward Insertion Improvement)** Let $\sigma$ be a SPT sequence and $\sigma'$ the sequence obtained by forward insertion of step $k$ of the last job of a $B$-block, with processing time $p^B_j$, in the adjacent $A$-block of size $a$ (see Figure 4 2.).

Then, $C^A_{sum}(\sigma') \leq C^A_{sum}(\sigma)$. Moreover, if $\sigma$ is feasible, $\sigma'$ is feasible if and only if $k \leq a$ and

$$\sum_{l=i}^{i+k} p^A_l \leq \epsilon - C^B_{sum}(\sigma)$$

The total flowtime of $\mathcal{J}^A$ and $\mathcal{J}^B$ are:

$$C^A_{sum}(\sigma') = C^A_{sum}(\sigma) - kp^B_j$$

$$C^B_{sum}(\sigma') = C^B_{sum}(\sigma) + \sum_{l=i}^{i+k} p^A_l$$

It is trivial to proof that $C^A_{sum}(\sigma') \leq C^A_{sum}(\sigma)$. The last job of a $B$-block is inserted forward $k \leq a$ positions, then

$$C^B_{sum}(\sigma') = C^B_{sum}(\sigma) + \sum_{l=i}^{i+k} p^A_l$$

$\sigma'$ is feasible $\Leftrightarrow C^B_{sum}(\sigma') \leq \epsilon \Leftrightarrow C^B_{sum}(\sigma) + \sum_{l=i}^{i+k} p^A_l \leq \epsilon \Leftrightarrow \sum_{l=i}^{i+k} p^A_l \leq \epsilon - C^B_{sum}(\sigma)$

In view of the previous properties, the following corollary holds:

The sequence obtained by the pairwise exchange of the last job of a $B$-block, with processing

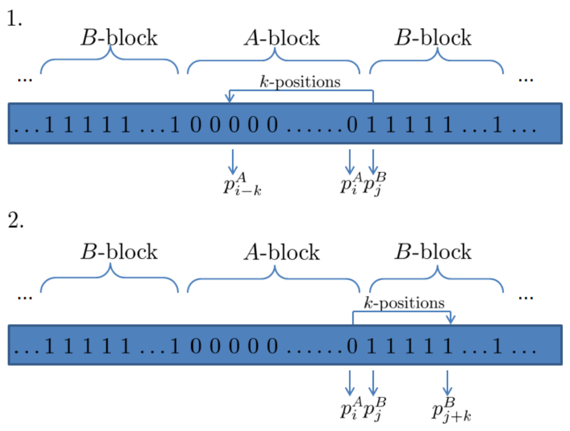

Figure 5: Backward and Forward Insertion Infeasibility Repair

time $p^B_j$, and the first job of the adjacent $A$-block, with processing time $p^A_i$, provides the same sequence than the backward (and forward) insertion improvement of step $k = 1$, being $C^A_{sum}(\sigma') = C^A_{sum}(\sigma) - p^B_j$ and $C^B_{sum}(\sigma') = C^B_{sum} + p^A_i$.

Backward and forward insertions can be used also to repair infeasible sequences at the expense of obtaining worst values of the flowtime of the jobs in $A$, according to the following properties (see Figure 5):

**(Backward Insertion Infeasibility Repair)** Let $\sigma$ be a SPT sequence and $\sigma'$ the sequence obtained by the backward insertion of step $k$ of the first job of a $B$-block, with processing time $p^B_j$, in the adjacent $A$-block with size $a$ (see Figure 5 1.).

Then, $C^B_{sum}(\sigma') \leq C^B_{sum}(\sigma)$. Moreover, if $\sigma$ is infeasible, then there are two possible minimum values of $k$:

- $k = a$ if $\sum_{\forall l \in A-\text{block}} p^A_l < C^B_{sum}(\sigma) - \epsilon$, and in this case $\sigma'$ is infeasible, or
- The minimum value of $k$ that verifies $\sum_{l=i-k}^{i} p^A_l \geq C^B_{sum}(\sigma) - \epsilon$. In this case $\sigma'$ is feasible.

The total flowtime of $\mathcal{J}^A$ and $\mathcal{J}^B$ are:

$$C^A_{sum}(\sigma') = C^A_{sum}(\sigma) + kp^B_j$$

$$C^B_{sum}(\sigma') = C^B_{sum}(\sigma) - \sum_{l=i-k}^{i} p^A_l$$

It is trivial to proof that $C^A_{sum}(\sigma') \geq C^A_{sum}(\sigma)$. Moreover, if $\sigma$ is infeasible then $C^B_{sum}(\sigma) > \epsilon$. Let $\delta = C^B_{sum}(\sigma) - \epsilon > 0$. Let $\sigma'$ be the sequence obtained by backward insertion of step $k$ ($k \leq a$) of the first job of a $B$-block. Then, the following equation holds:

$$C^B_{sum}(\sigma') = C^B_{sum}(\sigma) - \sum_{l=i-k}^{i} p^A_l$$

If $\sum_{l\in A-\text{block}} p_l^A < \delta$ then the maximum value of $k$ is $a$, and the job is inserted before all jobs in the adjacent $A$-block. In this case $C_{sum}^B(\sigma') = C_{sum}^B(\sigma) - \sum_{l\in A-\text{block}} p_l^A > C_{sum}^B(\sigma) - (C_{sum}^B(\sigma) - \epsilon) > \epsilon$, therefore $\sigma'$ is infeasible.

If $\sum_{l\in A-\text{block}} p_l^A \geq \delta$, let $k \leq a$ be the minimum value such that $\sum_{l=i-k}^{i} p_l^A \geq \delta$. In this case $C_{sum}^B(\sigma') = C_{sum}^B(\sigma) - \sum_{l=i-k}^{i} p_l^A \leq C_{sum}^B(\sigma) - (C_{sum}^B(\sigma) - \epsilon) \leq \epsilon$, therefore $\sigma'$ is feasible.

**(Forward Insertion Infeasibility Repair)** Let $\sigma$ be a SPT sequence and $\sigma'$ the sequence obtained by forward insertion of step $k$ of the last job of a $A$-block, with processing time $p_i^A$, in the adjacent $B$-block with size $b$ (see Figure 5 2.).

Then, $C_{sum}^B(\sigma') \leq C_{sum}^B(\sigma)$. Moreover, if $\sigma$ is infeasible, then the minimum value of $k$ is:

- $k = b$ if $bp_i^A < C_{sum}^B(\sigma) - \epsilon$, and in this case $\sigma'$ is infeasible, or
- $k = Int\left[\frac{C_{sum}^B(\sigma)-\epsilon}{p_i^A}\right] + 1$. In this case $\sigma'$ is feasible.

The total flowtime of $\mathcal{J}^A$ and $\mathcal{J}^B$ are:

$$C_{sum}^A(\sigma') = C_{sum}^A(\sigma) + \sum_{l=j}^{j+k} p_l^B$$

$$C_{sum}^B(\sigma') = C_{sum}^B(\sigma) - kp_i^A$$

It is trivial to proof that $C_{sum}^A(\sigma') \geq C_{sum}^A(\sigma)$. Moreover, if $\sigma$ is infeasible then $C_{sum}^B(\sigma) > \epsilon$. Let $\delta = C_{sum}^B(\sigma) - \epsilon > 0$. Let $\sigma'$ be the sequence obtained by forward insertion of step $k$ ($k \leq b$) of the last job of an $A$-block. Then, the following equation holds:

$$C_{sum}^B(\sigma') = C_{sum}^B(\sigma) - kp_i^A$$

If $bp_i^A < \delta$ then the maximum value of $k$ is $b$, and the job is inserted after all jobs in the adjacent $B$-block. In this case $C_{sum}^B(\sigma') = C_{sum}^B(\sigma) - bp_i^A > C_{sum}^B(\sigma) - (C_{sum}^B(\sigma) - \epsilon) > \epsilon$, therefore $\sigma'$ is infeasible.

If $bp_i^A \geq \delta$, let $k \leq b$ be the minimum value such that $kp_i^A \geq \delta$, i.e. $k = Int\left[\frac{\delta}{p_i^A}\right] + 1$. Therefore, $C_{sum}^B(\sigma') = C_{sum}^B(\sigma) - kp_i^A \leq C_{sum}^B(\sigma) - (C_{sum}^B(\sigma) - \epsilon) \leq \epsilon$ and $\sigma'$ is feasible.

According to the results provided by Properties 4 and 4, the following corollary holds:

The sequence obtained by pairwise exchange of the last job of an $A$-block with processing time $p_i^A$ and the first job of the adjacent $B$-block with processing time $p_j^B$, provides the same sequence than the forward (backward) insertion infeasibility repair of step $k = 1$, being $C_{sum}^A(\sigma') = C_{sum}^A(\sigma) + p_j^B$ and $C_{sum}^B(\sigma') = C_{sum}^B - p_i^A$.

Finally, a new property allows to reduce the space of search in a construction process, providing a bounding method by the following lower bounds:

Let $\sigma$ be a feasible schedule with $C_{sum}^A(\sigma) = UB$. Let $\sigma'$ be a partial schedule formed by $n' \leq n$ jobs in $\mathcal{J} = \mathcal{J}^A \bigcup \mathcal{J}^B$, and $S$ a set containing the $n - n'$ unscheduled jobs.

1. Let $\sigma'_A = \sigma' \bigcup [0, \ldots, 0]$ be the schedule formed by the union of $\sigma'$ and all unscheduled jobs in $\mathcal{J}^A$. If $C^A_{sum}(\sigma'_A) \geq UB$, then no schedule $\sigma' \bigcup \sigma_S$ (for all $\sigma_S$ formed by jobs in $S$) exists such as $C^A_{sum}(\sigma' \bigcup \sigma_S) < C^A_{sum}(\sigma)$.
2. Let $\sigma'_B = \sigma' \bigcup [1, \ldots, 1]$ be the schedule formed by the union of $\sigma'$ and all unscheduled jobs in $\mathcal{J}^B$. If $C^B_{sum}(\sigma'_B) > \epsilon$, then no feasible schedule $\sigma' \bigcup \sigma_S$ (for all $\sigma_S$ formed by jobs in $S$) exists.

Trivial.

As a summary of this section, we have presented a number of properties that may allow us to efficiently implement a number of neighbourhood operators. In the next section, these properties will be embedded in both exact and approximate solution procedures.

## 5 Solution methods

To the best of our knowledge, the only solution procedure available for the $1||\epsilon(C^A_{sum}/C^B_{sum})$ problem is the Dynamic Programming algorithm (DP) by Agnetis et al. (2004). This (exact) pseudo-polynomial algorithm exploits Property 2. The Bellman equation is the following:

$$F(i,j,q) = \min\{F(i-1,j,q) + P(i,j), F(i,j-1,q-P(i,j))$$

with $P(i,j) = \sum_{k=1}^{i} p_i^A + \sum_{k=1}^{j} p_j^B$. Note that $F(i,j,q)$ yields the value of an optimal solution for the $i$ first jobs in $J^A$, $j$ first jobs in $J^B$, and $q = 0, \ldots, \epsilon$. The formula is iteratively solved by increasing values of $i$ and $j$. We are not aware of approximate methods for the problem.

Using the binary encoding in Section 2, and the properties derived in Section 4, we propose both exact and approximate solution procedures. First, in Subsection 5.1 we present a Branch and Bound (B&B) algorithm for providing exact solutions for the problem. The computational experience carried out in Section 6.2 shows that it is more efficient than the DP. Regarding approximate methods, in subsection 5.2 we propose fast constructive heuristics while in Subsection 5.3 we present two versions of a Genetic Algorithm: GA(B) using the binary codification, and GA(B)_I using the binary codification plus the improvement properties seen in the previous section. The experiments in 6.4 show that these two versions are more efficient than the same structure of GA using the standard permutation codification, GA(P).

### 5.1 Branch and Bound Algorithm

Branch and bound has been widely applied in scheduling literature (see e.g. Sabouni et al., 2010; Gawiejnowicz et al., 2010; Framinan, 2007). In this section, we develop a B&B algorithm based on the binary encoding and Property 4 to generate lower bounds. In the following, we assume that, given a problem instance, the sequence $\sigma^B_{SPT} \cup \sigma^A_{SPT}$ yields a flowtime lower or equal than $\epsilon$ (i.e. it is feasible). Otherwise, as discussed in Section 2, there is no feasible solution for the instance. Then, the B&B consists of two steps:

```
procedure UpdateTree($\sigma$)
    if $\sigma$ is complete;
        if $C^A_{sum}(\sigma) < UB$
            $UB = C^A_{sum}(\sigma)$;
        end if
    else
        $LB_A = C^A_{sum}(\sigma \bigcup \sigma^A)$
        $LB_B = C^B_{sum}(\sigma \bigcup \sigma^B)$
        if $\sigma^A \neq$
            if $LB_B \leq \epsilon$ & $LB_A \leq UB$
                $\sigma = \sigma \bigcup 0$;
                $\sigma^A = \sigma - 0$;
                UpdateTree($\sigma$)
            end if
        end if
        if $\sigma^B \neq$
            if $LB_B \leq \epsilon$ & $LB_A \leq UB$
                $\sigma = \sigma \bigcup 1$;
                $\sigma^B = \sigma - 1$ ;
                UpdateTree($\sigma$)
            end if
        end if
    end if
```

Figure 6: Pseudo-code of UpdateTree

Step 1. Initialization: The initial upper bound, $UB$, is computed as the total flowtime of $\mathcal{J}^A$ for sequence $\sigma^B_{SPT} \cup \sigma^A_{SPT}$. Sequence $\sigma$ is initially the empty set ($\sigma :=$), sequence $\sigma^A$ contains all jobs in $\mathcal{J}^A$ ($\sigma^A := [0, \ldots, 0]$), and sequence $\sigma^B$ contains all jobs in $\mathcal{J}^B$ ($\sigma^B := [1, \ldots, 1]$).

Step 2. Update Tree: A recursive function is applied to construct the tree. More specifically, given a sequence $\sigma$, this procedure consists on:

- If $\sigma$ is complete, its flowtime $C^A_{sum}(\sigma)$ is computed. If $C^A_{sum}(\sigma)$ is lower than the current upper bound, then $UB := C^A_{sum}(\sigma)$.
- If $\sigma$ is a partial sequence, then the tree can be branched by adding one job from $\sigma_A$ (i.e. adding $\{0\}$ to $\sigma$) or one job from $\sigma_B$ (i.e. adding $\{1\}$ to $\sigma$). When the two options are available ($\sigma^A \neq$ and $\sigma^B \neq$), the branch with the addition of the job in $\sigma^A$ is prioritised. More specifically, we compute $LB^A$ the lower bound for $C^A_{sum}$ as $LB^A := C^A_{sum}(\sigma \bigcup \sigma^A)$; and $LB^B$ the lower bound for $C^B_{sum}$ as $LB^B := C^B_{sum}(\sigma \bigcup \sigma^B)$.
  - If $\sigma^A \neq$: If $LB^A < UB$ and $LB^B \leq \epsilon$, we add a job in $\mathcal{J}^A$ to $\sigma$, remove the job from $\sigma^A$, and invoke the Update Tree step. Otherwise, the branch is fathomed.
  - If $\sigma^B \neq$: If $LB^A < UB$ and $LB^B \leq \epsilon$, we add a job in $\mathcal{J}^B$ to $\sigma$, remove the job from $\sigma^B$, and invoke the Update Tree step. Otherwise, the branch is fathomed.

The pseudo-code of the Update Tree step is given in Figure 6.

```
procedure Maximum Improvement
$\sigma = \sigma^B_{SPT} \bigcup \sigma^A_{SPT}$;
$\delta = C^B_{sum}(\sigma) - \epsilon$;
while $\delta > 0$
$\sigma'$= Forward_Insertion_Improvement($\sigma$); % (see Property 4)
$\sigma''$= Backward_Insertion_Improvement($\sigma$); % (see Property 4)
    if $C^A_{sum}(\sigma') < C^A_{sum}(\sigma)$ & $C^A_{sum}(\sigma'') < C^A_{sum}(\sigma)$
        if $C^A_{sum}(\sigma') <= C^A_{sum}(\sigma'')$
            $\sigma = \sigma'$;
        else
            $\sigma = \sigma''$;
        end if
    else if  $C^A_{sum}(\sigma') < C^A_{sum}(\sigma)$ & $C^A_{sum}(\sigma'') = C^A_{sum}(\sigma)$
        $\sigma = \sigma'$;
    else
        $\sigma = \sigma''$;
    end if
$\delta = C^B_{sum}(\sigma) - \epsilon$;
end while
```

Figure 7: Pseudo-code of Maximum Improvement

## 5.2 Constructive heuristics

Given the NP-nature of the problem, it is unlikely that exact methods are practical for big instance sizes. For such cases, we develop several constructive heuristics that can provide approximate solutions in low CPU time. In addition, these fast heuristics could provide initial solutions for more sophisticated approximate methods, such as the GA presented in Section 5.3. In total, we present seven heuristics (denoted as CHX, X=1 to 7) which apply several improvement methods (based on the properties discussed in Section 4) to an initial sequence. The improvement methods are the following:

- Maximum Improvement: This method starts with the feasible sequence $\sigma^B_{SPT} \bigcup \sigma^A_{SPT}$ and performs a backward (forward) insertion on jobs in $\mathcal{J}^A$ ($\mathcal{J}^B$) using Property 4 (Property 4) in order to improve $C^A_{sum}$. The pseudo-code is given in Figure 7.
- Iterative Improvement: This method starts with any feasible sequence $\sigma$ and applies Corollary 4 to improve $C^A_{sum}(\sigma)$. The pseudo-code is given in Figure 8.
- Maximum Repair: This method starts with the sequence $\sigma^A_{SPT} \bigcup \sigma^B_{SPT}$ (infeasible unless the problem is trivial, see Section 2) and performs a backward (forward) insertion of jobs in $\mathcal{J}^B$ ($\mathcal{J}^A$) using Property 4 (Property 4) in order to obtain a feasible sequence by decreasing $C^B_{sum}$. The pseudo-code is given in Figure 9.
- Iterative Repair: This method starts with the sequence $\sigma^A_{SPT} \bigcup \sigma^B_{SPT}$ (infeasible unless the problem is trivial, see Section 2) and applies Corollary 4 in order to obtain a feasible sequence by decreasing $C^B_{sum}(\sigma)$. The pseudo-code is given in Figure 10.

These improvement methods are combined into the seven heuristics, as summarised in Table 2. CH1, CH2 and CH3 start with sequence $\sigma^B_{SPT} \bigcup \sigma^A_{SPT}$, which is feasible unless the problem

```
procedure Iterative Improvement($\sigma$)
$\delta = C^B_{sum}(\sigma) - \epsilon$;
while $\delta \geq p^A_1$
    $k = \max\{i \in \mathcal{J}^A : \delta \geq p^A_i\}$;
    Let $k'$ be the first job in the $A$-block where $k$ belongs to;
    Let $j$ be the last job of the adjacent $B$-block;
    Let $\sigma$ the sequence obtained by pairwise exchange of both jobs ($k'$ ajnd $j$); % (see Corollary 4)
    $\delta = C^B_{sum}(\sigma) - \epsilon$;
end while
```

Figure 8: Pseudo-code of Iterative Improvement

```
procedure Maximum Repair
$\sigma = \sigma^A_{SPT} \bigcup \sigma^B_{SPT}$;
$\delta = C^B_{sum}(\sigma) - \epsilon$;
while $\delta < 0$
$\sigma'$= Forward_Insertion_Unfeasibility_Repair($\sigma$); %(see Property 4)
$\sigma''$= Backward_Insertion_Unfeasibility_Repair($\sigma$); %(see Property 4)
    if $C^B_{sum}(\sigma') < C^B_{sum}(\sigma)$ & $C^B_{sum}(\sigma'') < C^B_{sum}(\sigma)$
        if $C^B_{sum}(\sigma') <= C^B_{sum}(\sigma'')$
            $\sigma = \sigma'$;
        else
            $\sigma = \sigma''$;
        end if
    else if  $C^B_{sum}(\sigma') < C^B_{sum}(\sigma)$ & $C^B_{sum}(\sigma'') = C^B_{sum}(\sigma)$;
        $\sigma = \sigma'$;
    else
        $\sigma = \sigma''$;
    end if
$\delta = C^B_{sum}(\sigma) - \epsilon$;
end while
```

Figure 9: Pseudo-code of Maximum Repair

```
procedure Iterative Repair($\sigma$)
$\delta = C^B_{sum}(\sigma) - \epsilon$;
while $\delta < 0$
    if $|\delta| \leq p^A_{n^A}$
        $k = \min\{i \in \mathcal{J}^A : |\delta| \leq p^A_i\}$;
    else
        $k = n^A$;
    Let $k'$ be the last job in the $A$-block where $k$ belongs to;
    Let $j$ be the first job of the adjacent $B$-block;
    Let $\sigma$ the sequence obtained by pairwise exchange of both jobs ($k'$ and $j$); % (see Corollary 4)
    $\delta = C^B_{sum}(\sigma) - \epsilon$;
end while
```

Figure 10: Pseudo-code of Iterative Repair

Table 2: Constructive Heuristics

| | CH1 | CH2 | CH3 | CH4 | CH5 | CH6 | CH7 |
|---|---|---|---|---|---|---|---|
| Initial Sequence | $\sigma^B_{SPT} \bigcup \sigma^A_{SPT}$ | | | $\sigma^A_{SPT} \bigcup \sigma^B_{SPT}$ | | | $\sigma_{SPT}$ |
| Maximum Improvement | x | | x (1st) | | | | |
| Iterative Improvement | | x | x (2nd) | | | x (2nd) | x |
| Maximum Repair | | | | x | | x (1st) | |
| Iterative Repair | | | | | x | | x |

does not have any feasible solution. This schedule provides the worst value of $C^A_{sum}$. CH1, CH2 and CH3 try to improve the flowtime while maintaining the feasibility. More specifically, CH1 applies Maximum Improvement, CH2 applies Iterative Improvement, and CH3 first applies Maximum Improvement and then Iterative Improvement. CH5, CH6 and CH7 start with the initial sequence $\sigma^A_{SPT} \bigcup \sigma^B_{SPT}$, which is infeasible (unless the problem is trivial). This schedule provides the best value of $C^A_{sum}$. CH5, CH6 and CH7 look for feasibility trying to provide a good value of $C^A_{sum}$. More specifically, CH4 applies Maximum Repair and CH5 applies Iterative Repair until feasibility is achieved. CH6 applies Maximum Repair until feasibility is achieved and then, Iterative Improvement tries to improve $C^A_{sum}$. Finally, CH7 starts with the initial sequence $\sigma_{SPT}$, i.e. all jobs are scheduled by the SPT rule. This schedule can be feasible or infeasible. If it is feasible, the Iterative Improvement method is applied, else the Iterative Repair method is applied.

## 5.3 Genetic Algorithm

Genetic Algorithms (GAs) have been frequently used in scheduling with very good results (see e.g. Chang et al., 2007; Luo et al., 2009; Damodaran et al., 2009; Ventura and Yoon, 2013; Li et al., 2014; Chang and Liu, 2015 for single criterion problems, Framinan, 2009; Tseng and Lin, 2010; Yao et al., 2011 for multicriteria problems, and Balasubramanian et al., 2009; Soltani et al., 2010; Li and Hsu, 2012 for interfering jobs problems). In the version presented by Ruiz and Allahverdi (2009), there is one population and new individuals do not replace their parents, but a new individual replaces the worst individual of the population if this new one is unique and better than the worst one. This GA was successfully employed by Ruiz et al. (2005); Ruiz and Maroto (2006); Ruiz and Allahverdi (2009). We have adapted this GA to our problem (see Figure 11). However, taking into account the original version presented by Ruiz and Allahverdi (2009), we have replaced the local search procedure by an improvement method based on Path Relinking.

This scheme of GA has been implemented using both codifications presented in the previous section: permutation and binary codification –denoted GA(P) and GA(B) respectively–. Both algorithms are identical, except with respect to the fitness computation, since the way to calculate the flowtime is simpler in the binary codification than in the permutation codification. Moreover, we have implemented an improved version of GA(B) –denoted GA(B)_I– which differs from the former in two phases: Initial population and Improvement method. Details about the phases of the proposed GA (Initial population, fitness calculation, parents selection, crossover operator, improvement method, mutation scheme, and the diversity of the population) are described below. Finally, as the algorithm does not have a natural stopping criterion, it is stopped when a given amount of CPU time has elapsed (see Section 6).

### 5.3.1 Initial population

The population size is given by the parameter *populationsize*. This population contains two super-individuals generated in the first step. For GA(P) and GA(B), two solutions using heuristics

```
procedure GA
    % STEP 1:  Initial Population
    $S^1$ := $CH1$;
    $S^2$ := $CH2$;
    insert $S^1$ and $S^2$ into population;
    for $k = 3$ to $size_{population}$ do
        $S^k$ := randomly generated sequence;
        insert $S^k$ into population;
    end for
    for $k = 1$ to $size_{population}$ do
        $F^k$ := fitness($S^k$);
    end for
    % STEP 2:  New Population
    while (termination criterion not satisfied) do
        for $k = 1$ to 2 do
            select $S^k$ by n_tournament(population);
        end for
        TP_Crossing($S^1$,$S^2$);
        Path_Relinking($S^1$,$S^2$, $S_{new}$);
        Shift_Mutation($S_{new}$);
        $F^{new}$ := fitness($S_{new}$);
        $uniqueness$= Uniqueness($S_{new}$, $population$);
        if($uniqueness = true$) then
            $F^{worst}$ := fitness($S^{worst}$);
            if($F^{new} < F^{worst}$) then
                remove $S^{worst}$ from population;
                insert $S^{new}$ in population;
            end if
        end if
    end while
end
```

Figure 11: Pseudo-code of GA

CH1 and CH2 are employed as super-individuals, whereas for GA(B)_I four super-individuals are generated using the best four constructive heuristics (see Section 6), i.e. CH2, CH3, CH5 and CH7.

### 5.3.2 Fitness calculation

In order to obtain better results, infeasible solutions in the population are allowed. Then, the fitness of sequence $\sigma$ is calculated depending on its feasibility. Since, for a given $\epsilon$, $\sigma$ is feasible if $C^B_{sum}(\sigma) \leq \epsilon$ (and infeasible otherwise), the infeasibility is penalized depending on the difference between $C^B_{sum}$ and $\epsilon$. Therefore, the fitness of a sequence $\sigma$ is given by:

$$fitness(\sigma) = \begin{cases} C^A_{sum}(\sigma) & \textit{if } \sigma \textit{ is feasible} \\ C^A_{sum}(\sigma) + C^B_{sum}(\sigma) - \epsilon & \textit{if } \sigma \textit{ is infeasible} \end{cases}$$

### 5.3.3 Parents selection

For parents selection, the $n$-tournament selection procedure by Ruiz and Allahverdi (2007) is chosen, i.e.:

Step 1: Select randomly a percentage of the population according to the pressure parameter (denoted as $pressure$).

Step 2: Select the individual with the best fitness value among the randomly chosen individuals as a parent

Step 3: If there are two parents then Stop. Else, remove the first parent from the population and return to Step 2

In this procedure, the first parent selected does not participate in the tournament for the selection of the second parent.

### 5.3.4 Crossover operator

The crossover procedure selected is the two-point (TP) crossing. Two-point crossover selects two crossing sites and chromosomal material is swapped between them. Let $S^1 = (S^1_1, \ldots, S^1_n)$ and $S^2 = (S^2_1, \ldots, S^2_n)$ be two parents, and $k$ and $l$ ($k < l$) the two points (i.e. two positions of vectors $S^1$ and $S_2$). Then the resulting offsprings have the same structure than their parents from job 1 to $k$ and from job $l$ to $n$, and the jobs from $k$ to $l$ swapped from $S^1$ to $S^2$. Applying this method to a schedule may result in some jobs repeated in the offspring and therefore infeasible sequences. This is avoided by the process explained in Figure 12. The probability of carrying out a crossover after selection is called $p_C$.

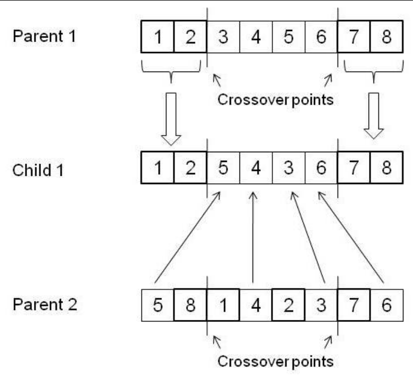

Figure 12: TP crossing

### 5.3.5 Mutation scheme

The mutation operator is based on a probability ($p_M$) per individual to be mutated. The mutation process is as follows:

Step 1: Extract one job from the individual

Step 2: Re-insert it in another random position

### 5.3.6 Diversity in the population

Finally, each new individual is checked to guarantee its 'uniqueness' once the fitness has been calculated in order to avoid clones in the population, following this two-level process:

- At objective function level: two identical individuals cannot have distinct objective function values, but two distinct solutions might have the same objective function value. Therefore, whenever an equal objective function value is found in the population, both solutions (the existing and the new one) are selected for the next level.
- At permutation level: check if the selected individuals in the previous level have the same exact permutation. In this case, the new individual is not introduced in the population.

### 5.3.7 Improvement Method

The GA proposed incorporates an improvement method based on Path Relinking (see e.g. Pinedo, 1995; Bozejko, 2010) applied to the pairs of individuals, denoted as $S^1$ and $S^2$, obtained by the $n$-tournament procedure, or to both offsprings obtained after crossover if it has been carried out. The procedure is the following:

Step 1: Let $S$ be the best sequence between $S^1$ and $S^2$

Step 2: Compare each component of $S^1$ to the same component of $S^2$, if they are different remove it and insert it at the end of $S^1$.

Step 3: If the sequence obtained for each comparison is better than $S$, then replace $S$ by this new sequence.

The new sequence obtained is denoted as $S^{new}$ in the pseudo-code in Figure 11. Moreover, GA(B)_I incorporates the Iterative Improvement method (see Figure 8) and Iterative Repair method (see Figure 10) applied to $S^1$ and $S^2$ depending on their feasibility. Both methods are applied before the Path Relinking method.

## 6 Computational experiments

Taking into account the conclusions regarding the empirical hardness of the problem derived in Section 3, we have designed a testbed in Section 6.1. This test bed allows us to compare the proposed B&B with the DP in order to test their efficiency (see Section 6.2). Additionally, we compare the best exact method (B&B) with the constructive heuristics (see Section 6.3). Finally, we calibrate the GA by a Design of Experiments, and compare GA(P), GA(B) and GA(B)_I to the best solution provided by the four best constructive heuristics (see Section 6.4).

### 6.1 Test bed design

In the analysis of the distribution of solutions carried out in Section 3, we concluded that the empirically hardest problems are those with a large value of $\epsilon$ and $n^A < n^B$. In order to test the different methods for a wide variety of problems, we design three test beds with different sizes and difficulty. Processing times are uniformly generated in the interval $[1, 99]$. $\epsilon$ is computed as in Section 3, depending on $\alpha$. All test beds contain 10 independent instances for each combination of jobs. More specifically, the test beds are as follows:

- Small Size Medium Difficulty (SSMD) test bed: In this case we consider different combination of sizes $n^A \times n^B$, with $n^A > n^B$, $n^A = n^B$ and $n^A < n^B$. In addition, $\alpha$ values would be uniformly generated in the interval $[0.4, 0.6]$. According to the results from Section 3, these problems cannot be considered very difficult. The values selected for $n^A \times n^B$ are:

| | | | |
|---|---|---|---|
| $5 \times 5$ | $10 \times 5$ | $15 \times 5$ | $20 \times 5$ |
| $5 \times 10$ | $10 \times 10$ | $15 \times 10$ | $20 \times 10$ |
| $5 \times 15$ | $10 \times 15$ | $15 \times 15$ | $20 \times 15$ |
| $5 \times 20$ | $10 \times 20$ | $15 \times 20$ | $20 \times 20$ |

  In total, we have 160 instances in the SSMD test bed.

- Small Size High Difficulty (SSHD) test bed: According to the results from Section 3 in this case we consider $n^A < n^B$ and $\alpha$ uniformly generated in the interval $[0.5, 0.8]$. Values selected for $n^A \times n^B$ are:

| | | | |
|---|---|---|---|
| $5 \times 10$ | $10 \times 15$ | $15 \times 20$ | $20 \times 25$ |
| $5 \times 15$ | $10 \times 20$ | $15 \times 25$ | $20 \times 30$ |
| $5 \times 20$ | $10 \times 25$ | $15 \times 30$ | |
| $5 \times 25$ | $10 \times 30$ | | |
| $5 \times 30$ | | | |

In total, we have 140 instances in the SSHD test bed.

- Big Size High Difficulty (BSHD) test bed: According to the results from Section 3 in this case we consider $n^A \leq n^B$ and $\alpha$ uniformly generated in the interval $[0.5, 0.6]$. Values selected for $n^A \times n^B$ are:

| | | | |
|---|---|---|---|
| $20 \times 20$ | $50 \times 50$ | $100 \times 100$ | $200 \times 200$ |
| $20 \times 50$ | $50 \times 80$ | $100 \times 200$ | $200 \times 500$ |
| $20 \times 80$ | $50 \times 100$ | $100 \times 500$ | $500 \times 500$ |

In total, we have 120 instances in the BSHD test bed.

### 6.2 Comparison of B&B and DP

Table 3 and Table 4 show the results for DP and B&B in SSMD and SSHD test beds respectively. Both methods are stopped when the computational time is 3,600 seconds. For some problem sizes, the methods do not obtain the optimal solution in all instances. Then, we compute for each instance the relative percentage deviation ($RPD$) as follows:

$$RPD = \frac{C^A_{sum}(\text{ALG}) - C^A_{sum}(MIN)}{C^A_{sum}(MIN)} \cdot 100$$

where $C^A_{sum}(\text{ALG})$ is the value of the objective function provided by the algorithm considered, and $C^A_{sum}(MIN)$ is the value of the flowtime provided by the minimum value of all algorithms (optimum or best bound).

Table 3 and Table 4 show the number of instances solved optimally (denoted as $N$), the average computational times (in seconds) required by each method, and average $RPD$ ($ARPD$) values for each size of the SSMD test bed and SSHD test bed respectively solved by B&B and DP. In particular, Table 3 presents the results for the SSMD test bed. It can be seen that the CPU times required by DP are higher than those of B&B. DP cannot provide the optimal solution in less than 3,600 seconds for all instances of size $20 \times 20$. For these instances, the best solution provided by DP in one hour is 32% worst in average than the optimal value. Table 4 presents the results for the SSHD test bed. B&B is faster than DP for all instances. Biggest size instances ($15 \times 25$, $15 \times 30$, $20 \times 25$ and $20 \times 30$) are not solved optimally by DP in less than 3,600 seconds, while the number of instances solved optimally by B&B are: all instances (10 of 10) for $15 \times 25$ and $15 \times 30$ in only 41.77 and 29.47 seconds on average respectively; 9 of 10 for $20 \times 25$; and 4 of 10 for $20 \times 30$ in less than 3,600 seconds. Moreover, $ARPD$ values for the solutions provided by

Table 3: Number of instances solved optimally $N$, average CPU times (secs.) and $ARPD$: Exact Methods applied to the Small Size Medium Difficulty test bed. * Optimum are not obtained for all instances.

| | $N$ | | Average CPU Times | | $ARPD$ | |
|---|---|---|---|---|---|---|
| $n^A \times n^B$ | B&B | DP | B&B | DP | B&B | DP |
| $5 \times 5$ | 10 | 10 | 0.0002 | 0.0006 | 0.0000 | 0.0000 |
| $5 \times 10$ | 10 | 10 | 0.0006 | 0.0017 | 0.0000 | 0.0000 |
| $5 \times 15$ | 10 | 10 | 0.0004 | 0.0061 | 0.0000 | 0.0000 |
| $5 \times 20$ | 10 | 10 | 0.0008 | 0.0225 | 0.0000 | 0.0000 |
| $10 \times 5$ | 10 | 10 | 0.0004 | 0.0041 | 0.0000 | 0.0000 |
| $10 \times 10$ | 10 | 10 | 0.0013 | 0.0870 | 0.0000 | 0.0000 |
| $10 \times 15$ | 10 | 10 | 0.0203 | 1.4639 | 0.0000 | 0.0000 |
| $10 \times 20$ | 10 | 10 | 0.1270 | 13.4427 | 0.0000 | 0.0000 |
| $15 \times 5$ | 10 | 10 | 0.0006 | 0.0117 | 0.0000 | 0.0000 |
| $15 \times 10$ | 10 | 10 | 0.0144 | 1.8762 | 0.0000 | 0.0000 |
| $15 \times 15$ | 10 | 10 | 0.3206 | 82.4253 | 0.0000 | 0.0000 |
| $15 \times 20$ | 10 | 10 | 4.2169 | 1661.1820 | 0.0000 | 0.0000 |
| $20 \times 5$ | 10 | 10 | 0.0009 | 0.0461 | 0.0000 | 0.0000 |
| $20 \times 10$ | 10 | 10 | 0.0907 | 20.5339 | 0.0000 | 0.0000 |
| $20 \times 15$ | 10 | 10 | 3.1844 | 1999.3910 | 0.0000 | 0.0000 |
| $20 \times 20$ | 10 | 0 | 133.0693 | $> 3{,}600$ | 0.0000 | 32.1909* |
| Aver. | 10.00 | 9.38 | 8.8156 | 461.2809 | 0.0000 | 2.0119 |

DP in these cases are high, which means that the approximate solutions provided by the B&B in one hour are better than approximate solutions provided by DP in the same time.

Therefore, we can conclude that the proposed B&B algorithm provides optimal solutions faster than the DP. The differences between each method are remarkable, from less than 9 seconds to more than 460 seconds on average for the SSMD test bed, and from less than 266 seconds to more than 1,198 seconds on average for the SSHD test bed. Moreover, on average, the biggest instances are solved in reasonable CPU times by the proposed algorithm while the DP cannot obtain the optimal solution after one hour of computation.

## 6.3 Comparison of constructive heuristics

The constructive heuristics proposed in Section 5.2, CH1 to CH7, have been tested for small sizes test beds. These heuristics do not give the optimal solutions so they are compared to the (optimal or best bound) solution provided by the B&B in less than 3,600 seconds.

Table 5 shows $ARPD$ values for each size of the SSMD test bed for the B&B, each constructive heuristic, and the minimum value among the four best methods (CH2, CH3, CH5 and CH7), denoted as MIN4 (it is computed for each instance and the average for each size). The best bound provided by B&B for these biggest instances in less than 3,600 seconds are worst than the approximate solutions provided by the constructive heuristics. It can be observed that the best constructive heuristics are CH3 and CH7. MIN4 obtains, on average, a flowtime which is around 3% worst than the optimum in negligible computation times. Additionally, the variability

Table 4: Number of instances solved optimally $N$, average CPU times (secs.) and $ARPD$: Exact Methods applied to the Small Size High Difficulty test bed. * Optimum are not obtained for all instances.

| | $N$ | | Average CPU Times | | ARPD | |
|---|---|---|---|---|---|---|
| $n^A \times n^B$ | B&B | DP | B&B | DP | B&B | DP |
| $5 \times 10$ | 10 | 10 | 0.0004 | 0.0018 | 0.0000 | 0.0000 |
| $5 \times 15$ | 10 | 10 | 0.0004 | 0.0063 | 0.0000 | 0.0000 |
| $5 \times 20$ | 10 | 10 | 0.0008 | 0.0228 | 0.0000 | 0.0000 |
| $5 \times 25$ | 10 | 10 | 0.0019 | 0.0646 | 0.0000 | 0.0000 |
| $5 \times 30$ | 10 | 10 | 0.0063 | 0.1398 | 0.0000 | 0.0000 |
| $10 \times 15$ | 10 | 10 | 0.0108 | 1.6415 | 0.0000 | 0.0000 |
| $10 \times 20$ | 10 | 10 | 0.1296 | 14.6060 | 0.0000 | 0.0000 |
| $10 \times 25$ | 10 | 10 | 0.6327 | 90.4479 | 0.0000 | 0.0000 |
| $10 \times 30$ | 10 | 10 | 2.4954 | 424.4971 | 0.0000 | 0.0000 |
| $15 \times 20$ | 10 | 10 | 4.1286 | 1841.0160 | 0.0000 | 0.0000 |
| $15 \times 25$ | 10 | 0 | 41.7778 | $> 3{,}600$ | 0.0000 | 35.7003* |
| $15 \times 30$ | 10 | 0 | 29.4793 | $> 3{,}600$ | 0.0000 | 71.3482* |
| $20 \times 25$ | 9 | 0 | 1151.0255 | $> 3{,}600$ | 0.0000* | 84.9178* |
| $20 \times 30$ | 4 | 0 | 2488.0908 | $> 3{,}600$ | 0.0000* | 101.3890* |
| Aver. | 9.50 | 7.14 | 265.5557 | 1198.0317 | 0.0000 | 20.9539 |

among sizes for these heuristics is low (see the standard deviation reported), which means that heuristics are robust with respect to the problem size. $ARPD$ values increase with $n^B$, and, considering instances with the same number of total jobs ($n^A + n^B$), it can be observed that the cases with $n^A < n^B$ have worse results than those where the opposite occurs. This may be seen as a confirmation of the results obtained in Section 3.

Table 6 shows the corresponding results for each size of the SSHD test bed. As it can be seen, $ARPD$ values are worse than those in Table 5. In general, all methods perform worse for this test bed, which is in line with the conclusions of the analysis on the distribution of solutions carried out in Section 3. Again, CH3 and CH7 are the best methods with the lowest values of average and standard deviation. Results show that MIN4 provide solutions which are, on average, at 6.63% from the best solution. Given the difficulty of the instances, these methods provide a nearly instantaneous good solution to the problem.

## 6.4 Calibration and comparison of GA

The GA versions proposed in Section 5 have the following parameters: Population size ($populationsize$) which determines the size of the population created and used in the GA; Pressure ($pressure$), which provides the percentage of population involved in the $n$-tournament selection procedure; Probability of crossing $p_C$; Probability of mutation $p_{mut}$; and finally the stopping criteria $time$, which determines the CPU time provided to the GA to solve the problem ($\frac{n \cdot time}{1000}$ seconds, with $n = n^A + n^B$).

Based on the levels tested for each parameter (factor) used by Ruiz and Allahverdi (2009); Ruiz and Maroto (2006); Ruiz et al. (2005), we have selected the following: $populationsize \in$

Table 5: Constructive Heuristics for Small Size Medium Difficulty test bed

| | | | | | *ARPD* | | | | | CPU Times | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $n^A \times n^B$ | B&B | CH1 | CH2 | CH3 | CH4 | CH5 | CH6 | CH7 | MIN4 | B&B | MIN4 |
| $5 \times 5$ | 0.0000 | 8.4918 | 3.2877 | 1.7596 | 20.2461 | 9.3539 | 10.4636 | 3.0455 | 0.4123 | 0.0002 | 0.0061 |
| $5 \times 10$ | 0.0000 | 18.7286 | 15.2634 | 7.4265 | 29.4643 | 16.3847 | 17.5867 | 7.4265 | 3.2740 | 0.0006 | 0.0065 |
| $5 \times 15$ | 0.0000 | 24.5432 | 22.0990 | 4.1013 | 37.6441 | 23.4061 | 26.7973 | 4.1013 | 4.1013 | 0.0004 | 0.0069 |
| $5 \times 20$ | 0.0000 | 28.6363 | 28.2512 | 9.8982 | 81.5602 | 31.7349 | 30.2301 | 9.8982 | 8.4770 | 0.0008 | 0.0064 |
| $10 \times 5$ | 0.0000 | 4.4782 | 3.5412 | 3.4344 | 7.0223 | 2.7761 | 6.5191 | 3.2261 | 1.5946 | 0.0004 | 0.0062 |
| $10 \times 10$ | 0.0000 | 3.9960 | 2.7143 | 3.0000 | 13.0221 | 5.2794 | 15.3995 | 3.6407 | 1.7722 | 0.0013 | 0.0066 |
| $10 \times 15$ | 0.0000 | 9.5574 | 7.4764 | 9.5234 | 15.3379 | 8.0826 | 20.3262 | 9.5234 | 6.1673 | 0.0203 | 0.0072 |
| $10 \times 20$ | 0.0000 | 21.2348 | 19.9943 | 7.6330 | 16.1047 | 8.3904 | 25.7940 | 7.6330 | 5.4378 | 0.1270 | 0.007 |
| $15 \times 5$ | 0.0000 | 2.8318 | 2.1298 | 2.0413 | 6.0639 | 1.4715 | 5.0153 | 1.8876 | 1.0571 | 0.0006 | 0.0064 |
| $15 \times 10$ | 0.0000 | 5.4673 | 4.2801 | 4.5909 | 10.2436 | 4.1900 | 12.2794 | 4.3187 | 3.0591 | 0.0144 | 0.0068 |
| $15 \times 15$ | 0.0000 | 4.8215 | 3.4695 | 6.1755 | 13.5036 | 6.3630 | 17.9188 | 3.8728 | 2.8235 | 0.3206 | 0.0076 |
| $15 \times 20$ | 0.0000 | 5.9754 | 4.6898 | 5.8526 | 13.3377 | 6.1640 | 21.8438 | 5.7325 | 4.3777 | 4.2169 | 0.0072 |
| $20 \times 5$ | 0.0000 | 1.5720 | 1.1932 | 1.0334 | 4.0221 | 1.3050 | 6.3397 | 1.0422 | 0.5038 | 0.0009 | 0.0071 |
| $20 \times 10$ | 0.0000 | 3.8852 | 3.4352 | 3.3044 | 6.2721 | 2.2270 | 9.2054 | 2.4254 | 1.5753 | 0.0907 | 0.0072 |
| $20 \times 15$ | 0.0000 | 3.2943 | 2.3571 | 3.8404 | 8.2567 | 3.5420 | 14.0641 | 3.1864 | 1.9390 | 3.1844 | 0.0061 |
| $20 \times 20$ | 0.0000 | 5.3966 | 4.7062 | 6.1786 | 9.8100 | 5.4109 | 18.6657 | 2.5146 | 2.4896 | 133.0693 | 0.0064 |
| Aver. | 0.0000 | 9.5569 | 8.0555 | 4.9871 | 18.2445 | 8.5051 | 16.1530 | 4.5922 | 3.0663 | 8.8156 | 0.0067 |
| Std. Dev. | 0.0000 | 8.6279 | 8.4277 | 2.6759 | 19.0732 | 8.4519 | 7.6570 | 2.6827 | 2.2019 | 33.1583 | 0.0005 |

Table 6: Constructive Heuristics for Small Size High Difficulty test bed

| | | | | | *ARPD* | | | | | CPU Times | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $n^A \times n^B$ | B&B | CH1 | CH2 | CH3 | CH4 | CH5 | CH6 | CH7 | MIN4 | B&B | MIN4 |
| $5 \times 10$ | 0.0000 | 21.8319 | 20.9624 | 4.0354 | 37.7012 | 22.8314 | 20.6174 | 4.0354 | 2.6322 | 0.0004 | 0.0069 |
| $5 \times 15$ | 0.0000 | 27.3264 | 26.3915 | 8.3615 | 52.5849 | 21.8366 | 21.0223 | 8.3615 | 7.5119 | 0.0004 | 0.0067 |
| $5 \times 20$ | 0.0000 | 32.3473 | 30.8796 | 8.0291 | 58.3927 | 33.3875 | 31.5060 | 8.0291 | 6.4734 | 0.0008 | 0.0067 |
| $5 \times 25$ | 0.0000 | 31.8541 | 29.2141 | 12.2470 | 81.9308 | 36.2388 | 36.0655 | 12.2470 | 10.6151 | 0.0019 | 0.0065 |
| $5 \times 30$ | 0.0000 | 33.7142 | 31.8267 | 11.4447 | 45.8394 | 33.7054 | 35.1947 | 11.4447 | 9.9047 | 0.0063 | 0.0067 |
| $10 \times 15$ | 0.0000 | 5.7016 | 4.4005 | 5.2098 | 17.0224 | 9.1380 | 21.9361 | 5.2098 | 4.3259 | 0.0108 | 0.0072 |
| $10 \times 20$ | 0.0000 | 16.2264 | 12.8269 | 12.0889 | 18.8685 | 11.0054 | 21.8328 | 12.0889 | 6.3250 | 0.1296 | 0.0064 |
| $10 \times 25$ | 0.0000 | 23.1331 | 21.1691 | 8.1807 | 19.8593 | 13.8358 | 28.2659 | 8.1807 | 7.4113 | 0.6327 | 0.0073 |
| $10 \times 30$ | 0.0000 | 30.8885 | 30.1709 | 12.2075 | 24.0247 | 16.2861 | 35.5662 | 12.2075 | 9.2334 | 2.4954 | 0.0073 |
| $15 \times 20$ | 0.0000 | 6.9520 | 5.6825 | 8.3836 | 14.5568 | 10.2456 | 21.8648 | 8.3543 | 5.6605 | 4.1286 | 0.0069 |
| $15 \times 25$ | 0.0000 | 10.1093 | 8.6825 | 8.7632 | 15.6656 | 11.2731 | 25.3717 | 8.7632 | 6.4082 | 41.7778 | 0.0072 |
| $15 \times 30$ | 0.0000 | 20.6720 | 19.5952 | 6.8067 | 17.5758 | 10.6801 | 30.9389 | 6.8067 | 5.8180 | 29.4793 | 0.0075 |
| $20 \times 25$ | 0.0000 | 7.7050 | 6.4602 | 9.2131 | 11.8188 | 8.5276 | 23.0170 | 8.7296 | 5.6773 | 1151.0255 | 0.0078 |
| $20 \times 30$ | 0.0000 | 6.5043 | 4.8900 | 7.3538 | 12.0022 | 7.4508 | 26.6582 | 7.3538 | 4.8732 | 2488.0908 | 0.0082 |
| Aver. | 0.0000 | 19.6404 | 18.0823 | 8.7375 | 30.5602 | 17.6030 | 27.1327 | 8.7009 | 6.6336 | 265.5557 | 0.0071 |
| Std. Dev. | 0.0000 | 10.6885 | 10.6624 | 2.5483 | 21.4879 | 10.2269 | 5.7760 | 2.5449 | 2.1807 | 708.7810 | 0.0005 |

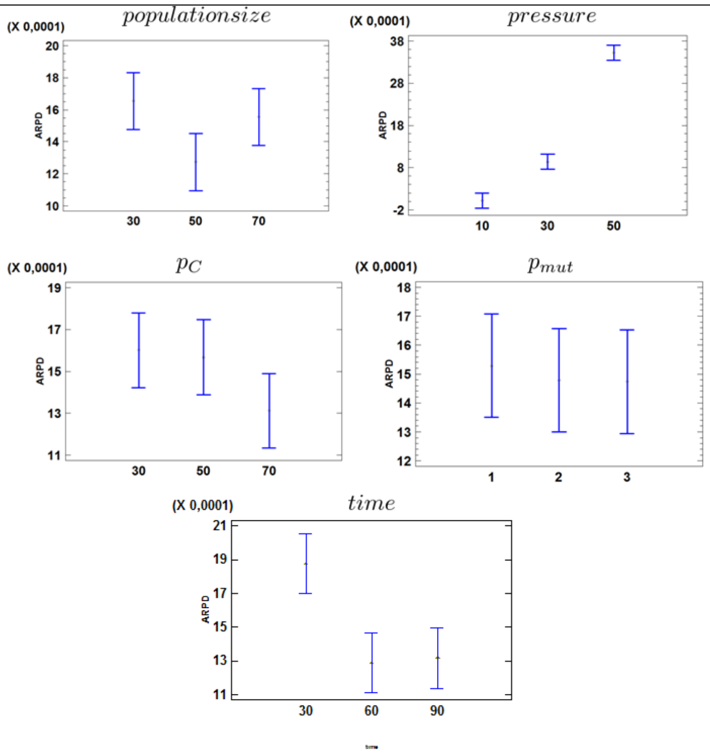

Figure 13: Mean and LSD Intervals 95% for *populationsize*, *pressure*, $p_C$, $p_{mut}$ and *time*

$\{30, 50, 70\}$, $pressure \in \{10, 30, 50\}$, $p_C \in \{30, 50, 70\}$, $p_{mut} \in \{1, 2, 3\}$ and $time \in \{30, 60, 90\}$, and we have developed a Design of Experiments, using the SSMD test bed in order to obtain the best combination of the parameters. The Analysis of Variance (ANOVA) applied to the $ARPD$ values obtained by the GAB_I reveals that only the parameters *pressure* and *time* have statistical influence on the variable $ARPD$ with 95% confidence level. This result indicates that the levels selected for the parameters *popsize*, $p_C$ and $p_{mut}$ do not greatly affect the $ARPD$. Figure 13 shows the LSD intervals for the $ARPD$ in order to determine the best value for each parameter. For *popsize*, there are not significant differences between 30 and 60, but these exist between 60 and 90, being 60 the best option. There are significant differences among all levels of *pressure*, increasing the $ARPD$ values as it grows from 10 to 50. The probability of crossing $p_C$ and mutation $p_{mut}$ does not provide significant differences among levels, but the lowest means are obtained for $p_C = 70$ and $p_{mut} = 2$. Finally, the best option for parameter *time* is provided by 60, since there are not significant differences between 60 and 90. From the study, we conclude that the best combination of parameters for the GA are $popsize = 50$, $pressure = 10$, $p_C = 70$, $p_{mut} = 2$ and $time = 60$.

Once the calibration has been carried out, we compare GA with MIN4 (the combination of the four best constructive heuristics), in order to determine its efficiency for the high difficulty test beds (SSHD and BSHD). We solve the instances using GA(P), GA(B) and GA(B)_I. Table 7 shows the results for the SSHD test bed. It can be observed that GA(B)_I provides the

Table 7: Comparison of GA to MIN4 for Small Size High Difficulty test bed

| | *ARPD* | | | |
|---|---|---|---|---|
| $n^A \times n^B$ | MIN4 | GA(P) | GA(B) | GA(B)_I |
| $5 \times 10$ | 2,6322 | 11,0456 | 0,0000 | 0,0000 |
| $5 \times 15$ | 7,5119 | 14,0179 | 0,0000 | 0,0000 |
| $5 \times 20$ | 6,4734 | 21,3967 | 0,0000 | 0,0000 |
| $5 \times 25$ | 10,6151 | 25,3148 | 0,0000 | 0,0000 |
| $5 \times 30$ | 9,9047 | 23,0635 | 0,0000 | 0,0000 |
| $10 \times 15$ | 4,3259 | 14,7907 | 0,0000 | 0,0000 |
| $10 \times 20$ | 6,3250 | 15,4136 | 0,1088 | 0,0959 |
| $10 \times 25$ | 7,4113 | 17,3734 | 0,0000 | 0,0000 |
| $10 \times 30$ | 9,2334 | 22,4824 | 0,0142 | 0,0000 |
| $15 \times 20$ | 5,6605 | 12,1942 | 0,0055 | 0,0000 |
| $15 \times 25$ | 6,4082 | 17,1291 | 0,0177 | 0,0000 |
| $15 \times 30$ | 5,8180 | 24,1633 | 0,0287 | 0,0000 |
| $20 \times 25$ | 5,6746 | 13,9534 | 0,0333 | 0,0000 |
| $20 \times 30$ | 5,3589 | 19,3639 | 0,0292 | 0,0000 |
| Aver. | 6,6681 | 17,9788 | 0,0170 | 0,0068 |
| Std. Dev. | 2,1544 | 4,6662 | 0,0293 | 0,0256 |

best results for all instances (except one). The permutation codification is not efficient for this problem, as the difference between GA(P) and GA(B) is notable. Regarding CPU times, we have reported the same times provided in Table 6 for MIN4, together with the CPU times of all versions of the GA, which have been stopped at $n \cdot 60/1000$ seconds, with $n = n^A + n^B$. Finally, Table 8 shows the results for the BSHD test bed, being GA(B)_I the best method. The relatively good performance of the constructive heuristics can be noted as well, which confirms that it may be interesting to apply local search to improve the GA.

## 7 Conclusions

In this paper, we have addressed a single machine interfering jobs scheduling problem with objective total flowtime for two sets of jobs, denoted $1||\epsilon(C^A_{sum}/C^B_{sum})$. This problem is known to be weakly NP-hard, and a Dynamic Programming algorithm exists to solve it. However, the computational performance of this algorithm was unknown, and, given the NP nature of the problem, there were no methods to provide approximate solutions to the problem in short time intervals.

Taking into account a property of the problem, we propose a codification scheme, called binary encoding, which reduces the space of solutions of our problem as compared to the classical permutation encoding. The distribution of solutions is analysed depending on the instance size ($n^A$ and $n^B$) and $\epsilon$. We study the empirical hardness of the problem with respect to its feasibility and quality of the solutions by analysing the percentage of feasible solutions and its relative distances to the optimal solution respectively. As a result, the problem turns out to be harder (in statistical terms) for large values of $\epsilon$, and $n^A < n^B$. This analysis help us to derive conclusions about the importance to provide properties for the neighbourhood structure (in terms of feasibility

Table 8: Comparison of GA and MIN4 for Big Size High Difficulty test bed

| | *ARPD* | | | |
|---|---|---|---|---|
| $n^A \times n^B$ | MIN4 | GA(P) | GA(B) | GA(B)_I |
| $20 \times 20$ | 0.0302 | 0.0848 | 0.0028 | 0.0000 |
| $20 \times 50$ | 0.1655 | 0.1367 | 0.0339 | 0.0000 |
| $20 \times 80$ | 0.1874 | 0.1393 | 0.0792 | 0.0000 |
| $50 \times 50$ | 0.0265 | 0.0686 | 0.0024 | 0.0000 |
| $50 \times 80$ | 0.1134 | 0.0854 | 0.0039 | 0.0000 |
| $50 \times 100$ | 0.1426 | 0.1013 | 0.0410 | 0.0000 |
| $100 \times 100$ | 0.0182 | 0.0700 | 0.0013 | 0.0000 |
| $100 \times 200$ | 0.1391 | 0.1006 | 0.0540 | 0.0000 |
| $100 \times 500$ | 0.2389 | 0.1393 | 0.0901 | 0.0000 |
| $200 \times 200$ | 0.0192 | 0.0780 | 0.0023 | 0.0000 |
| $200 \times 500$ | 0.1882 | 0.1040 | 0.0695 | 0.0000 |
| $500 \times 500$ | 0.0151 | 0.1160 | 0.0056 | 0.0000 |
| Aver. | 0.1070 | 0.1020 | 0.0322 | 0.0000 |
| Std. Dev. | 0.0813 | 0.0261 | 0.0339 | 0.0000 |

and quality) to be embedded on heuristics. Moreover, the analysis also allows us to determine the values of the parameters of the problem that make the instances to be easy or hard, in order to generate appropriate test beds.

We have proposed both exact and approximate solution procedures using the binary codification and the aforementioned properties. Regarding exact procedure, a branch and bound algorithm has been developed. Regarding approximate procedures, different methods to improve feasible schedules and repair infeasible schedules have been designed. These methods have been embedded in seven constructive heuristics labelled CH1 to CH7. Finally, three versions of a Genetic Algorithm have been developed. All of them follow the same phases without using sophisticated local search methods. GA(P) uses permutation encoding, GA(B) uses the binary encoding, and GA(B)_I is a variation of GA(B) where the properties of the problem are applied into two phases of the algorithm. The methods presented in this paper have been exhaustively tested in different test beds. The B&B solves small size instances more efficiently than the existing DP method. The constructive heuristics presented provide approximate solutions almost instantaneously for all instances. The best heuristics regarding the quality of the solutions (i.e. CH3 and CH7) are shown to be robust for different problem sizes. Regarding the versions of the GAs, GA(B) (binary encoding) performs better than GA(P) (permutation encoding), showing the importance of the encoding used. Moreover, embedding additional problem properties into GA(B) produces an improved version –GA(B)_I – that outperforms the other versions.

A future research line could be to apply the binary encoding and its properties to similar problems, testing adapted versions of the methods presented, and designing local search procedures based on the properties, since the good performance of some constructive heuristics indicates that there is room for improving the results. Finally, the application of pseudo-polynomial approximation algorithms based on the binary encoding can be very interesting to solve the problem, determining the dependency of the complexity of these algorithms with respect to parameter $\epsilon$.

## Acknowledgements

The authors are sincerely grateful to the anonymous referees, who provide very valuable comments on the earlier version of the paper. This research has been funded by the Spanish Ministry of Science and Innovation, under projects "SCORE" with reference DPI2010-15573/DPI, and "ADDRESS" with reference DPI2013-44461-P/DPI.

## References

Agnetis, A., De Pascale, G., and Pacciarelli, D. (2009). A lagrangian approach to single-machine scheduling problems with two competing agents. *Journal of Scheduling*, 12(4):401–415.

Agnetis, A., Mirchandani, P. B., Pacciarelli, D., and Pacifici, A. (2004). Scheduling problems with two competing agents. *Operations Research*, 52(2):229–242.

Agnetis, A., Nicosia, G., Pacifici, A., and Pferschy, U. (2013). Two agents competing for a shared machine.

Arbib, C., Smriglio, S., and Servilio, M. (2004). A competitive scheduling problem and its relevance to umts channel assignment. *Networks*, 44(2):132–141.

Armentano, V. A. and Ronconi, D. P. (1999). Tabu search for total tardiness minimization in flowshop scheduling problems. *Computers and Operations Research*, 26(3):219–235.

Balasubramanian, H., Fowler, J., Keha, A., and Pfund, M. (2009). Scheduling interfering job sets on parallel machines. *European Journal of Operational Research*, 199(1):55–67.

Bozejko, W. (2010). Parallel path relinking method for the single machine total weighted tardiness problem with sequence-dependent setups. *Journal of Intelligent Manufacturing*, 21(6):777–785. cited By 10.

Chang, H.-C. and Liu, T.-K. (2015). Optimisation of distributed manufacturing flexible job shop scheduling by using hybrid genetic algorithms. *Journal of Intelligent Manufacturing*. cited By 0; Article in Press.

Chang, J. L., Gong, D. W., and Ma, X. P. (2007). A heuristic genetic algorithm for no-wait flowshop scheduling problem. *Journal of China University of Mining and Technology*, 17(4):582–586.

Damodaran, P., Hirani, N. S., and Velez-Gallego, M. C. (2009). Scheduling identical parallel batch processing machines to minimise makespan using genetic algorithms. *European Journal of Industrial Engineering*, 3(2):187–206.

Fan, J. (2010). Integrated production and delivery scheduling problem with two competing agents. volume 2 of *ICAMS 2010 - Proceedings of 2010 IEEE International Conference on Advanced Management Science*, pages 157–160.

Framinan, J. M. (2007). An adaptive branch and bound approach for transforming job shops into flow shops. *Computers and Industrial Engineering*, 52(1):1–10.

Framinan, J. M. (2009). A fitness-based weighting mechanism for multicriteria flowshop scheduling using genetic algorithms. *The International Journal of Advanced Manufacturing Technology*, 43(9):939–948.

Framinan, J. M., Leisten, R., and Ruiz, R. (2014). *Manufacturing Scheduling Systems: An Integrated View on Models, Methods, and Tools.* Springer, London (UK).

Framinan, J. M., Ruiz Usano, R., and Leisten, R. (2001). Sequencing conwip flow-shops: analysis and heuristics. *International Journal of Production Research*, 39(12):2735–2749.

Gawiejnowicz, S., Lee, W. C., Lin, C. L., and Wu, C. C. (2010). A branch-and-bound algorithm for two-agent single-machine scheduling of deteriorating jobs. pages 207–211.

Hall, N. G. and Potts, C. N. (2004). Rescheduling for new orders. *Operations Research*, 52(3):440–453.

Kellerer, H. and Strusevich, V. A. (2010). Fully polynomial approximation schemes for a symmetric quadratic knapsack problem and its scheduling applications. *Algoritmica*, 57(4):769–795.

Khelifati, S. L. and Bouzid-Sitayeb, F. (2011). A multi-agent scheduling approach for the joint scheduling of jobs and maintenance operations in the flow shop sequencing problem. *Lecture Notes in Computer Science*, 6923 LNAI(PART 2):60–69.

Khowala, K., Fowler, J., Keha, A., and Balasubramanian, H. (2014). Single machine scheduling with interfering job sets. *Computers and Operations Research*, 45(0):97–107.

Li, D., Meng, X., Liang, Q., and Zhao, J. (2014). A heuristic-search genetic algorithm for multi-stage hybrid flow shop scheduling with single processing machines and batch processing machines. *Journal of Intelligent Manufacturing*, pages 1–18. cited By 0; Article in Press.

Li, D. C. and Hsu, P. H. (2012). Solving a two-agent single-machine scheduling problem considering learning effect. *Computers and Operations Research*, 39(7):1644–1651.

Luo, H., Huang, G. Q., Zhang, Y., Dai, Q., and Chen, X. (2009). Two-stage hybrid batching flowshop scheduling with blocking and machine availability constraints using genetic algorithm. *Robotics and Computer-Integrated Manufacturing*, 25(6):962–971.

Meiners, C. R. and Torng, E. (2007). Mixed criteria packet scheduling. *Lecture Notes in Computer Science*, 4508:120–133.

Mu, Y. and Gu, C. (2010). Rescheduling to minimize makespan under a limit on the makespan of the original jobs. volume 1 of *The 2nd International Conference on Computer and Automation Engineering, ICCAE 2010*, pages 613–617.

Ni, Y. and Zhao, Z. (2014). Two-agent scheduling problem under fuzzy environment. *Journal of Intelligent Manufacturing*. cited By 0; Article in Press.

Peha, J. M. (1995). Heterogeneous-criteria scheduling: Minimizing weighted number of tardy jobs and weighted completion time. *Computers and Operations Research*, 22(10):1089–1100.

Peha, J. M. and Tobagi, F. A. (1990). Evaluating scheduling algorithms for traffic with heterogeneous performance objectives. Global Telecommunications Conference, 1990, and Exhibition.'Communications: Connecting the Future', GLOBECOM '90., IEEE, pages 21–27.

Perez-Gonzalez, P. and Framinan, J. M. (2009). Scheduling permutation flowshops with initial availability constraint: Analysis of solutions and constructive heuristics. *Computers and Operations Research*, 36(10):2866–2876.

Perez-Gonzalez, P. and Framinan, J. M. (2010). Setting a common due date in a constrained flowshop: A variable neighbourhood search approach. *Computers and Operations Research*, 37(10):1740–1748.

Perez-Gonzalez, P. and Framinan, J. M. (2014). A common framework and taxonomy for multicriteria scheduling problems with interfering and competing jobs: Multi-agent scheduling problems. *European Journal of Operational Research*, 235(1):1–16.

Pinedo, M. (1995). *Scheduling: Theory, algorithms, and systems.* Prentice Hall International Series in Industrial and System Engineering. Prentice Hall, Englewood Cliffs, New Jersey (USA).

Ruiz, R. and Allahverdi, A. (2007). No-wait flowshop with separate setup times to minimize maximum lateness. *The International Journal of Advanced Manufacturing Technology*, 35(5):551–565.

Ruiz, R. and Allahverdi, A. (2009). Minimizing the bicriteria of makespan and maximum tardiness with an upper bound on maximum tardiness. *Computers and Operations Research*, 36(4):1268–1283.

Ruiz, R. and Maroto, C. (2006). A genetic algorithm for hybrid flowshops with sequence dependent setup times and machine eligibility. *European Journal of Operational Research*, 169(3):781–800.

Ruiz, R., Maroto, C., and Alcaraz, J. (2005). Solving the flowshop scheduling problem with sequence dependent setup times using advanced metaheuristics. *European Journal of Operational Research*, 165(1):34–54.

Sabouni, M., Jolai, F., and Mansouri, A. (2010). Heuristics for minimizing total completion time and maximum lateness on identical parallel machines with setup times. *Journal of Intelligent Manufacturing*, 21(4):439–449. cited By 4.

Soltani, R., Jolai, F., and Zandieh, M. (2010). Two robust meta-heuristics for scheduling multiple job classes on a single machine with multiple criteria. *Expert Systems with Applications*, 37(8):5951–5959.

Taillard, E. D. (1990). Some efficient heuristic methods for the flow shop sequencing problem. *European Journal of Operational Research*, 47(1):65–74.

T'kindt, V. and Billaut, J. C. (2002). *Multicriteria scheduling: Theory, models and algorithms.* Springer, Berlin (Germany), second edition.

Tseng, L. Y. and Lin, Y. T. (2010). A genetic local search algorithm for minimizing total flowtime in the permutation flowshop scheduling problem. *International Journal of Production Economics*, 127(1):121–128.

Unal, A. T., Uzsoy, R., and Kiran, A. S. (1997). Rescheduling on a single machine with part-type dependent setup times and deadlines. *Annals of Operations Research*, 70:93–113.

Ventura, J. and Yoon, S.-H. (2013). A new genetic algorithm for lot-streaming flow shop scheduling with limited capacity buffers. *Journal of Intelligent Manufacturing*, 24(6):1185–1196. cited By 7.

Wan, G., Vakati, S. R., Leung, J. Y. T., and Pinedo, M. L. (2010). Scheduling two agents with controllable processing times. *European Journal of Operational Research*, 205(3):528–539.

Wu, W.-H., Cheng, S.-R., Wu, C.-C., and Yin, Y. (2012). Ant colony algorithms for a two-agent scheduling with sum-of processing times-based learning and deteriorating considerations. *Journal of Intelligent Manufacturing*, 23(5):1985–1993. cited By 0.

Yao, L., Shi, H., Liu, C., and Han, Z. (2011). Solving the two-objective shop scheduling problem in mto manufacturing systems by a novel genetic algorithm. *Advanced Materials Research*, 314-316:1315–1320.

Yuan, J. and Mu, Y. (2007). Rescheduling with release dates to minimize makespan under a limit on the maximum sequence disruption. *European Journal of Operational Research*, 182(2):936–944.

Yuan, J., Mu, Y., Lu, L., and Li, W. (2007). Rescheduling with release dates to minimize total sequence disruption under a limit on the makespan. *Asia - Pacific Journal of Operational Research*, 24(6):789–796.

Yuan, J. J., Ng, C. T., and Cheng, T. C. E. (2013). Two-agent single-machine scheduling with release dates and preemption to minimize the maximum lateness.