Noname manuscript No.
(will be inserted by the editor)

A memetic algorithm for restoring feasibility in scheduling with limited makespan

Raúl Mencía · Carlos Mencía · Ramiro Varela

Received: date / Accepted: date

Abstract When solving a scheduling problem, users are often interested in finding a schedule optimizing a given objective function. However, in some settings there can be hard constraints that make the problem unfeasible. In this paper we focus on the task of repairing infeasibility in job shop scheduling problems with a hard constraint on the makespan. In this context, earlier work addressed the problem of computing the largest subset of the jobs that can be scheduled within the makespan constraint. Herein, we face a more general weighted version of the problem, consisting in computing a feasible subset of jobs maximizing their weighted sum. To this aim, we propose an efficient memetic algorithm, that combines a genetic algorithm with a local search method, also proposed in the paper. The results from an experimental study show the practical suitability of our approach.

1 Introduction

Scheduling problems have attracted a large body of research over the last decades, due to their practical interest and their usually high computational complexity [7]. As a result, a wide range of methods have been proposed for solving increasingly challenging scheduling problems. Arguably, the job shop scheduling problem (JSP) has been one of the most studied problems in this class, which serves to model a variety of different settings while keeping its formulation simple.

Typically, when solving a scheduling problem the goal is to find a schedule that optimizes a given objective function. However, in some cases there can be additional constraints that make the problem unfeasible, that is, with no possible solutions. This situation gives rise to the task of repairing the observed inconsistecy, so that *solving* the problem to some extent.

In this paper we focus on repairing unfeasible job shop scheduling problems with a hard constraint imposing that all the jobs must be completed within a

Department of Computer Science,
University of Oviedo, Campus of Gijón, Gijón 33204, Spain
{menciaraul, menciacarlos, ramiro}@uniovi.es
`http://www.di.uniovi.es/iscop`

given makespan limit. This kind of constraint appears naturally in practice, and has been studied in the literature (e.g. [9,8,1]).

Recently, in [17], a setting was considered in which unfeasible problem instances can be relaxed by removing some of the jobs, so that the remaining ones can be scheduled under the given makespan constraint. This setting yields the optimization problem of computing the largest feasible subset of the jobs, for which an efficient genetic algorithm (GA) was proposed in [17]. Whereas the largest subset of jobs that can be scheduled under the given makespan constraint represents a useful notion for repairing inconsistency, it does not take into account that some jobs might be more prioritary than others. This way, we consider a more general formulation of the problem, where each job has a positive weight, and focus on computing a feasible subset of jobs maximizing their weighted sum.

For solving this problem, we show that the genetic algorithm proposed in [17] can be easily adapted to handle the weights, and that the search space where it looks for solutions is still dominant when weights are considered. In addition, we propose an effective local search algorithm for this problem, that aims at improving the quality of solutions. The local search method is combined with GA, resulting in a memetic algorithm (MA). The results from an experimental study indicate that the proposed methods are successful at solving the problem, and that the local search approach enables the memetic algorithm to outperform the genetic algorithm.

The remainder of the paper is organized as follows. In Section 2 we give a formal definition of the problem. Section 3 reviews the genetic algorithm proposed in [17], which is adapted to the weighted version of the problem considered herein. Section 4 is devoted to the local search algorithm we propose, which is integrated in the memetic algorithm presented in Section 5. All the methods are evaluated in an experimental study reported in Section 6. Finally, we summarize the main conclusions in Section 7.

2 Problem formulation

The job shop scheduling problem (JSP) consists in scheduling a set of n jobs $\mathcal{J} = \{J_1, \ldots, J_n\}$ on a set of m resources or machines $\mathcal{M} = \{M_1, \ldots, M_m\}$. Each job $J_i \in \mathcal{J}$ consists of a sequence of m tasks or operations $(\theta_{i1}, \ldots, \theta_{im})$, where each operation requires a machine $M(\theta_{ij})$ during a (positive integer) processing time $p_{\theta_{ij}}$.

A schedule S is an assignment of a starting time $st_{\theta_{ij}}$ to each of the operations such that the following constraints are satisfied:

i. The operations in a job must be scheduled in the order they appear in the job, i.e., $st_{\theta_{ij}} + p_{\theta_{ij}} \leq st_{\theta_{i(j+1)}}$ with $i = 1, ..., n$ and $j = 1, ..., m-1$.
ii. No machine can process more than one operation at a time, which translates in disjunctive constraints of the form $(st_u + p_u \leq st_v) \vee (st_v + p_v \leq st_u)$ for all operations u, v with $u \neq v$ and $M(u) = M(v)$.
iii. Preemption is not allowed, i.e., $C_u = st_u + p_u$ for all u, where C_u denotes the completion time of operation u.

The makespan of a schedule S is defined as the maximum completion time of the operations in S, and it is denoted as $C_{max}(S)$. This metric has been commonly considered as an objective function to be minimized, resulting in the optimization version of the JSP denoted $J||C_{max}$ in the $\alpha|\beta|\gamma$ notation proposed in [12].

Considering the makespan, the decision version of the JSP is the problem of deciding, for a given problem instance, whether there exists a schedule S such that $C_{max}(S) \leq C$, where C is a fixed limit on the maximum makespan allowed. If such a schedule S exists, the problem instance is said to be *feasible*, whereas it is *unfeasible* otherwise.

In this paper, we focus on the task of repairing unfeasible problem instances with a hard constraint on the makespan in the best possible way. For this purpose, we consider a setting that allows for dropping some of the jobs, so that the remaining ones can be scheduled within the makespan limit. Based on this, in the following, we refer to an unfeasible problem instance by a pair $\mathcal{I} = (\mathcal{J}, C)$, consisting in scheduling the set of jobs $\mathcal{J}$ within a maximum makespan of C.

The following definitions serve to characterize suitable repairs, which build on related concepts in the field of the analysis of unsatisfiable propositional formulas, such as *maximal satisfiable subformula* (MSS) or *maximum satisfiability* (maxSAT) [2,14,23].

Definition 1 (FSJ) Given an unfeasible instance $\mathcal{I} = (\mathcal{J}, C)$, $\mathcal{S} \subsetneq \mathcal{J}$ is a *feasible subset of jobs* (FSJ) of $\mathcal{J}$ iff $(\mathcal{S}, C)$ is feasible.

Definition 2 (MFSJ) Given an unfeasible instance $\mathcal{I} = (\mathcal{J}, C)$, $\mathcal{S} \subsetneq \mathcal{J}$ is a *maximal feasible subset of jobs* (MFSJ) of $\mathcal{J}$ iff $(\mathcal{S}, C)$ is feasible and for all $\mathcal{S}' \subseteq \mathcal{J}$ with $\mathcal{S} \subsetneq \mathcal{S}'$, $(\mathcal{S}', C)$ is unfeasible.

Definition 3 (maxFSJ) Given an unfeasible instance $\mathcal{I} = (\mathcal{J}, C)$, $\mathcal{S}^* \subsetneq \mathcal{J}$ is a *maximum feasible subset of jobs* (maxFSJ) of $\mathcal{J}$ iff $(\mathcal{S}^*, C)$ is feasible and for all FSJs $\mathcal{S}'$ of $\mathcal{J}$, $|\mathcal{S}'| \leq |\mathcal{S}^*|$.

FSJs constitute subsets of the jobs that can be scheduled within the makespan limit. Among the set of all possible FSJs, MFSJs and maxFSJs exhibit two different forms of maximality. MFSJs are maximal w.r.t. set inclusion, i.e. no superset of an MFSJ is an FSJ and so they represent a kind of *local* maxima. On the other hand, maxFSJs are FSJs with maximum cardinality, i.e. the largest possible FSJs. If only one of the previous sets is to be computed, arguably, maxFSJs might be the most useful option, due to their largest size.

Recent work [17] addressed the problem approximating maxFSJs. Therein, the following two results were proven (we omit the proofs, and refer the interested reader to [17]):

Proposition 1 *Let $\mathcal{I} = (\mathcal{J}, C)$ be an unfeasible problem instance and $\mathcal{S} \subsetneq \mathcal{J}$ an FSJ of $\mathcal{J}$. Then, for all $\mathcal{S}' \subseteq \mathcal{S}$, $\mathcal{S}'$ is an FSJ of $\mathcal{J}$.*

Proposition 2 *Let $\mathcal{I} = (\mathcal{J}, C)$ be an unfeasible problem instance. $\mathcal{S} \subsetneq \mathcal{J}$ is an MFSJ of $\mathcal{J}$ iff $(\mathcal{S}, C)$ is feasible and for all $j \in \mathcal{J} \setminus \mathcal{S}$, $(\mathcal{S} \cup \{j\}, C)$ is unfeasible.*

Proposition 1 states that all subsets of an FSJ are feasible subsets of jobs as well. This monotonicity property allows for an alternative definition of MFSJs, stated in Proposition 2.

As pointed out, maxFSJs constitute useful ways of repairing infeasibility. However, in some settings some jobs may be more prioritary than others, and maxFSJs

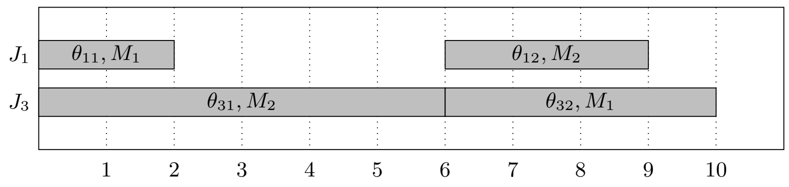


Fig. 1 Gantt chart of a schedule for the weighted-maxFSJ $\{J_1, J_3\}$ in Example 1.

do not take this information into account. In this respect, we consider a more general formulation, in which each job $j \in \mathcal{J}$ has a (positive integer) weight, denoted $w(j)$. Then, for a set of jobs $\mathcal{S}$, we define its *total weight* as $w(\mathcal{S}) = \sum_{j \in \mathcal{S}} w(j)$. Weights allow for an alternative kind of repairs:

Definition 4 (weighted-maxFSJ) Given an unfeasible instance $\mathcal{I} = (\mathcal{J}, C)$, $\mathcal{S}^* \subsetneq \mathcal{J}$ is a *weighted maximum feasible subset of jobs* (weighted-maxFSJ) of $\mathcal{J}$ iff $(\mathcal{S}^*, C)$ is feasible and for all FSJs $\mathcal{S}'$ of $\mathcal{J}$, $w(\mathcal{S}') \leq w(\mathcal{S}^*)$.

So, weighted-maxFSJs are feasible subsets of jobs with the maximum possible total weight. Clearly, if all the weights are equal, weighted-maxFSJs correspond to maxFSJs. Thus, the problem of computing a maxFSJ is a particular case of that of computing a weighted-maxFSJ for a given unfeasible problem instance.

Example 1 Consider a job shop with four jobs $\mathcal{J} = \{J_1, J_2, J_3, J_4\}$ and two machines $\mathcal{M} = \{M_1, M_2\}$. Each job J_i has a weight $w(J_i)$ and consists of a sequence of two operations $(\theta_{i1}, \theta_{i2})$, with processing times and required machine as shown in the following table:

	J_1	J_2	J_3	J_4
θ_{i1}	2 (M_1)	3 (M_1)	6 (M_2)	5 (M_2)
θ_{i2}	3 (M_2)	2 (M_2)	4 (M_1)	5 (M_1)
w	4	2	7	4

With a hard constraint limiting the makespan to $C = 10$ the instance $(\mathcal{J}, C)$ is unfeasible. There are 11 FSJs of $\mathcal{J}$: $\emptyset$, $\{J_1\}$, $\{J_2\}$, $\{J_3\}$, $\{J_4\}$, $\{J_1, J_2\}$, $\{J_1, J_3\}$, $\{J_1, J_4\}$, $\{J_2, J_3\}$, $\{J_2, J_4\}$ and $\{J_1, J_2, J_4\}$ as there exists a schedule with makespan less than or equal to 10 for each of them. Three of these sets are MFSJs: $\mathcal{S}_1 = \{J_1, J_3\}$, $\mathcal{S}_2 = \{J_2, J_3\}$ and $\mathcal{S}_3 = \{J_1, J_2, J_4\}$, with $w(\mathcal{S}_1) = 11$, $w(\mathcal{S}_2) = 9$ and $w(\mathcal{S}_3) = 10$. Besides, only $\mathcal{S}_3$ is a maxFSJ of $\mathcal{J}$, and the only weighted-maxFSJ is $\mathcal{S}_1$. Fig. 1 shows a schedule for the weighted-maxFSJ with makespan not exceeding the limit of 10.

In this paper, we address the maximization problem of approximating weighted-maxFSJs for a given unfeasible problem instance $(\mathcal{J}, C)$, that is finding feasible subsets of jobs with the maximum possible total weight. This problem is related to the problem considered in [9], where the goal is to find a feasible subset of jobs in a two-stage flow shop maximizing a weighted sum of the jobs. However, deciding the feasibility of a two-stage flow shop can be done in polynomial time, whereas deciding the feasibility of a JSP instance is known to be NP-complete [10].

Algorithm 1: Solution Builder.

```
Data: Set of jobs $\mathcal{J}$, makespan limit $C$
Result: $\mathcal{S} \subsetneq \mathcal{J}$ an MFSJ of $\mathcal{J}$
$\mathcal{R} \leftarrow \mathcal{J}$; // Initialize reference set to the set of all jobs
$\mathcal{S} \leftarrow \emptyset$; // Initialize $\mathcal{S}$ to the empty set
while $\mathcal{R} \neq \emptyset$ do
    Pick $j \in \mathcal{R}$; //Non-deterministically
    $\mathcal{R} \leftarrow \mathcal{R} \setminus \{j\}$;
    if $Feasible(\mathcal{S} \cup \{j\}, C)$ then
        $\mathcal{S} \leftarrow \mathcal{S} \cup \{j\}$;
    end
end
return $\mathcal{S}$;
```

3 Genetic algorithm for computing weighted-maxFSJs

In this section we review the genetic algorithm (GA) proposed in [17] for computing (unweighted) max-FSJs, and show that it can be easily extended, keeping all its properties, to handle weights by only changing its evaluation function.

The GA looks for solutions in the search space defined by a *solution builder* aimed at approximating MFSJs of $\mathcal{J}$ for a given unfeasible problem instance $\mathcal{I} = (\mathcal{J}, C)$. Restricting the search to the set of MFSJs allows for reducing the size of the search space while still being dominant, i.e., this set contains optimal solutions for any given problem instance. Noticeably, this property holds for the weighted version of the problem as well, what follows from the fact that all weighted-maxFSJs are MFSJs too, as we prove next:

Proposition 3 *Let $\mathcal{S}^* \subsetneq \mathcal{J}$ be a weighted-maxFSJ of $\mathcal{J}$ for a given unfeasible problem instance $\mathcal{I} = (\mathcal{J}, C)$. Then, $\mathcal{S}^*$ is an MFSJ of $\mathcal{J}$ as well.*

Proof Since $\mathcal{S}^*$ is a weighted-maxFSJ of $\mathcal{J}$, by Definition 4, the instance $(\mathcal{S}^*, C)$ is feasible, i.e., $\mathcal{S}^*$ is an FSJ of $\mathcal{J}$. Let us suppose that $\mathcal{S}^*$ is not an MFSJ of $\mathcal{J}$. Then, by Definition 2, there must exist an FSJ $\mathcal{S}'$ with $\mathcal{S}^* \subsetneq \mathcal{S}' \subsetneq \mathcal{J}$. As every job in $\mathcal{J}$ has a positive weight and $\mathcal{S}'$ is a strict superset of $\mathcal{S}^*$, it necessarily follows that $w(\mathcal{S}') > w(\mathcal{S}^*)$, and so $\mathcal{S}^*$ is not a weighted-maxFSJ of $\mathcal{J}$. A contradiction.

The result above enables searching for optimal solutions in the (reduced) space of MFSJs. In practice, as pointed out above, the search space is defined with the use of a solution builder which is integrated into the genetic algorithm in its decoding phase. In the following subsections we describe both the solution builder and the main components of the GA.

3.1 Solution builder

When solving scheduling problems, the search space is commonly defined by means of *schedule builders*, also known as *schedule generation schemes*, e.g. [11,3,22,19]. These are non-deterministic methods that enable the computation and enumeration of a subset of the schedules for a given problem instance, thus defining a search space. However, for the task of approximating weighted-maxFSJs, besides

Table 1 Trace of Algorithm 1 on the instance in Example 1

It.	j	$\mathit{Feasible}(\mathcal{S} \cup \{j\}, C)$	$\mathcal{S}$	$\mathcal{R}$
Init.	-	-	$\emptyset$	$\{J_1, J_2, J_3, J_4\}$
1	J_3	`true`	$\{J_3\}$	$\{J_1, J_2, J_4\}$
2	J_4	`false`	$\{J_3\}$	$\{J_1, J_2\}$
3	J_1	`true`	$\{J_1, J_3\}$	$\{J_2\}$
4	J_2	`false`	$\{J_1, J_3\}$	$\emptyset$

computing schedules we need to search in the subset space of the jobs in order to identify feasible subsets of jobs, so a different kind of procedure is necessary for this purpose. This way, the search space is defined by means of a *solution builder* that aims at computing MFSJs of a given unfeasible problem instance.

Its pseudocode is shown in Algorithm 1. The solution builder follows a *linear search* approach [4,13] for computing an MFSJ. The algorithm maintains a set $\mathcal{S}$ (initially empty) and a set $\mathcal{R}$ with all the jobs that have not yet been considered (initialized as $\mathcal{J}$). Throughout the course of the algorithm, $\mathcal{S}$ represents an FSJ of $\mathcal{J}$ and this set grows until eventually representing an MFSJ of $\mathcal{J}$. Iteratively, until all the jobs have been considered, the algorithm selects a job $j \in \mathcal{R}$ and tests whether the instance $(\mathcal{S} \cup \{j\}, C)$ is feasible or not, i.e., whether there exists a schedule for the jobs in $\mathcal{S} \cup \{j\}$ with makespan not exceeding the given limit C. If such a schedule exists, the job j is added to $\mathcal{S}$. In either case, j is removed from $\mathcal{R}$. The algorithm terminates after $|\mathcal{J}|$ iterations, when $\mathcal{R}$ becomes empty.

The solution builder relies on the use of a procedure *Feasible* for testing the feasibility of different subsets of jobs. Whenever *Feasible* is a complete decision procedure (e.g. [16]), Algorithm 1 correctly computes an MFSJ of $\mathcal{J}$, which follows from Proposition 2.

We note that the selection of the job j in each iteration is non-deterministic. As an example, if we consider the unfeasible problem instance in Example 1, the sequence of choices (J_1, J_2, J_3, J_4) would result in the MFSJ $\{J_1, J_2, J_4\}$. Alternatively, the sequence of choices (J_3, J_4, J_1, J_2) would lead the algorithm to computing the MFSJ $\{J_1, J_3\}$. In addition, considering all possible sequences of choices (i.e. permutations of the jobs) results in the definition of a search space that contains all MFSJs of a given problem instance, including all the weighted-maxFSJs (what follows from Proposition 3). In order to clarify how this algorithm works, we show a trace in the following example.

Example 2 Let us consider the unfeasible problem instance in Example 1, consisting in scheduling the jobs $\{J_1, J_2, J_3, J_4\}$ with a hard constraint limiting the makespan to $C = 10$. Table 1 shows a trace of Algorithm 1 assuming that it implements *Feasible* as a complete decision procedure, following the sequence of choices (J_3, J_4, J_1, J_2). Concretely, for each iteration, the table shows the selected job $j \in \mathcal{R}$, the result of the feasibility test on the instance $(\mathcal{S} \cup \{j\}, C)$ as well as the content of the sets $\mathcal{S}$ and $\mathcal{R}$ at the end of the corresponding iteration. As we can observe, Algorithm 1 produces the MFSJ $\{J_1, J_3\}$.

As pointed out, each feasibility test requires solving an NP-complete problem, so using a complete decision procedure (i.e. an exact algorithm) for this purpose may be impractical. Alternatively one could use an incomplete procedure, at the expense of losing the guarantee that the computed subset represents an MFSJ.

Algorithm 2: Schedule builder $G\&T$.

```
Data: A JSP problem instance $\mathcal{P}$
Result: A feasible schedule $S$ for $\mathcal{P}$
$\mathcal{A} \leftarrow \{t_{i1}; J_i \in \mathcal{J}\}$;
$SC \leftarrow \emptyset$;
while $\mathcal{A} \neq \emptyset$ do
    $v^* \leftarrow argmin\{r_v + p_v; v \in \mathcal{A}\}$;
    $\mathcal{B} \leftarrow \{u \in \mathcal{A}; M(u) = M(v^*), r_u < r_{v^*} + p_{v^*}\}$;
    Pick $u \in \mathcal{B}$ non deterministically;
    Set $st_u \leftarrow r_u$ in $S$;
    Add $u$ to $SC$ and update $r_v$ for all $v \notin SC$;
    $\mathcal{A} \leftarrow \{v; v \notin SC, P(v) \subseteq SC\}$;
end
return the built schedule $S$;
```

Given a feasibility test on an instance $(\mathcal{S} \cup \{j\}, C)$, the incomplete procedure would look for a schedule for the set of jobs in $\mathcal{S} \cup \{j\}$ with makespan not exceeding C. If such a schedule is found, the instance would be correctly declared feasible. Otherwise, there would not be the guarantee that the instance is unfeasible, although Algorithm 1 would treat it as such (not adding the job j to the set $\mathcal{S}$). As a result, the computed set could be an under-approximation of an MFSJ but, in any case, it will always be a feasible subset of the jobs (since it cannot happen that an unfeasible instance is declared feasible).

In order to search for schedules for a given JSP instance, our approach uses the well-known G&T schedule builder [11]. This algorithm schedules one operation at a time iteratively. Algorithm 2 shows its pseudocode, where for an operation u, $P(u)$ denotes its immediate predecessor in its job and r_u denotes its earliest possible starting time, referred to as the *head* of u. The algorithm maintains a set SC with the operations scheduled so far and the set $\mathcal{A}$ containing all the unscheduled operations that are either the first one in their job or whose immediate predecessor in their job has been already scheduled. At each iteration, the algorithm identifies the operation $v^* \in \mathcal{A}$ with the earliest possible completion time and builds a set $\mathcal{B}$ with all the operations $v \in \mathcal{A}$ that require the same machine as v^* and can be scheduled before the earliest possible completion time of v^*. At this point, one operation in $\mathcal{B}$ is selected and scheduled at its earliest possible starting time, updating SC, $\mathcal{A}$ and the heads of the operations that remain to be scheduled.

On termination, the G&T algorithm is guaranteed to return an *active* schedule, in which no operation can be scheduled earlier without delaying the starting time of some other operation. The selection of the job $j \in \mathcal{B}$ to be scheduled at a given iteration is non-deterministic. This way, the G&T schedule builder defines a search space, particularly that formed by the set of all active schedules, which is dominant for the makespan, i.e. it always contains at least one schedule with the minimum possible makespan.

3.2 Main components of the genetic algorithm

This section reviews the main components of the GA proposed in [17] for approximating maxFSJs, that we extend herein for approximating weighted-maxFSJs. Its structure is shown in Algoritm 3: it is a generational genetic algorithm with

Algorithm 3: Genetic Algorithm.

```
Data: A problem instance $\mathcal{P}$ and a set of parameters $(P_c, P_m, \#gen, \#popsize)$
Result: A solution for $\mathcal{P}$
Generate and evaluate the initial population $P(0)$;
for t=1 to #gen-1 do
    Selection: organize the chromosomes in $P(t-1)$ into pairs at random ;
    Recombination: mate each pair of chromosomes and mutate the two offsprings
      in accordance with $P_c$ and $P_m$;
    Evaluation: evaluate the resulting chromosomes;
    Replacement: make a tournament selection among every two parents and their
      offsprings to generate $P(t)$;
end
return the best solution built so far;
```

random selection and replacement by tournament among parents and offsprings, which confers the GA an implicit form of elitism. The algorithm requires the following four parameters: crossover and mutation probabilities (P_c and P_m), number of generations (*#gen*) and population size (*#popsize*). Below, we describe its most important elements:

3.2.1 Coding schema

The GA exploits a coding schema commonly used for solving job shop scheduling problems, in which chromosomes are permutations with repetitions [6]. Particularly, for an instance with n jobs and m machines, a chromosome is a permutation of the job indices where each job appears m times.

A permutation represents a tentative ordering of the operations, that we refer to as *operation sequence*: in a chromosome the j-th occurrence of job J_i (from left to right) corresponds to the operation θ_{ij}. For example, the chromosome (4, 1, 1, 3, 2, 4, 2, 3) represents the operation sequence $(\theta_{41}, \theta_{11}, \theta_{12}, \theta_{31}, \theta_{21}, \theta_{42}, \theta_{22}, \theta_{32})$ for an instance with 4 jobs and 2 machines. This sequence can be exploited to guide a schedule builder, such as the G&T algorithm, in the search of a schedule for a given JSP instance.

In addition, this representation can be also used to guide a solution builder in the construction of an (approximate) MSFJ as it establishes a total order of the jobs, that we refer to as *job sequence*. Concretely, given a chromosome the job sequence it represents is defined by the order of the jobs w.r.t. their first appearance in the chromosome. For instance, the chromosome (4, 1, 1, 3, 2, 4, 2, 3) represents the job ordering (J_4, J_1, J_3, J_2). As a result, the coding schema, in combination with the decoding algorithm presented below allows the GA to search in both the subset space of the set of jobs and in the space of schedules for any given subset of jobs *at the same time.*

3.2.2 Decoding algorithm

The GA uses the solution builder depicted in Algorithm 1 as a decoder. Given a chromosome, the decoder builds an (approximate) MFSJ using the solution builder in a way that it selects the jobs in the order they appear in the job sequence encoded in the chromosome. In other words, at the i-th iteration of the algorithm, it picks the job appearing in the i-th position of the job sequence.

The procedure *Feasible* in Algorithm 1 is implemented by an invocation to the G&T schedule builder described in Algorithm 2, which is guided by the operation sequence extracted from the chromosome. This way, it is used as a greedy algorithm. At each iteration, the operation in the set $\mathcal{B}$ that appears first in the operation sequence is chosen to be scheduled. If the makespan of the resulting schedule satisfies the hard constraint on the makespan, the instance declared feasible, and unfeasible otherwise. Since this does not constitute a complete decision procedure, there is no guarantee that an MFSJ would be computed, but an approximation instead, as discussed in Section 3.1.

After the execution of the solution builder, the total weight of the computed set is taken as the fitness of the chromosome. Notice that this is the only difference with respect to the GA in [17], which considered the cardinality of the computed set instead.

3.2.3 Crossover and mutation

For crossover, the GA uses the well-known *Job-based Order Crossover* (JOX) [21], specifically designed for permutations with repetitions. Given a pair of individuals, JOX selects a random subset of the jobs and copies their genes to the offspring in the same positions as they are in the first chromosome; then the remaining genes are taken from the second chromosome maintaining their relative order. The second child is computed by the same procedure swapping the role of the parents.

On the other hand, the mutation operator implements a simple procedure, which randomly swaps two consecutive positions of the chromosome.

4 Local search

Local search stands out as a very effective framework for solving hard combinatorial optimization problems and, as such, it has been successfuly applied in the field of scheduling as well (e.g. [20, 27, 18, 26]). Although many different local search algorithms exist in the literature, they all follow the same underlying principle: starting from a solution to a given problem instance, a local search algorithm introduces small perturbations to the solution, replacing it by a new found solution if the latter has higher quality. This process is repeated until no further improvements are observed (or other termination condition is met).

In this section, we propose a local search algorithm aimed at improving the solutions computed by the GA described above. Note that, because of the solution builder used by the GA in its decoding phase, these solutions constitute approximations of MFSJs so trying to add additional jobs to these sets does not seem to lead to any improvements if the same procedure is used for testing the feasibility of a given subset of the jobs. In contrast, our approach consists in iteratively *substituting* one job in the current solution by another one with a greater weight and such that it is not contained in the corresponding FSJ. However, such substitutions may lead to unfeasible problem instances and so not all posible substitutions may lead to new actual solutions (i.e. FSJs).

Algorithm 4: Procedure *ComputePotentialNeighbors*.

```
Data: An unfeasible instance $\mathcal{I} = (\mathcal{J}, C)$, an FSJ $\mathcal{S} \subsetneq \mathcal{J}$
Result: Set of pairs representing potential neigboring FSJs improving $\mathcal{S}$
$\mathcal{N} \leftarrow \emptyset$;
for $s \in \mathcal{S}$ do
    for $u \in \mathcal{J} \setminus \mathcal{S}$ do
        if $w(u) > w(s)$ then
            $\mathcal{N} \leftarrow \mathcal{N} \cup \{(s, u)\}$;
        end
    end
end
return $\mathcal{N}$;
```

Algorithm 5: Local search algorithm.

```
Data: An unfeasible instance $\mathcal{I} = (\mathcal{J}, C)$, an FSJ $\mathcal{S} \subsetneq \mathcal{J}$
Result: An FSJ $\mathcal{S}'$ with $w(\mathcal{S}') \geq w(\mathcal{S})$
$\mathcal{S}' \leftarrow \mathcal{S}$;
stop $\leftarrow$ false;
while $\neg$stop do
    stop $\leftarrow$ true;
    $\mathcal{N} \leftarrow ComputePotentialNeighbors(\mathcal{I}, \mathcal{S}')$;
    for $(s, u) \in \mathcal{N}$ do
        $\mathcal{S}'' \leftarrow (\mathcal{S} \setminus \{s\}) \cup \{u\}$;
        if $Feasible(\mathcal{S}'')$ then
            $\mathcal{S}' \leftarrow \mathcal{S}''$;
            stop $\leftarrow$ false;
            break;
        end
    end
end
return $\mathcal{S}'$;
```

In this respect, we define a neighborhood structure by first identifying all the pairs of jobs that represent *potential* neighboring solutions. For a given solution, i.e. an FSJ $\mathcal{S}$ of $\mathcal{J}$, a pair (s, u) with $s \in \mathcal{S}$ and $u \in \mathcal{J} \setminus \mathcal{S}$, represents the set of jobs obtained by substituting s by u in $\mathcal{S}$. Since substituting a job by another one with a lower or equal weight cannot lead to improving the quality of an FSJ in a single step, we only consider pairs (s, u) such that $w(u) > w(s)$.

For the sake of clarity, the way all pairs representing potential neighboring solutions are computed is depicted in Algorithm 4. As can be observed, the number of potential neighboring solutions is quadratic at most. Any of such pairs (s, u) giving rise to an actual solution, i.e., a set of jobs $\mathcal{S}' = (\mathcal{S} \setminus \{s\}) \cup \{u\}$ such that the instance $(\mathcal{S}', C)$ is feasible, are of higher quality than the original one. Indeed, it is easy to observe that $w(\mathcal{S}') = (w(\mathcal{S}) - w(s)) + w(u)$. Since $w(u) > w(s)$, it follows that $w(\mathcal{S}') > w(\mathcal{S})$.

The neighborhood structure described above is exploited by the proposed local search algorithm, which is shown in Algorithm 5. Given an FSJ $\mathcal{S}'$ of $\mathcal{J}$, the method considers all potential neighbors of $\mathcal{S}'$ until finding one representing an actual solution, which by definition is of higher quality than $\mathcal{S}'$. At this point, the new solution replaces $\mathcal{S}'$ and the process is repeated until no further improvements are achieved. Notice that the algorithm terminates either when the current solution does not have any potential neighboring solutions or when none of the potential neighbors results in an actual solution.

The local search algorithm, as described in Algorithm 5, follows a *hill climbing* search approach: at each iteration the first neighboring solution found improving the current one replaces it. This may lead to improvements smaller than possible, and so a *gradient descent* approach may be preferable, in which the current solution is replaced by the best neighboring solution improving it. Noticeably, Algorithm 5 can be easily instrumented to perfom such search by just considering the pairs $(s, u) \in \mathcal{N}$ in non-decreasing order of the values $w(u) - w(s)$. This way, the first actual neighboring solution found in the inner for loop is guaranteed to be the one with the highest quality. We will consider both approaches in the experimental study.

We remark that the local search algorithm has been devised in order to be integrated in the GA described in the previous section, so the feasibility tests are performed in the same way, by using the G&T algorithm guided by an operation sequence encoded in a chromosome. More details are given in the next section.

5 Memetic algorithm

Memetic algorihms are metaheuristics aiming at an adequate balance between exploration and intensification of the search by means of a combination of genetic algorithms and local search approaches [25]. In this setting, the GA is responsible for exploring diverse and promising regions in the search space, whereas local search intensifies the search in such regions, leading to higher quality solutions.

Herein we describe the memetic algorithm (MA) proposed for the problem of approximating weighted-maxFSJs of a given unfeasible problem instance. The proposed memetic algorithm combines the genetic algorithm described in Section 3 and the local search algorithm depicted in Algorithm 5. It follows the same evolutionary scheme as shown in Algorithm 3, but after the decoding phase is performed obtaning a solution to the problem, the local search method is applied to the computed solution, potentially improving its quality. If such improvement is achieved, the fitness value of the chromosome is updated with the total weight of the new solution.

As pointed out above, the procedure *Feasible* used by the local search algorithm when combined with GA is the same as the one used by the decoding algorithm of GA. This way, the operation sequence encoded in a chromosome corresponding to the solution to be improved by local search is used to guide the G&T schedule builder (Algorithm 2). If the schedule computed has a makespan not exceeding the maximum limit C, the given problem instance is declared feasible, while it is treated as unfeasible by Algorithm 5 otherwise. This way, the running times of the local search procedure are kept short.

6 Experimental results

We conducted an experimental study in order to assess the performance of the methods proposed in this work. To this aim, we coded a prototype in C++ and ran a series of experiments on a Linux cluster (Intel Xeon 2.26 GHz, 128 GB RAM).

The experiments were performed over a set of unfeasible instances derived from classical JSP instances from the *OR*-library [5] as well as some Taillard's instances [24]. Concretely, the benchmark set consists of instances with a different number of jobs $n \in \{10, 15, 20, 50\}$ and machines $m \in \{5, 10, 15, 20\}$. For each JSP instance, we created a new instance by assigning a random weight to each job, following a uniform distribution in the interval $[1, 100]$. Among the JSP instances considered, there are 21 instances with $n = 10$: LA01-05 ($m = 5$), FT10, ORB01-10 and LA16-20 ($m = 10$); 15 instances with $n = 15$: LA06-10 ($m = 5$), LA21-25 ($m = 10$) and LA36-40 ($m = 15$); 11 instances with $n = 20$: LA11-15, FT20 ($m = 5$) and LA26-30 ($m = 10$); and 20 instances with $n = 50$: tai50_15_01-10 ($m = 15$) and tai50_20_01-10 ($m = 20$). Finally, we built three unfeasible instances by imposing different values for the makespan limit C to be 70%, 80% and 90% of the optimal makespan C_{opt} for the instance. So, there are 201 instances in all.

Our goal is to evaluate the performance of the proposed methods. Bearing this in mind, we compare three algorithms: the original genetic algorithm (GA), and two memetic algorithms that use the local search approaches described in Section 4, namely a MA with hill climbing as local search approach (MA-HC), and a MA that uses gradient descent (MA-GD). We hypothesize that both MA-HC and MA-GD will reach better results than GA, and that they will take more time as well.

We divided the experimental analysis in two parts. First, we evaluate the effectiveness of the local search approaches. Then, we compare GA, MA-HC and MA-GD. The results are reported in the two following subsections.

6.1 Assessment of the local search

The objective of the first series of experiments is to evaluate the effectiveness of the local search approaches at improving solutions built by the solution builder (No LS), comparing the hill climbing (LS-HC) and the gradient descent (LS-GD) approaches.

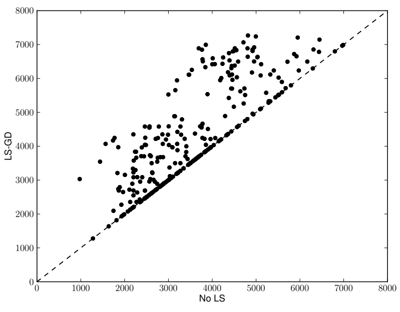
For this purpose, we generated 100 random solutions for three instances and solved them with the two local search methods. The chosen instances were LA16 ($n = 10$), LA21 ($n = 15$) and LA26 ($n = 20$), imposing a makespan limit C of 80% of their optimal makespan.

The results are presented in Table 2, which reports the value of the best and the average solutions, over the 100 considered, obtained by each method, averaged for each considered instance. As we can observe, for every instance, local search allows us to find (much) better solutions in average. When comparing the two local search approaches, we see that gradient descent improves random solutions to a greater extent than hill climbing. We can also see that the differences grow with the number of jobs, which suggests that local search is more effective for larger, and harder, instances.

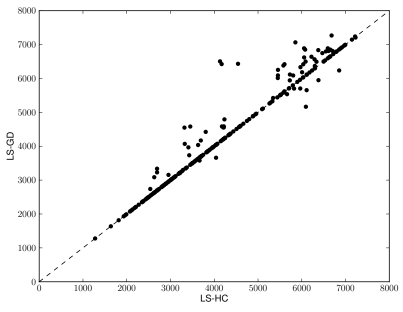
Figure 2(a) shows a scatter plot comparing No LS and LS-GD regarding the total weight of the computed solutions. The points that are on the main diagonal line represent random solutions that were not improved by LS-GD, whereas the points above this line represent solutions improved by LS-GD. Noticeably, many

Table 2 Comparison between No LS, LS-HC and LS-GD

	No LS		LS-HC		LS-GD	
Instance	Best	Avg.	Best	Avg.	Best	Avg.
LA16	3971	2631.41	3971	2884.98	3971	2905.33
LA21	4603	3131.64	4890	3761.05	4890	3831.33
LA26	6987	4743.60	7228	5809.46	7268	5971.19



(a) No LS vs. LS-GD



(b) LS-HC vs. LS-GD

Fig. 2 Scatter plots comparing the considered approaches.

of the solutions were improved by LS-GD, in many cases by a wide margin. On the other hand, Figure 2(b) shows a comparison between LS-HC and LS-GD. As we can observe, LS-GD is better than LS-HC in more cases, although the differences are not as significant as before. From these experiments we can conclude that the local search approaches are effective.

6.2 Assessment of the memetic algorithms

The following series of experiments was conducted to evaluate the performance of GA, MA-HC and MA-GD. In all the experiments, GA, MA-HC and MA-GD evolve a population of 100 individuals with crossover probability of 0.9 and mutation probability of 0.1 until a stopping criteria is met. The algorithm was run 20 times on each instance, recording the best and average solutions found, as well as the running times.

First, we carried out a series of experiments with the termination condition of completing 250 generations. The results of these experiments are shown in Table 3. For each number of jobs n, the table shows the error of the best and average solutions found over the 20 runs, as well as the running times of GA, MA-HC and MA-GD, averaged for each C. The average values for all the experiments is also shown in the last row of the table. The best value for each category is highlighted in bold. The errors are calculated w.r.t. the best solution found for each instance in all of the experiments.

Let us first compare GA with the two memetic algorithms. As can be observed, both MA-HC and MA-GD return better best and average values for all the sets of instances. We can point out as well that the difference seems to grow with the

Table 3 Summary of results after 250 generations.

		GA			MA-HC			MA-GD		
n	$\%C_{opt}$	Best	Avg.	Time (s)	Best	Avg.	Time (s)	Best	Avg.	Time (s)
10	70	0.55	1.23	0.50	**0.00**	**0.85**	0.66	**0.00**	0.90	0.66
	80	**0.10**	2.31	0.68	**0.10**	**1.71**	0.88	0.11	1.78	0.88
	90	0.07	2.00	0.85	**0.00**	1.31	1.04	0.02	**1.18**	1.03
	Avg.	0.24	1.85	0.67	**0.03**	1.29	0.86	0.05	**1.29**	0.86
15	70	**0.00**	2.32	1.55	0.44	1.84	2.28	**0.00**	**1.61**	2.27
	80	1.19	3.65	2.05	0.80	3.10	2.84	**0.35**	**2.97**	2.82
	90	1.59	3.71	2.57	**1.45**	3.40	3.41	1.55	**3.34**	3.37
	Avg.	0.92	3.23	2.06	0.90	2.78	2.84	**0.63**	**2.64**	2.82
20	70	1.87	3.91	3.39	1.51	3.55	4.98	**1.47**	**3.42**	4.83
	80	**1.17**	3.05	4.05	1.24	2.96	5.66	1.23	**2.76**	5.50
	90	0.92	1.91	4.58	**0.70**	1.82	6.13	0.99	**1.72**	5.99
	Avg.	1.32	2.96	4.01	**1.15**	2.77	5.59	1.23	**2.64**	5.44
50	70	2.73	7.58	63.86	1.20	3.62	138.83	**0.22**	**2.48**	125.02
	80	1.95	6.22	80.69	0.97	2.60	164.72	**0.14**	**1.69**	150.03
	90	1.31	4.54	95.94	0.53	1.70	177.18	**0.07**	**1.13**	165.33
	Avg.	2.00	6.11	80.16	0.90	2.64	160.24	**0.14**	**1.77**	146.79
Average		1.10	3.61	25.26	0.67	2.27	49.66	**0.40**	**1.95**	45.61

number of jobs. In addition, as expected, GA is less time consuming than MA-HC and MA-GD, especially for the large instances. Comparing the two memetic algorithms, MA-GD seems to return better best and average values than MA-HC, and also smaller running times. So, in average, there is no reason to doubt that MA-GD performs better than MA-HC. However, there are a few instances for which MA-HC reaches better solutions than MA-GD. Regarding how the methods behave depeding on the imposed C, it can be noticed that the running times grow with it.

Figure 3(a) shows a boxplot comparing the three methods. As can be observed, both MAs perform better than GA and MA-GD performs better than MA-HC.

The results above indicate that MA-HC and MA-GD return solutions of higher quality at the cost of increased running times. It is interesting to see how the algorithms compare if instead of stopping at a given number of generations, we have a time limit as stop condition. So, we carried out a new series of experiments, having a time limit of 60 seconds. Table 4 shows the results of these experiments. The results show that GA is still worse than MA-HC and MA-GD, and that MA-GD is still better than MA-HC, even though now the difference is smaller, mainly due to the small instances.

Figure 3(b) shows a boxplot summarizing the results in Table 4. If we compare it with Figure 3(a), we see that the algorithms rank in the same order, but now the difference is smaller.

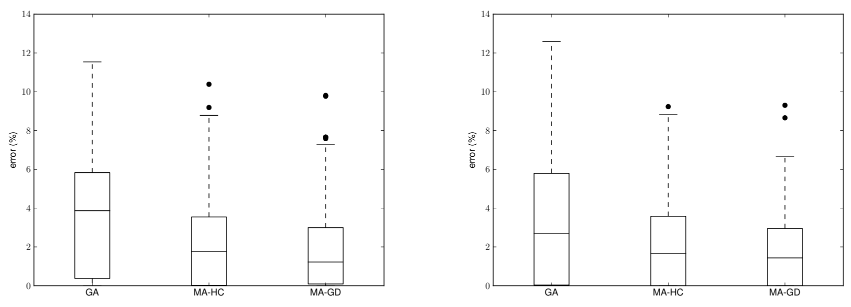
In all, we can conclude that the memetic algorithms perform better than the genetic algorithm, and that gradient descent seems to be a better local search strategy than hill climbing in most cases.

7 Conclusions

This paper addresses the task of repairing unfeasible job shop scheduling problems with a hard constraint on the maximum makespan allowed. To this aim, we con-

Table 4 Summary of results after 60s.

		GA		MA-HC		MA-GD	
n	$\%C_{opt}$	Best	Avg.	Best	Avg.	Best	Avg.
10	70	**0.00**	0.92	**0.00**	0.80	**0.00**	**0.72**
	80	0.06	1.64	**0.00**	**1.30**	0.04	1.42
	90	**0.00**	1.50	**0.00**	0.96	0.07	**0.96**
	Avg.	0.02	1.35	**0.00**	**1.02**	0.04	1.03
15	70	**0.00**	1.47	**0.00**	1.24	0.44	**1.09**
	80	0.31	2.64	0.59	2.34	**0.28**	**2.14**
	90	1.24	3.01	0.37	2.43	**0.21**	**2.39**
	Avg.	0.52	2.37	0.32	2.00	**0.31**	**1.87**
20	70	0.91	2.68	1.07	2.46	**0.04**	**2.09**
	80	0.65	2.13	**0.02**	1.74	0.41	**1.68**
	90	0.24	1.22	0.11	1.07	**0.00**	**1.04**
	Avg.	0.60	2.01	0.40	1.75	**0.15**	**1.60**
50	70	2.77	7.55	2.38	4.52	**1.30**	**3.21**
	80	2.67	6.73	1.84	3.43	**1.39**	**2.71**
	90	2.15	5.12	1.45	2.44	**0.89**	**1.94**
	Avg.	2.53	6.47	1.89	3.47	**1.19**	**2.62**
Average		0.98	3.22	0.70	2.09	**0.46**	**1.79**



(a) Generations (b) Time

Fig. 3 Boxplots comparing the considered approaches.

sider a setting in which the problem can be relaxed by dropping some of the jobs. In this context, we face the problem of computing a feasible subset of the jobs that maximizes their weighted sum. This problem generalizes the problem considered in [17], which aimed at maximizing cardinality. Building on the genetic algorithm proposed in [17], we propose a memetic algorithm that combines the GA with a local search method, also proposed in the paper. Experimental results show that the proposed approaches are effective at solving the problem, and that the memetic algorithm bring significant gains in the quality of the solutions computed.

As future work, we plan to study alternative solution builders for the problem, as well as alternative local search algorithms. Besides, considering hard constraints on metrics different from the makespan, such as the total tardiness [15] seems a promising line of research.

Acknowledgements

This research is supported by the Spanish Government under project TIN2016-79190-R and by the Principality of Asturias under grant IDI/2018/000176.

References

1. Allahverdi, A., Aydilek, H.: Total completion time with makespan constraint in no-wait flowshops with setup times. European Journal of Operational Research **238**(3), 724 – 734 (2014)
2. Ansótegui, C., Bonet, M.L., Levy, J.: SAT-based MaxSAT algorithms. Artif. Intell. **196**, 77–105 (2013)
3. Artigues, C., Lopez, P., Ayache, P.: Schedule generation schemes for the job shop problem with sequence-dependent setup times: Dominance properties and computational analysis. Annals of Operations Research **138**, 21–52 (2005)
4. Bailey, J., Stuckey, P.J.: Discovery of minimal unsatisfiable subsets of constraints using hitting set dualization. In: PADL, pp. 174–186 (2005)
5. Beasley, J.E.: Or-library: Distributing test problems by electronic mail. J Oper Res Soc **41**(11), 1069–1072 (1990)
6. Bierwirth, C.: A generalized permutation approach to job shop scheduling with genetic algorithms. OR Spectrum **17**, 87–92 (1995)
7. Brucker, P., Knust, S.: Complex Scheduling. Springer (2006)
8. Choi, J.Y.: Minimizing total weighted completion time under makespan constraint for two-agent scheduling with job-dependent aging effects. Computers & Industrial Engineering **83**, 237 – 243 (2015)
9. Dawande, M., Gavirneni, S., Rachamadugu, R.: Scheduling a two-stage flowshop under makespan constraint. Mathematical and Computer Modelling **44**(1), 73 – 84 (2006)
10. Garey, M., Johnson, D., Sethi, R.: The complexity of flowshop and jobshop scheduling. Mathematics of Operations Research **1**(2), 117 – 129 (1976)
11. Giffler, B., Thompson, G.L.: Algorithms for solving production scheduling problems. Operations Research **8**, 487–503 (1960)
12. Graham, R., Lawler, E., Lenstra, J., Kan, A.: Optimization and approximation in deterministic sequencing and scheduling: a survey. Annals of Discrete Mathematics **5**, 287 – 326 (1979)
13. Marques-Silva, J., Heras, F., Janota, M., Previti, A., Belov, A.: On computing minimal correction subsets. In: Proc. of IJCAI, pp. 615–622 (2013)
14. Marques-Silva, J., Janota, M., Mencía, C.: Minimal sets on propositional formulae. Problems and reductions. Artif. Intell. **252**, 22–50 (2017)
15. Mencía, C., Sierra, M.R., Mencía, R., Varela, R.: Evolutionary one-machine scheduling in the context of electric vehicles charging. Integrated Computer-Aided Engineering **26**, 49–63 (2019)
16. Mencía, C., Sierra, M.R., Varela, R.: Depth-first heuristic search for the job shop scheduling problem. Annals OR **206**(1), 265–296 (2013)
17. Mencía, R., Mencía, C., Varela, R.: Repairing infeasibility in scheduling via genetic algorithms. In: From Bioinspired Systems and Biomedical Applications to Machine Learning - 8th International Work-Conference on the Interplay Between Natural and Artificial Computation, IWINAC 2019, Almería, Spain, June 3-7, 2019, Proceedings, Part II, pp. 254–263 (2019)
18. Mencía, R., Sierra, M.R., Mencía, C., Varela, R.: Memetic algorithms for the job shop scheduling problem with operators. Appl. Soft Comput. **34**, 94–105 (2015)
19. Mencía, R., Sierra, M.R., Mencía, C., Varela, R.: Schedule generation schemes and genetic algorithm for the scheduling problem with skilled operators and arbitrary precedence relations. In: Proc. of ICAPS, pp. 165–173. AAAI Press (2015)
20. Nowicki, E., Smutnicki, C.: An advanced tabu search algorithm for the job shop problem. Journal of Scheduling **8**, 145–159 (2005)
21. Ono, I., Yamamura, M., Kobayashi, S.: A genetic algorithm for job-shop scheduling problems using job-based order crossover. In: Proceedings of 1996 IEEE International Conference on Evolutionary Computation, pp. 547–552 (1996)

22. Palacios, J.J., Vela, C.R., Rodríguez, I.G., Puente, J.: Schedule generation schemes for job shop problems with fuzziness. In: Proc. of ECAI, pp. 687–692 (2014)
23. Previti, A., Mencía, C., Järvisalo, M., Marques-Silva, J.: Premise set caching for enumerating minimal correction subsets. In: Proc. of AAAI, pp. 6633–6640 (2018)
24. Taillard, E.: Benchmarks for basic scheduling problems. European Journal of Operational Research **64**(2), 278–285 (1993)
25. Talbi, E.: Metaheuristics - From Design to Implementation. Wiley (2009)
26. Van Laarhoven, P., Aarts, E., Lenstra, K.: Job shop scheduling by simulated annealing. Operations Research **40**, 113–125 (1992)
27. Zhang, C.Y., Li, P., Rao, Y., Guan, Z.: A very fast TS/SA algorithm for the job shop scheduling problem. Computers and Operations Research **35**, 282–294 (2008)