Efficient repairs of infeasible job shop problems by evolutionary algorithms

Raúl Mencía[a,*], Carlos Mencía[a], Ramiro Varela[a]

[a] *Department of Computer Science, University of Oviedo, Spain*

Abstract

We address the task of repairing infeasibility in the context of infeasible job shop scheduling problems with a hard constraint on the maximum makespan allowed. For this purpose, we adopt a job-based view of repairs, that allows for dropping some of the jobs and so gives rise to the problem of computing the largest subset of jobs that can be scheduled under the makespan constraint. Recent work proposed a genetic algorithm for solving this problem, which integrates an efficient solution builder for defining the search space. In this paper, we build on this earlier work and make several contributions. We provide a formal analysis of both the search space and the solution builder. Then, we propose two important enhancements to the genetic algorithm: first, we develop a new solution builder aimed at reducing the number of feasibility tests, making the search process more efficient. In addition, we propose a more effective procedure for testing the feasibility of different subsets of jobs under the given makespan constraint based on the use of a light-weight genetic algorithm. Experimental results show that the proposed methods are effective at solving the problem, and that the enhancements bring significant improvements.

Keywords: Job Shop Scheduling, Infeasibility, Repairs, Evolutionary Algorithms, Solution Builders

---

1. Introduction

Scheduling problems arise profusely in a wide range of areas, especially in manufacturing and engineering, where a proper organization of limited resources is usually necessary. In addition, these problems are often computationally intractable, what makes them a natural application domain for advanced artificial intelligence and operations research methods. As a consequence, scheduling problems have been thoroughly studied in the literature over the last decades, giving rise to a large body of solving approaches, both exact and approximate, e.g. (Brucker et al., 1994; Nowicki and Smutnicki, 2005; Beck, 2007; Zhang et al., 2008; Mencía et al., 2015; Peng et al., 2015; Deng et al., 2020).

*Corresponding author

Email address:

Scheduling problems usually require computing schedules that optimize a given objective function. However, in some settings there can be constraints that make the problem infeasible, i.e., with no possible solution whatsoever. Such scenario may appear when, for instance, considering a hard constraint that imposes a limit on the maximum makespan allowed, enforcing all the jobs in the problem to be scheduled (and completed) by a given time limit. This kind of constraint is natural in practice, and infeasibility may easily arise under this constraint if the given scheduling horizon is too tight. Furthermore, a number of scheduling problems with a constraint on the makespan have been studied in the past, e.g. (Dawande et al., 2006; Allahverdi and Aydilek, 2014; Guyon et al., 2014; Choi, 2015)). In this context, beyond detecting infeasibility, users may be interested in identifying its causes, or in finding possible ways of *repairing* it, so that being able to solve the problem to some extent.

In this paper, we address the task of repairing infeasible job shop scheduling problems with a hard constraint on the makespan. For this purpose, we adopt a *job-based* view of repairs, which enables dropping some of the jobs so that the remaining ones can be scheduled within the makespan constraint. This view was recently taken in (Mencía et al., 2019), giving rise to different notions of repairs, such as feasible subsets of jobs (FSJs), set-wise maximal feasible subsets of jobs (MFSJs) and feasible subsets of jobs of maximum cardinality (maxFSJs), i.e., feasible subsets of jobs with the greatest possible number of jobs. These concepts are inspired in analogous notions in the analysis of inconsistency in logic, where repairing (or correcting) inconsistent formulas has been widely investigated (Marques-Silva and Mencía, 2020). In addition, Mencía et al. (2019) preliminarily addressed the problem of approximating maxFSJs, by means of a genetic algorithm. This algorithm looks for solutions in the search space of (approximations of) MFSJs, defined by a solution builder used in its decoding phase. The genetic algorithm was recently adapted to a weighted version of the problem and combined with a local search algorithm specifically tailored to the case that jobs have different weights (Mencía et al., 2020).

We focus on the problem of approximating maxFSJs, that is, computing the largest subsets of jobs that can be scheduled under the hard makespan constraint. Building on (Mencía et al., 2019), we make several contributions:

- We first provide a formal analysis of the search space of MFSJs and the solution builder, proving relevant properties.

- The analysis gives rise to important enhancements to the genetic algorithm. In this respect, we propose a new solution builder that defines the same search space but in a more efficient way. By integrating a binary search phase, it aims at reducing the number of feasibility tests needed to compute a solution, what allows for saving time.

- Furthermore, we propose a new (incomplete) procedure for testing the feasibility of different subsets of the jobs (i.e. deciding whether these can be scheduled within the given makespan limit), which is iteratively invoked by the solution builders. Whereas earlier work used a greedy algorithm for this purpose, the new procedure integrates a light-weight genetic algorithm which is more effective, leading to better results.

We conducted an extensive experimental study to evaluate the proposed algorithms over a large set of instances with different sizes and characteristics. The experimental results reveal that genetic algorithms are successful at solving the problem and confirm that the new methods bring significant improvements in practice.

The remainder of the paper is structured as follows: Section 2 introduces the necessary background and notation, including the definition of the problem. Section 3 reviews related work. Section 4 presents a formal analysis of the search space and the solution builder from (Mencía et al., 2019), and describes the new solution builder that exploits binary search. The main components of the genetic algorithm are presented in Section 5. The new procedure for testing the feasibility of a given subset of jobs is described in Section 6. Section 7 is devoted to the experimental study. Finally, we summarize the main conclusions and outline ideas for future research in Section 8.

2. Preliminaries

In the classical job shop scheduling problem (JSP) we are given a set of n jobs $\mathcal{J} = \{J_1, \ldots, J_n\}$ that must be scheduled on a set of m resources or machines $\mathcal{M} = \{M_1, \ldots, M_m\}$. Job $J_i \in \mathcal{J}$ consists of a sequence of m tasks or operations $(\theta_{i1}, \ldots, \theta_{im})$, and each of its operations θ_{ij} requires a particular machine $M(\theta_{ij}) \in \mathcal{M}$ during a (positive integer) processing time $p_{\theta_{ij}}$. Besides, without loss of generality, it is commonly assumed that any two operations of the same job require different machines.

A schedule S is an allocation of a starting time $st_{\theta_{ij}}$ to each operation satisfying the following constraints:

i. **Conjunctive constraints**. Operations must be scheduled in the order they appear in their job, i.e., $st_{\theta_{ij}} + p_{\theta_{ij}} \leq st_{\theta_{i(j+1)}}$ for all $i = 1, ..., n$ and $j = 1, ..., m-1$.
ii. **Disjunctive constraints**. Machines cannot process more than one operation at a time, that is, $(st_u + p_u \leq st_v) \vee (st_v + p_v \leq st_u)$ for all operations u, v with $u \neq v$ and $M(u) = M(v)$.
iii. **Non-preemption**. The processing of operations cannot be interrupted, i.e., $C_u = st_u + p_u$ for every operation u, where C_u denotes the completion time of u.

The quality of a schedule can be evaluated with respect to several different metrics (Brucker and Knust, 2006), such as the *makespan*, *total flow time* or *tardiness* and *lateness* measures (when due dates are considered), to name a few. In this work, we focus on the makespan: given schedule S, its makespan is defined as the maximum completion time of the operations in S, and it is denoted $C_{max}(S)$.

This metric gives rise to one of most studied optimization versions of the JSP, denoted $J||C_{max}$ in the standard $\alpha|\beta|\gamma$ notation (Graham et al., 1979)), which requires computing a schedule with the minimum possible makespan.

Taking the makespan into account, the decision version of the JSP requires determining 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, we call the problem instance *feasible*, whereas *infeasible* otherwise. The JSP, in its decision version, is well-known to be NP-complete (Garey et al., 1976).

This paper focuses on infeasible problem instances due to a hard constraint on the makespan and, more specifically, on the task of repairing infeasibility in the best possible manner. To this aim, we adopt a *job-based* view of repairs, that enables relaxing the problem by dropping some of the jobs in a way that the remaining ones can be scheduled within the makespan limit imposed by the hard constraint. Throughout, we will refer to such an infeasible instance by a pair $\mathcal{I} = (\mathcal{J}, C)$, requiring scheduling the set of jobs $\mathcal{J}$ without exceeding the maximum makespan C.

In this setting, the following definitions serve to characterize different notions of repairs (Mencía et al., 2019):

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

An FSJ is a subset of jobs that can be scheduled under the makespan constraint. Thus, it represents a possible way of repairing infeasibility. However, an arbitrary FSJ may leave out a number of jobs unnecessarily, and so considering some maximality criteria would be beneficial.

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

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

MFSJs and maxMFSJs exhibit different forms of maximality. On the one hand, MFSJs are maximal with respect to set inclusion, that is, no superset of an MFSJ is an FSJ. On the other hand, maxFSJs are maximal with respect to set cardinality, i.e., maxFSJs are the largest possible FSJs. As we will see in the following section, maxFSJs are MFSJs as well, but the opposite does not always hold true. Arguably, MFSJs represent a kind of local optima with respect to approximating maxFSJs, since these cannot be extended with any jobs without losing feasibility.

These definitions build on related concepts in the analysis of inconsistent propositional formulas, such as *maximal satisfiable subformulas* (MSSes) or *maximum satisfiability* (maxSAT), representing analogous concepts to MFSJs and maxFSJs respectively (see (Marques-Silva and Mencía, 2020) for a recent survey).

The following example (Mencía et al., 2019) illustrates all the definitions:

Example 1. *Let us consider a job shop with jobs $\mathcal{J} = \{J_1, J_2, J_3, J_4\}$ and machines $\mathcal{M} = \{M_1, M_2\}$. Each job J_i consists of a sequence of two operations $(\theta_{i1}, \theta_{i2})$, with processing times and machine requirements as indicated in Table 1 (e.g., job J_1 consists of the sequence of operations $(\theta_{11}, \theta_{12})$; θ_{11} requires machine M_1 during 2 time units, and θ_{12} requires machine M_2 during 3 time units, ...).*

If we consider a hard constraint limiting the makespan to at most $C = 10$ the instance $(\mathcal{J}, C)$ is infeasible. 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. Out of these

Table 1: Instance data.

	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)

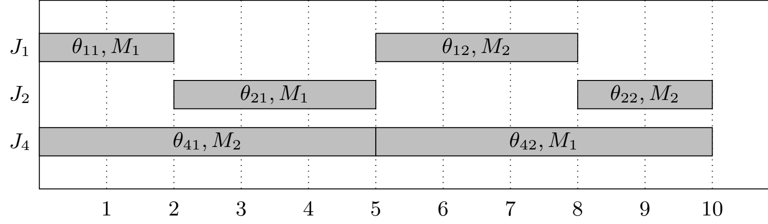


Figure 1: Gantt chart of a schedule for the maxFSJ $\{J_1, J_2, J_4\}$ in Example 1.

sets, three are MFSJs: $\{J_1, J_3\}$, $\{J_2, J_3\}$ *and* $\{J_1, J_2, J_4\}$*; and only the last one is a maxFSJ, with 3 jobs. Figure 1 shows a schedule for the maxFSJ with makespan 10, which does not exceed the limit* C*.*

With the definitions above, computing maxFSJs may be seen as the best way of repairing infeasibility, due to their maximum size. Hence, in this paper we address the maximization problem of approximating maxFSJs for a given infeasible instance $(\mathcal{J}, C)$, that is finding feasible subsets of jobs with maximum cardinality.

Table 2 summarizes the main notation and terminology introduced in this section. It will be used throughout the paper.

Table 2: Main notation and terminology.

Notation	Definition
JSP	Job shop scheduling problem
$\mathcal{M}$	Set of machines
M_i	i-th machine
$\mathcal{J}$	Set of jobs
J_i	i-th job
θ_{ij}	j-th operation of job J_i
$st_{\theta_{ij}}$	Starting time of operation θ_{ij}
$p_{\theta_{ij}}$	Processing time of operation θ_{ij}
C_u	Completion time of operation u
$C_{max}(S)$	Makespan of schedule S
C	Maximum makespan
$(\mathcal{J}, C)$	Problem instance, with jobs $\mathcal{J}$ and maximum makespan C
FSJ	Feasible subset of jobs
MFSJ	Maximal feasible subset of jobs
maxFSJ	Maximum feasible subset of jobs

3. Related work

The problem studied in this paper was first addressed in (Mencía et al., 2019). In this previous work, the notions of MFSJ and maxFSJ were proposed as a means to repairing infeasibility in the context of job shop scheduling problems with a hard constraint on the makespan. These definitions are based on concepts commonly used in the field of Boolean satisfiability in the analysis of unsatisfiable propositional formulas. For approximating maxFSJs, Mencía et al. (2019) proposed a genetic algorithm that looks for solutions in the space of (approximations of) MFSJs. To this aim, it integrates a solution builder based on a *linear search* approach (Bailey and Stuckey, 2005; Marques-Silva et al., 2013) and uses a greedy algorithm (Giffler and Thompson, 1960) as an incomplete procedure to test the feasibility of subsets of jobs. To our knowledge, this genetic algorithm is the current best performing approach for solving the problem. This algorithm was later adapted to handle a weighted version of the problem and combined with a local search approach (Mencía et al., 2020). The local search approach is only applicable when jobs have different weights, so it cannot bring any benefits to the unweighted version of the problem considered herein. In contrast, improvements to the core components of the genetic algorithm can be expected to be effective in the weighted version of the problem as well.

In this paper, we build on this earlier work and make significant contributions, in both theory and practice. First, we provide a detailed formal analysis proving relevant properties, as the completeness of the search space of MFSJs, and the soundness of the solution builder. In addition, we propose two key enhancements to the genetic algorithm: a new, more efficient, solution builder and a new procedure to test the feasibility of subsets of jobs more effectively. Despite restricting our study to the task of repairing infeasible job shop scheduling problems with a hard constraint on the makespan, the scope and applicability of this framework should be underscored. The same notions for repairing infeasible problem instances could be applied to other versions of job shop scheduling (e.g., considering setup times, uncertainty or additional constraints on the resources) or to other scheduling problems where the input is a set of jobs. Furthermore, the hard constraint that makes the problem instance infeasible could be defined on metrics other than the makespan. In any case, computing MFSJs and maxFSJs may be a suitable approach for dealing with infeasibility. In addition, the methodology in the formal analysis and the algorithms proposed herein could serve as a solid basis to tackle such problems in the future.

In the remainder of this section, we first discuss some related problems and then review genetic algorithms for solving scheduling problems.

3.1. Related problems

The problem of computing maxFSJs addressed in this paper is related to other problems studied in the field of scheduling.

For example, Dawande et al. (2006) considered the problem of finding a feasible subset of jobs in a two-stage flow shop maximizing a weighted sum of the jobs. However, testing the feasibility of a two-stage flow shop can be done in polynomial time (Garey et al., 1976), whereas testing the feasibility of general job shop instances is an NP-complete problem.

Another related problem is the one addressed in (Della Croce et al., 2017), which considers polynomially solvable two machine flow shop and job shop problems, and focuses on selecting the set of jobs of a given fixed size (introduced as a parameter) that optimizes the makespan. In contrast, we focus on optimizing the size of the selected set of jobs, while fixing the limit on the maximum makespan allowed.

The problem addressed herein can be also related to the classical objective function of minimizing the number of late, or tardy, jobs (Moore, 1968; Della Croce et al., 2000). In this setting, each job is associated a due date, and the goal is to compute a schedule that minimizes the number jobs that are completed after their indicated due dates. This objective function plays an important role in the context of overloaded single-machine real-time systems (Liao et al., 2019), where the goal is completing the maximum number of tasks on time. In this paper, our goal is not computing a schedule that optimizes a given objective function, but identifying the largest subset of jobs that can be scheduled under the given additional hard constraint, removing any other jobs that do not take part in the solution.

3.2. Genetic algorithms

Genetic algorithms (GAs) stand out among the most successful population-based metaheuristics (Talbi, 2009). GAs were originally proposed by Holland (1975) and are inspired in the theory of evolution. These algorithms maintain a population of solutions which is evolved by the application of selection, recombination and replacement operators, with the goal of reaching an appropriate tradeoff between exploration of the search space and intensification in its most promising regions.

Genetic algorithms have been widely used to solve a variety of hard scheduling problems, including single machine (Mustu and Eren, 2018; Mencía et al., 2019), parallel machines (Vallada and Ruiz, 2011; Tan et al., 2019), open shop (Andresen et al., 2008; Hosseinabadi et al., 2019), flow shop (Branda et al., 2021), job shop (Gonçalves and Resende, 2014; Zhang et al., 2020) or resource constrained project scheduling problems (Gonalves et al., 2008). In addition, GAs have been successfully applied in numerous domains, as heterogeneous computing systems Akbari et al. (2017) or operations management (Lee, 2018), to name a few.

A large body of genetic algorithms has been proposed for solving job shop scheduling problems in their optimization versions. The performance of these algorithms relies on coding schemas specific to the problem, as well as on the use of sophisticated crossover operators. Some examples of coding schemas are those based on preference rules (Della Croce et al., 1995), permutations of jobs with repetitions (Bierwirth, 1995) or random keys (Gonçalves and Resende, 2014). Crossover operators play an important role in the evolutionary process, since these allow for an effective transmission of relevant characteristics from parents to their offspring. Examples of crossover operators include order-based crossover (Davis, 1985), multi-step crossover (Yamada and Nakano, 1995) or job-based order crossover (Ono et al., 1996), among others.

In order to decode chromosomes into actual schedules, GAs commonly exploit *schedule builders*. These are methods that allow for restricting the search to subsets of all possible schedules. For instance, the well-known G&T algorithm (Giffler and Thompson, 1960) defines the space of the so-called *active* schedules. This algorithm has been extended to cope with variants of the job shop scheduling problem that include

additional elements and constraints, for instance sequence-dependent setup times (Artigues et al., 2005) uncertainty in the processing times (Palacios et al., 2014), or skilled operators that assist the processing of the operations (Mencía et al., 2015).

Genetic algorithms have also been successfully combined with other metaheuristics, as local search methods. For example, Meeran and Morshed (2012) combined a genetic algorithm with tabu search to solve real-life job shop scheduling problems. Kurdi (2015) proposed a hybrid genetic algorithm which incorporates a self-adaptation strategy based on tabu search and random mutation operators. Hybrid genetic algorithms have also been successful in different variants of the problem, as dynamic (Kundakc and Kulak, 2016) or multiobjective (Gong et al., 2019) job shop scheduling problems, among others.

4. Search space

The definition of a search space is a fundamental step towards solving hard combinatorial problems. Ideally, the search space would exhibit some desirable properties, such as being of reasonable size and containing high-quality solutions to the problem, including optimal ones.

When facing scheduling problems, the use of so-called *schedule builders* is common for this purpose, e.g. (Giffler and Thompson, 1960; Kolisch, 1996; Artigues et al., 2005; Palacios et al., 2014; Mencía et al., 2015). Schedule builders, also referred to as *schedule generation schemes*, are constructive methods for computing and enumerating a subset of the schedules for a given problem instance, and so enable the definition of a search space.

However, for solving the problem considered herein, i.e., approximating maxFSJs for a given infeasible problem instance, computing schedules alone does not suffice, since it is also necessary to identify feasible subsets of jobs. As a consequence, the definition of the search space needs to be done in two levels: first on the subset space of the set of all the jobs, and then on the set of schedules for any given subset of jobs in order to test its feasibility. We describe both in the following subsections.

4.1. Subset space of the set of jobs

Our approach aims at restricting the search to the set of MFSJs, what comes with several benefits. First, the search space defined by the set of all MFSJs is complete, i.e., it contains all optimal solutions to any given problem instance. This follows from the fact that all maxFSJs are MFSJs as well, as proven next.

Proposition 1. *Let $\mathcal{I} = (\mathcal{J}, C)$ be an infeasible problem instance and $\mathcal{S}^* \subsetneq \mathcal{J}$ a maxFSJ of $\mathcal{J}$. $\mathcal{S}^*$ is also an MFSJ of $\mathcal{J}$.*

Proof. $\mathcal{S}^*$ is a maxFSJ of $\mathcal{J}$ so, by Definition 3 it is an FSJ of $\mathcal{J}$. Suppose that $\mathcal{S}^*$ is not an MFSJ of $\mathcal{J}$. Then, by Definition 2, there must exist a proper superset $\mathcal{S}' \subsetneq \mathcal{J}$ of $\mathcal{S}^*$ which is also an FSJ of $\mathcal{J}$. As $\mathcal{S}^* \subsetneq \mathcal{S}'$, it necessarily follows that $|\mathcal{S}^*| < |\mathcal{S}'|$, so $\mathcal{S}^*$ is not a maxFSJ of $\mathcal{J}$. A contradiction. □

FSJs exhibit a useful monotonicity property that will be used throughout. We prove that all the subsets of an FSJ are feasible subsets of jobs as well.

Proposition 2. *Let $\mathcal{I} = (\mathcal{J}, C)$ be an infeasible 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 $\mathcal{J}$.*

Proof. Since $\mathcal{S}$ is an FSJ of $\mathcal{J}$, there exists a schedule S for the jobs in $\mathcal{S}$ with $C_{max}(S) \leq C$. Given $\mathcal{S}' \subseteq \mathcal{S}$, we can build a schedule S' by assigning each operation in $\mathcal{S}'$ the same starting time as in S. S' is a schedule for the jobs in $\mathcal{S}'$ and $C_{max}(S') \leq C_{max}(S) \leq C$, as the operations in S' are a subset of those in S. Thus, by Definition 1, $\mathcal{S}'$ is an FSJ of $\mathcal{J}$. □

A consequence of Proposition 2 is the dual result that all the supersets of an infeasible set of jobs are infeasible too, as proven next:

Proposition 3. *Let $\mathcal{I} = (\mathcal{J}, C)$ be an infeasible problem instance and let $\mathcal{U} \subseteq \mathcal{J}$ be such that the instance $(\mathcal{U}, C)$ is infeasible. Then, for all $\mathcal{U}'$ with $\mathcal{U} \subseteq \mathcal{U}' \subseteq \mathcal{J}$, the instance $(\mathcal{U}', C)$ is infeasible.*

Proof. Consider an arbitrary $\mathcal{U}'$ such that $\mathcal{U} \subseteq \mathcal{U}' \subseteq \mathcal{J}$. Since, the instance $(\mathcal{U}, C)$ is infeasible and $\mathcal{U} \subseteq \mathcal{U}'$, not all the subsets of $\mathcal{U}'$ are FSJs. By the contrapositive of Proposition 2 it follows that that $\mathcal{U}'$ is not an FSJ of $\mathcal{J}$, i.e., the instance $(\mathcal{U}', C)$ is infeasible. □

It is clear that the number of MFSJs for any given problem instance is never greater than the number of FSJs since, by definition, all MFSJs are FSJs too. However, Proposition 2 allows for easily proving that the number of FSJs can be exponentially greater than that of MFSJs.

Proposition 4. *There are infeasible problem instances with a number of FSJs exponentially greater than the number of MFSJs.*

Proof. Consider a feasible problem instance $(\mathcal{S}, C)$ and a job $j \notin \mathcal{S}$ such that the sum of the processing times of its operations exceeds C. The job j alone cannot be scheduled under the makespan constraint, i.e., the instance $(\{j\}, C)$ is infeasible. Now define the instance $(\mathcal{J}, C)$, with $\mathcal{J} = \mathcal{S} \cup \{j\}$. By Proposition 3, this instance is infeasible since $\{j\} \subseteq \mathcal{J}$. The set $\mathcal{S}$ is the only MFSJ of $\mathcal{J}$, whereas by Proposition 2 each set in the power set of $\mathcal{S}$ is an FSJ of $\mathcal{J}$, that is, there are $2^{|\mathcal{S}|}$ FSJs of $\mathcal{J}$. □

Noticeably, the monotonicity properties stated above enable an alternative definition of MFSJs:

Proposition 5. *Let $\mathcal{I} = (\mathcal{J}, C)$ be an infeasible problem instance. $\mathcal{S} \subsetneq \mathcal{J}$ is an MFSJ of $\mathcal{J}$ if and only if $(\mathcal{S}, C)$ is feasible and for all $j \in \mathcal{J} \setminus \mathcal{S}$, $(\mathcal{S} \cup \{j\}, C)$ is infeasible.*

Proof. (If) The instance $(\mathcal{S}, C)$ is feasible, so $\mathcal{S}$ is an FSJ of $\mathcal{J}$. Let us suppose that $\mathcal{S}$ is not an MFSJ of $\mathcal{J}$. Then, there must exist an FSJ $\mathcal{S}' \subsetneq \mathcal{J}$ such that $\mathcal{S} \subsetneq \mathcal{S}'$. As $\mathcal{S}'$ is a proper superset of $\mathcal{S}$, $\mathcal{S}'$ must contain some job $j \in \mathcal{J} \setminus \mathcal{S}$. Since $(\mathcal{S} \cup \{j\}, C)$ is infeasible for all $j \in \mathcal{J} \setminus \mathcal{S}$, by Proposition 3, $(\mathcal{S}', C)$ is necessarily infeasible. A contradiction.

(Only if) $\mathcal{S}$ is an MFSJ of $\mathcal{J}$ so, by Definition 2, for all $\mathcal{S}' \subseteq \mathcal{J}$ such that $\mathcal{S} \subsetneq \mathcal{S}'$, the instance $(\mathcal{S}', C)$ is infeasible. Hence, for all $j \in \mathcal{J} \setminus \mathcal{S}$, $(\mathcal{S} \cup \{j\}, C)$ is infeasible. □

Algorithm 1 Solution Builder based on Linear Search.

Data: Set of jobs $\mathcal{J}$, makespan limit C
Result: $\mathcal{S} \subsetneq \mathcal{J}$ an MFSJ of $\mathcal{J}$

```
R ← J;
S ← ∅;
U ← ∅;
while R ≠ ∅ do
    Pick j ∈ R; // Non-deterministically
    R ← R \ {j};
    if Feasible (S ∪ {j}, C) then
        S ← S ∪ {j};
    else
        U ← U ∪ {j};
end
return S;
```

Based on the result above, a solution builder was proposed in (Mencía et al., 2019) for computing an MFSJ of $\mathcal{J}$, which requires a linear number of feasibility tests.

4.1.1. Solution builder

Algorithm 1 shows the solution builder[1]. Given an infeasible problem instance $(\mathcal{J}, C)$, it produces an MFSJ of $\mathcal{J}$ by following a *linear search* approach (Bailey and Stuckey, 2005; Marques-Silva et al., 2013).

The algorithm maintains a partition $\{\mathcal{S},\mathcal{U},\mathcal{R}\}$ of $\mathcal{J}$, where $\mathcal{S}$ represents a feasible subset of jobs, $\mathcal{U}$ contains jobs j such that $(\mathcal{S} \cup \{j\}, C)$ is infeasible, and $\mathcal{R}$, referred to as *reference set*, includes the jobs that still need to be tested. At the beginning, $\mathcal{R} = \mathcal{J}$ and both $\mathcal{S}$ and $\mathcal{U}$ are initialized as empty sets. Then, iteratively until $\mathcal{R}$ becomes empty, the algorithm selects a job $j \in \mathcal{R}$ and tests whether the instance $(\mathcal{S} \cup \{j\}, C)$ is feasible. If it is, $\mathcal{S}$ is extended with the job j; otherwise, j is added to $\mathcal{U}$. In either case, j is removed from $\mathcal{R}$, which guarantees termination after $|\mathcal{J}|$ iterations. Note that when the algorithm terminates, it holds that $\mathcal{U} = \mathcal{J} \setminus \mathcal{S}$.

Algorithm 1 uses a procedure, termed Feasible, to test the feasibility of subsequent subsets of jobs under the given hard constraint on the makespan. More concretely, given a set of jobs $\mathcal{J}'$, and a limit C on the maximum makespan allowed, Feasible $(\mathcal{J}', C)$ returns the Boolean value true if there exists a schedule for the jobs in $\mathcal{J}'$ with makespan not exceeding C. If such a schedule does not exist, the procedure returns the value false. In this section, Feasible acts as an abstract procedure, to make the formal analysis general. Later in the paper, Section 5 and Section 6 detail specific implementations of this procedure.

Noticeably, whenever Feasible is a complete decision procedure, there is the guarantee that upon termination $\mathcal{S}$ represents an actual MFSJ of $\mathcal{J}$. We prove this result as

[1]For the sake of clarity, Algorithm 1 extends the pseudocode in (Mencía et al., 2019) by considering the set $\mathcal{U}$, which allows for proving soundness more easily (Proposition 6).

follows:

Proposition 6. *Given an infeasible problem instance $\mathcal{I} = (\mathcal{J}, C)$, Algorithm 1 always computes an MFSJ of $\mathcal{J}$ if* Feasible *is a complete decision procedure.*

Proof. It suffices to note that the following invariant holds: $\mathcal{S}$ is an FSJ of $\mathcal{J}$ and for every $j \in \mathcal{U}$, it holds that for all FSJs $\mathcal{S}'$ of $\mathcal{J}$ such that $\mathcal{S} \subseteq \mathcal{S}'$, the instance $(\mathcal{S}' \cup \{j\}, C)$ is infeasible. Since $\mathcal{S}$ is only extended with a job j if $(\mathcal{S} \cup \{j\})$ is feasible, $\mathcal{S}$ is guaranteed to be an FSJ of $\mathcal{J}$ along the whole process. On the other hand, the job j is added to the set $\mathcal{U}$ whenever $(\mathcal{S} \cup \{j\}, C)$ is infeasible. Now, let $\mathcal{S}'$ be an FSJ such that $\mathcal{S} \subseteq \mathcal{S}'$ and a job $j \in \mathcal{U}$. By Proposition 3, since $(\mathcal{S} \cup \{j\}, C)$ is infeasible and $\mathcal{S} \cup \{j\} \subseteq \mathcal{S}' \cup \{j\}$, it follows that the instance $(\mathcal{S}' \cup \{j\}, C)$ is infeasible. On termination $\mathcal{U} = \mathcal{J} \setminus \mathcal{S}$; hence by Proposition 5 $\mathcal{S}$ is an MFSJ of $\mathcal{J}$, since $(\mathcal{S}, C)$ is feasible and for all $j \in \mathcal{J} \setminus \mathcal{S}$, the instance $(\mathcal{S} \cup \{j\}, C)$ is infeasible. □

As shown in the pseudocode, at each iteration Algorithm 1 picks a job $j \in \mathcal{R}$ non-deterministically. Depending on the choices made, different MFSJs can be computed. For example, for the instance from Example 1, if led by the sequence of choices (J_1, J_2, J_3, J_4) the algorithm would compute the MFSJ $\{J_1, J_2, J_4\}$. On the other hand, the sequence of choices (J_1, J_3, J_2, J_4) would yield the MFSJ $\{J_1, J_3\}$. The sequence of choices (J_3, J_1, J_4, J_2) would also lead to building the MFSJ $\{J_1, J_3\}$, so the mapping is many-to-one.

Noticeably, there is no MFSJ that cannot be computed by the solution builder. This follows from the next result:

Proposition 7. *Let $\mathcal{S}$ be an MFSJ of $\mathcal{J}$ for a given infeasible instance $\mathcal{I} = (\mathcal{J}, C)$. There is a sequence of choices $\sigma = (\sigma_1, ..., \sigma_{|\mathcal{J}|})$ that leads Algorithm 1 to computing $\mathcal{S}$.*

Proof. Define $\sigma_{\mathcal{S}}$ as a permutation of the jobs in $\mathcal{S}$, and $\sigma_{\mathcal{J} \setminus \mathcal{S}}$ as a permutation of the remaining jobs. Build $\sigma = \sigma_{\mathcal{S}} \sigma_{\mathcal{J} \setminus \mathcal{S}}$, by appending both permutations. If led by σ, Algorithm 1 would build the MFSJ $\mathcal{S}$, since it would pick the jobs in this set before any other jobs. □

This, together with the fact that the solution builder is sound (Proposition 6), implies that when Feasible is a complete decision procedure the search space defined by the solution builder contains exactly the set of all MFSJs for any given problem instance, including all maxFSJs as well.

4.1.2. Reducing the number of feasibility tests

As pointed out, the solution builder in Algorithm 1 performs a linear number of feasibility tests for computing an MFSJ. Although this number is rather low, the solution builder is to be invoked a large number of times (as it will act as the decoder of a genetic algorithm). So, we propose an alternative method that aims at reducing the number of feasibility tests. The idea is to first conduct a binary search in order to obtain an initial under-approximation quickly, and then extend it to an actual MFSJ by performing a linear search on the remaining jobs.

Algorithm 2 Solution Builder with Binary Search.

Data: Set of jobs $\mathcal{J}$, makespan limit C
Result: $\mathcal{S} \subsetneq \mathcal{J}$ an MFSJ of $\mathcal{J}$
$low \leftarrow 0$;
$up \leftarrow |\mathcal{J}|$;
while $(up - low) > 1$ **do**
  $mid \leftarrow \lfloor (low + up)/2 \rfloor$;
  if Feasible $(\mathcal{J}_{1..mid}, C)$ **then**
    $low \leftarrow mid$;
  else
    $up \leftarrow mid$;
end
$\mathcal{S} \leftarrow \mathcal{J}_{1..low}$;
$\mathcal{U} \leftarrow \mathcal{J}_{up..up}$;
$\mathcal{R} \leftarrow \mathcal{J}_{(up+1)..|\mathcal{J}|}$;
while $\mathcal{R} \neq \emptyset$ **do**
  Pick the first job $j \in \mathcal{R}$;
  $\mathcal{R} \leftarrow \mathcal{R} \setminus \{j\}$;
  if Feasible $(\mathcal{S} \cup \{j\}, C)$ **then**
    $\mathcal{S} \leftarrow \mathcal{S} \cup \{j\}$;
  else
    $\mathcal{U} \leftarrow \mathcal{U} \cup \{j\}$;
end
return $\mathcal{S}$;

The alternative solution builder is depicted in Algorithm 2. This algorithm assumes a fixed order of the jobs in the sets, and uses subindices to refer to specific subsets: $\mathcal{J}_{i..j}$ refers to the set that consists of the elements from the i-th, to the j-th of $\mathcal{J}$ whenever $i < j$. If $i = j$ it refers to its i-th element, whereas it denotes an empty set if $i > j$.

As can be observed, the algorithm consists of two phases. First, a binary search is performed by using the pointers low, up and mid. Throughout this phase it holds that the instance $(\mathcal{J}_{1..low}, C)$ is feasible and $(\mathcal{J}_{1..up}, C)$ is infeasible. Notice that initially $low = 0$ and $up = |\mathcal{J}|$, so $\mathcal{J}_{1..low}$ is the empty set and $\mathcal{J}_{1..up}$ contains all the jobs. At each iteration, the algorithm computes the position mid in the middle of low and up and tests the feasibility of the instance $(\mathcal{J}_{1..mid}, C)$. If it is feasible, low is set to mid. Otherwise, up is set to mid. The binary search terminates when $low = up - 1$.

At this point, the set $\mathcal{S}$ is initialized with the jobs in $\mathcal{J}_{1..low}$, $\mathcal{U}$ is initialized with the job $\mathcal{J}_{up..up}$ and $\mathcal{R}$ with the remaining jobs in $\mathcal{J}$. Then, in the second phase, all the remaining jobs are processed in order (picking the first one in $\mathcal{R}$ at each iteration), following a linear search approach as in Algorithm 1. This guarantees that the computed set is an MFSJ of $\mathcal{J}$ (if Feasible is a complete decision procedure).

In the best case, Algorithm 2 computes an MFSJ with a logarithmic number of feasibility tests in the size of $\mathcal{J}$, i.e. when the set $\mathcal{J}_{1..(|\mathcal{J}|-1)}$ is an FSJ of $\mathcal{J}$, which constitutes an exponential improvement over Algorithm 1. On the other hand, in the worst case, i.e. when the set $\mathcal{J}_{1..1}$ is not an FSJ of $\mathcal{J}$, the algorithm would invoke the

Algorithm 3 $G\&T$ schedule builder.

Data: A JSP problem instance $\mathcal{P}$, given by set of jobs $\mathcal{J}$
Result: A schedule S for $\mathcal{P}$
$\mathcal{A} \leftarrow \{\theta_{i1}; J_i \in \mathcal{J}\}$;
$SC \leftarrow \emptyset$;
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;

procedure Feasible more times, but with only a logarithmic overhead.

In addition, it is easy to see that for a given fixed order of the jobs representing a sequence of choices taken in Algorithm 1, the two-phase solution builder would produce the same MFSJ (provided that Feasible is a complete decision procedure). This, together with Proposition 7, serves to prove that the search space defined by the new solution builder contains exactly the set of all MFSJs for any given problem instance.

4.2. Space of schedules for a given set of jobs

The solution builders presented above (Algorithms 1 and 2) rely on iteratively testing the feasibility of subsets of jobs, which is done by invocations to the procedure Feasible. As pointed out, whenever Feasible is a complete decision procedure the solution builders are guaranteed to produce an actual MFSJ of $\mathcal{J}$. However, since each of these feasibility tests requires solving an NP-complete problem (i.e., an instance of the JSP), using an exact algorithm for this task may be prohibitive.

A more efficient approach would be using an incomplete decision procedure for this purpose, at the cost of not having the guarantee that the solution builders will produce an actual MFSJ of $\mathcal{J}$, but an under-approximation instead. The accuracy of the approximation would depend on the effectiveness of the decision procedure used. This way, for testing the feasibility of an instance $(\mathcal{J}', C)$, with $\mathcal{J}' \subseteq \mathcal{J}$, the incomplete decision procedure would search (during a short time) for a schedule of the jobs in $\mathcal{J}'$ with makespan not exceeding the limit C. If such a schedule is found, the set would be correctly declared feasible, and the schedule found would be a witness of the feasibility of $\mathcal{J}'$. Otherwise, despite not having the guarantee that the instance is infeasible, it would be handled as infeasible by the solution builders.

In order to compute schedules for a given instance, Mencía et al. (2019) proposed using the well-known G&T schedule builder (Giffler and Thompson, 1960), depicted in Algorithm 3. Given a JSP instance $\mathcal{P}$, this method iteratively schedules one operation at a time until building a schedule. In the pseudocode, $P(u)$ denotes the immediate predecessor of the operation u in its job sequence, and r_u denotes the *head* of u, i.e. its earliest possible starting time at a given iteration. In addition, the algorithm uses a

Algorithm 4 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 offspring in accordance with P_c and P_m;
  Evaluation: evaluate the resulting chromosomes;
  Replacement: make a tournament selection among every two parents and their offspring to generate $P(t)$;
end
return *the best solution built so far*;

set with all the operations scheduled so far, termed SC, as well as the set $\mathcal{A}$ consisting of all the unscheduled operations being either the first one in their job or whose immediate predecessor in their job has already been scheduled. At each iteration, the algorithm builds a set $\mathcal{B}$ with the *eligible* operations that can be scheduled next. To this aim, it identifies the operation $v^* \in \mathcal{A}$ with the earliest possible completion time, and considers all the operations requiring the same machine as v^* that can start their processing before the earliest possible completion time of v^*. Then, one operation in $\mathcal{B}$ is scheduled at its earliest possible starting time (i.e., at its head), updating the sets SC and $\mathcal{A}$, as well as the heads of all the operations that still need to be scheduled.

The G&T algorithm always returns an *active* schedule, in which no operation can be scheduled earlier without delaying the starting time of some other operation. In addition, notice that the selection of the job $j \in \mathcal{B}$ scheduled at a given iteration is non-deterministic. Noticeably, by considering all possible sequences of choices, the G&T schedule builder defines the search space formed by the set of all active schedules. This set is well-known to be dominant for the makespan, what means that it contains schedules with the minimum possible makespan for any given instance.

5. Genetic algorithm

In this section, we describe a genetic algorithm for the problem of approximating maxFSJs, whose main structure is shown in Algorithm 4. The input to the GA is a problem instance $\mathcal{P}$ (which in this case consists of an infeasible job shop problem instance given by a pair $(\mathcal{J}, C)$), and it has four parameters: crossover and mutation probabilities (P_c and P_m), number of generations ($\#gen$) and population size ($popsize$). As output, the GA returns the largest feasible subset of the jobs in $\mathcal{J}$ (under the makespan limit C) found along the whole search process. The main components of this algorithm are presented below:

Evolutionary model. The GA follows a generational scheme, as can be observed in Algorithm 4. As a first step, the initial population is generated at random and the individuals are evaluated by using the decoding algorithm. At each generation, the

population undergoes a selection phase, that organizes potential parents in pairs randomly. Each pair of parents gives rise to two new individuals, obtained by crossover and mutation operators (according to probabilities P_c and P_m). Then, the replacement phase builds the new population by performing a tournament among each pair of parents and their two offspring. As a consequence, the best individual found along the search is guaranteed to be included in the next generation, what constitutes an implicit form of elitism. The termination criterion considered in Algorithm 4 is completing $\#gen$ generations, although other alternatives can be used in this respect, as letting the GA perform as many generations as possible by a given time limit.

Coding schema. Chromosomes are permutations with repetitions (Bierwirth, 1995). Concretely, for a problem instance with n jobs and m machines, an individual is a permutation of the job indices where each job is repeated m times. This kind of encoding has been commonly used in GAs for solving job shop scheduling problems, and it admits a number of effective and well-studied genetic operators.

In this setting, the j-th occurrence of the index i in a chromosome represents the j-th operation of the job J_i, that is, the operation θ_{ij}. This way, a chromosome encodes an ordering of the operations that can guide a schedule builder, such as the G&T algorithm, in the computation of a schedule. We refer to this ordering as *operation sequence*. For example, for an instance with 4 jobs and 2 machines, the chromosome (4, 1, 1, 3, 2, 4, 2, 3) encodes the operation sequence $(\theta_{41}, \theta_{11}, \theta_{12}, \theta_{31}, \theta_{21}, \theta_{42}, \theta_{22}, \theta_{32})$.

Noticeably, chromosomes defined as permutations with repetitions also encode information for defining MFSJs, what allows the GA to search in both the subset space of the set of jobs and the space of schedules for a given subset of jobs *at the same time*. In particular, a chromosome encodes a total ordering of the jobs, that we refer to as *job sequence*, given by the first appearance of each job. For instance, in this respect the chromosome (4, 1, 1, 3, 2, 4, 2, 3) yields the job sequence (J_4, J_1, J_3, J_2). This information can be used to guide a solution builder, such as Algorithm 1 and Algorithm 2.

Another example of a possible chromosome for an instance with 4 jobs and 2 machines could be the following one: (3, 1, 2, 2, 4, 4, 1, 3). The chromosome encodes the operation sequence $(\theta_{31}, \theta_{11}, \theta_{21}, \theta_{22}, \theta_{41}, \theta_{42}, \theta_{12}, \theta_{32})$ and the job sequence (J_3, J_1, J_2, J_4).

Decoding algorithms. Decoding algorithms transform chromosomes into the actual solutions they represent. For this purpose, the GA can use any of the two solution builders presented in Section 4, aiming at approximating MFSJs by exploiting the information encoded in the chromosomes. This information is used to both guide the solution builders and to determine the feasibility of a given subset of jobs.

At the highest level, the solution builders process the jobs in the order they appear in the job sequence encoded in the chromosome. In this respect, Algorithm 1 selects, at the i-iteration, the i-th job in the sequence, whereas Algorithm 2 assumes that jobs are ordered according to the job sequence.

On the other hand, in order to check the feasibility of different subsets of jobs efficiently, the procedure Feasible in Algorithms 1 and 2 is implemented by a single run of the greedy G&T algorithm. In this context, the G&T algorithm exploits the operation sequence encoded in the chromosome. More specifically, for testing the

feasibility of a given problem instance $(\mathcal{J}', C)$, with $\mathcal{J}' \subseteq \mathcal{J}$ the G&T algorithm builds a schedule S for the jobs in $\mathcal{J}'$ scheduling, at each iteration, the operation in the set $\mathcal{B}$ (see Algorithm 3) that appears first in the operation sequence. We notice that the operation sequence contains all the operations of the jobs in $\mathcal{J}$, so the operations of jobs not in $\mathcal{J}'$ are simply ignored. If the makespan of the computed schedule S does not exceed the maximum limit C, the instance is declared feasible (with S being a witness that $\mathcal{J}$ is a feasible set of jobs). Otherwise, although there is no guarantee that the instance is infeasible, the solution builder would handle it as such. This does not constitute a complete decision procedure, so the computed subset of jobs may not be an actual MFSJ, but an approximation instead.

Anyway, the computed set is a feasible set of jobs, and its cardinality is the fitness value of the chromosome.

Crossover and Mutation. The GA exploits the *Job-based Order Crossover* (JOX) (Ono et al., 1996) operator, which is particularly tailored to permutations with repetitions. It works as follows: given a pair of (parent) chromosomes, JOX selects a random subset of the job indices and copies them to the offspring in the same positions as they appear in the first parent. The remaining positions are filled from the second parent, maintaining their relative order. As an example, let us consider the following two chromosomes:

Parent 1: (**2** 1 1 3 **2** 3 1 **2** 3) Parent 2: (3 3 1 2 1 3 2 2 1)

If the selected subset of jobs only includes the job 2, the generated offspring is:

Offspring: (**2** 3 3 1 **2** 1 3 **2** 1).

The second offspring is obtained by the same procedure, but switching the role of the parents. On the other hand, the mutation operator introduces small changes by swapping two consecutive positions of the chromosome randomly.

Overall integrated workflow. Figure 2 shows the overall workflow of the genetic algorithm. Given a problem instance and the GA parameters, the GA starts by generating and evaluating the initial population. Then, until the termination condition is met, the population evolves by undergoing selection, recombination, evaluation and replacement phases. At each generation, the chromosomes in the population are paired randomly (selection). In the recombination phase, each pair of chromosomes gives rise to two offspring, by the application of crossover and mutation operators. Then, the new individuals are evaluated, obtaining actual solutions from the chromosomes. To this aim, for each individual the job sequence and the operations sequence are extracted from the chromosome. Next, the solution builder is invoked, which is guided by the job sequence in the approximation of an MFSJ. For this purpose, either the one based on linear search (Algorithm 1) or the new one that integrates binary search (Algorithm 2) can be used. The solution builder makes several invocations to the procedure Feasible, which is guided by the operations sequence extracted from the chromosome. The implementation of this procedure is based on the G&T algorithm, as explained in this section. An improvement to this procedure is presented in the following section (Algorithm 5). After all the chromosomes are evaluated, the ones that will survive for the

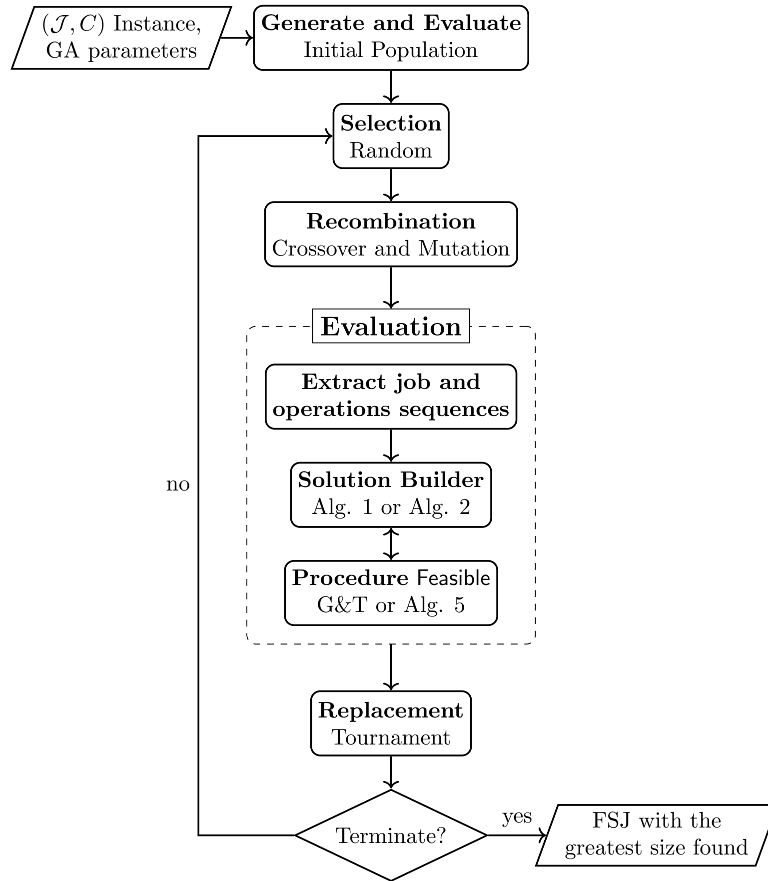


Figure 2: Overall workflow of the genetic algorithm.

next generation are those that are fitter, according to a tournament between parents and offspring (replacement). After the GA is run for the given number of generations, the FSJ with the greatest size found along the whole process is returned.

6. Improving the effectiveness of the decoding algorithms

The decoding algorithms used by the GA aim at under-approximating an MFSJ efficiently by using the greedy G&T algorithm to test the feasibility of subsequent subsets of jobs. Arguably, using a greedy algorithm for this purpose represents the most efficient approach possible but, in turn, the approximations produced may not always be accurate. If only effectiveness is sought for, one could use a complete decision procedure; however, as already pointed out, using an exact algorithm, e.g. (Brucker et al., 1994; Beck, 2007; Mencía et al., 2014; Vilím et al., 2015), may be too time consuming, making the approach impractical. Between these two extremes there is a

wide range of possibilities, such as using metaheuristics more effective (although more expensive) than greedy algorithms to test the feasibility of a given subset of jobs.

Herein, we aim at improving the effectiveness of the decoding algorithms by means of an alternative implementation of the procedure Feasible used in Algorithms 1 and 2. To this aim, we propose an approach that not only issues the G&T algorithm for this task, but also uses a light-weight genetic algorithm, termed LWGA throughout, if the former fails at proving feasibility.

The proposed method is shown in Algorithm 5. Given a set of jobs $\mathcal{J}'$, a limit on the makespan C and a probability P_{LWGA}, the algorithm first computes a schedule S by using the greedy G&T algorithm. If the makespan of S is not greater than C, the instance $(\mathcal{J}', C)$ is efficiently declared feasible. Otherwise, the method performs a second phase with probability P_{LWGA} (in this respect, the procedure $Random(0, 1)$ generates a random number in the interval [0,1] with uniform distribution). In this phase the procedure invokes LWGA, that searches for a schedule minimizing the makespan and returns the best schedule S' found after a short running time. Finally, if S' has a makespan not exceeding C, the instance is declared feasible, whereas otherwise the feasibility of the instance is deemed unknown.

We note that the procedure returns a Boolean value indicating whether the instance $(\mathcal{J}', C)$ has been proven feasible. In this regard, the value true means that a schedule with makespan not exceeding C has been found and so the instance is feasible. In contrast, the value false does not mean that the instance is necessarily infeasible, but that a schedule with makespan not exceeding C has not been found. So, Algorithm 5 is an incomplete (but sound) decision procedure. However, since it uses a more effective procedure for searching for high-quality schedules, it can be expected to detect feasibility more times than only using the G&T algorithm, thus leading to better approximations of MFSJs. In any case, whenever an instance is declared feasible, there is the guarantee that the result is correct, and so the computed approximations of MFSJs are correct as well.

Moreover, although it is not shown in the pseudocode, the procedure exploits the operation sequences encoded in the chromosomes of the main GA both when issuing the G&T algorithm (as described in Section 5), and LWGA (as described below). So, it is integrated in the global search carried out by the main GA.

Since the procedure Feasible is invoked many times along the execution of the main GA, in order to keep the overhead reasonably low, LWGA is only invoked with the given probability P_{LWGA} introduced as a parameter. The value of this parameter is the same for all the invocations along the execution of the main GA. This way, the potential improvements are equally distributed among all the individuals.

6.1. Light-weight genetic algorithm (LWGA)

As pointed out, LWGA searches for schedules aiming at minimizing the makespan. It follows the same general structure as the main GA, shown in Algorithm 4. However, since LWGA must not take a long running time, we need to limit its population size and number of generations to small values. In addition, LWGA terminates whenever it finds a schedule with a makespan less than or equal to the limit C.

As the main GA described in Section 5, LWGA uses permutations with repetitions for encoding chromosomes. In this case, a chromosome only includes the tasks of the

Algorithm 5 Procedure Feasible.

Data: Set of jobs $\mathcal{J}'$, makespan limit C, probability P_{LWGA}
Result: Boolean value indicating that the instance $(\mathcal{J}', C)$ has been proven feasible

$S \leftarrow G\&T(\mathcal{J}')$;
if $C_{max}(S) \leq C$ **then**
  return true;
else if $Random(0, 1) \leq P_{LWGA}$ **then**
  $S' \leftarrow LWGA(\mathcal{J}', C)$;
  if $C_{max}(S') \leq C$ **then**
    return true;
  end
end
return false

jobs in $\mathcal{J}'$, representing operation sequences. Besides, it uses the same crossover (JOX) and mutation operators. As decoding algorithm, LWGA exploits the G&T algorithm guided by the operation sequences encoded in the chromosomes. After building a schedule, the fitness of the individual is the makespan of the computed schedule.

Furthermore, two additional processes are carried out as a means to effectively integrating LWGA in the global search process conducted by the main GA.

First, in order to exploit the information encoded in a given chromosome c of the main GA for guiding the search in the space of schedules, LWGA creates its initial population by the following procedure: a chromosome c' is created by considering the tasks of the jobs in $\mathcal{J}'$ keeping their order as they appear in c, and the resulting chromosome c' is included in the population. Then another two chromosomes are generated by introducing small random perturbations to c' (by swapping two random positions of c'), which are included in the initial population as well. The remaining individuals are generated at random. This way, the characteristics of c are included in the initial population of LWGA, but not in an extent that would cause it to converge very prematurely.

On the other hand, when using LWGA, the main GA instruments a Lamarckian evolution model in order to transfer the characteristics that led to an improvement back to the main GA's population. For this purpose, after an approximation of an MFSJ has been computed by the decoding algorithm of the main GA, if LWGA proved the feasibility of the subset of jobs, the corresponding chromosome c of the main GA is replaced by a chromosome c'' built as follows: the first positions of c'' correspond to the chromosome that led LWGA to compute a schedule for such subset of jobs $\mathcal{J}'$ without exceeding the makespan limit; then, the remaining positions of c'' are filled with the tasks of the remaining jobs in the order they appear in c. Lamarckian evolution models are commonly used by memetic algorithms, that combine a genetic algorithm with a local search procedure. However, these could also be expected of practical use in this this context.

7. Experimental results

An experimental study was conducted to evaluate the methods proposed in this work. For this purpose, we coded a prototype in C++ implementing the algorithms and ran experiments on a Linux machine (Intel Xeon 2.26 GHz. 128 GB RAM).

The experiments were carried out over a set of infeasible instances derived from classical JSP benchmarks from the OR-library (Beasley, 1990) as well as larger Taillard's instances (Taillard, 1993). The benchmark set consists of instances of different sizes $n \times m$ with a number of jobs $n \in \{10, 15, 20, 30, 50\}$ and a number of machines $m \in \{5, 10, 15, 20\}$. Specifically, we consider 5 instances of size 10×5: LA01-05; 5 instances of size 15×5: LA06-10; 6 instances of size 20×5: LA11-15 and FT20; 16 instances of size 10×10: LA016-20, ORB01-10 and FT10; 5 instances of size 15×10: LA21-25; 5 instances of size 20×10: LA26-30; 5 instances of size 30×10: LA31-35; 5 instances of size 15×15: LA36-40; 10 instances of size 50×15: tai50_15_01-10; and 10 instances of size 50×20: tai50_20_01-10. For each of these instances, three infeasible instances were built by fixing different values of the makespan limit C to be 70%, 80% and 90% of the optimal makespan of the JSP instance, denoted C_{opt}. So, in all there are 216 instances.

The goal of the experimental study is to assess the performance of the proposed methods. To this aim we compare three genetic algorithms, termed GA_{LS}, GA_{BS} and GA^* throughout. All these algorithms share the main structure and components described in Section 5, but differ in the solution builder used in the decoding phase. GA_{LS} uses the solution builder based on linear search depicted in Algorithm 1, whereas GA_{BS} exploits the two-phase solution builder that integrates a binary search phase shown in Algorithm 2. Both GA_{LS} and GA_{BS} use a single run of the greedy G&T algorithm (Algorithm 3) for testing the feasibility of different subsets of jobs. On the other hand, motivated by the first series of experiments shown below, GA^* integrates the two-phase solution builder, but using the procedure Feasible given in Algorithm 5, which issues the light-weight genetic algorithm (termed LWGA in Section 6) with a given probability P_{LWGA} when the single run of the G&T algorithm fails at proving feasibility. We notice that GA_{LS} corresponds to the genetic algorithm proposed in (Mencía et al., 2019), which was shown to perform (much) better than a simple enumeration of random approximations of MFSJs. To our knowledge, this is the only algorithm proposed so far for the problem tackled in this paper, so it serves as a baseline method in the experimental evaluation.

In all the experiments, the considered GAs evolve a population of 100 individuals with crossover probability of 0.9 and mutation probability of 0.1 until a termination condition is met (either completing a number of generations or surpassing a given time limit). In the case of GA^*, the underlying LWGA has the same crossover and mutation probabilities. The algorithms were run 20 times on each instance.

Furthermore, in the experimental study the quality of the solutions computed is evaluated in terms of their error in percentage w.r.t. the best solution found for each instance along all the experiments. More concretely, if for a given instance the best known solution has N_{best} jobs and an algorithm finds a solution with N jobs (with $N \leq N_{best}$), the error in percentage is computed as $100 \times (N_{best} - N)/N_{best}$.

Our first hypothesis is that GA_{BS} and GA_{LS} will yield similar results in terms of the

Table 3: Main notation used in the experimental study.

Notation	Definition
n	Number of jobs
m	Number of machines
C	Makespan limit
C_{opt}	Optimal makespan
LWGA	Light-weight genetic algorithm
P_{LWGA}	Probability of invoking LWGA in procedure Feasible (Algorithm 5)
GA_{LS}	GA with the solution builder based on linear search (Algorithm 1)
GA_{BS}	GA with the solution builder based on binary search (Algorithm 2)
GA^*	GA with the solution builder based on binary search issuing LWGA in procedure Feasible
$\#c$	Number of individuals (population size) for LWGA
$\#g$	Number of generations for LWGA
$GA^{P_{LWGA}}_{\#c/\#g}$	GA^* using a probability P_{LWGA} and setting the underlying LWGA with a population size of $\#c$ individuals and $\#g$ generations

quality of solutions reached, while GA_{BS} will be faster, since the solution builder with the binary search phase is expected to make less invocations to the procedure Feasible than the solution builder based on linear search. On the other hand, we expect GA^* to reach better solutions than GA_{BS} and GA_{LS}, due to a more effective implementation of the procedure Feasible, at the expense of longer running times.

The experimental study is divided in three parts. We first compare GA_{LS} and GA_{BS} in order to assess the two solution builders. Then, we focus on GA^* and analyze the effect that the parameter P_{LWGA} and the configuration of the underlying LWGA have on the overall performance of the algorithm. Finally, we provide a detailed comparison of the three genetic algorithms.

Table 3 summarizes the main notation used in this section.

7.1. Comparing GA_{LS} and GA_{BS}

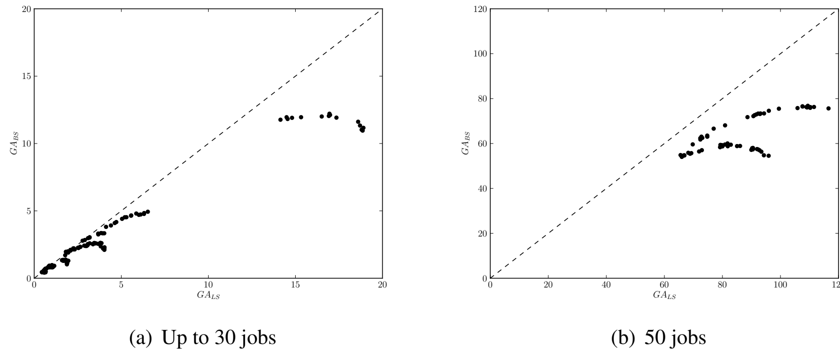
We conducted a first series of experiments in order to compare the two solution builders and assess the efficiency gains that introducing a binary search phase has on the performance of the algorithm. For this purpose, we ran both GA_{LS} and GA_{BS} on the whole benchmark set, limiting the number of generations to 250.

Table 4 summarizes the results. It shows the error (in percentage terms) of the best (Best) and average solutions (Avg.) obtained by each method over the 20 independent runs, as well as the computation times in seconds, averaged for each group of instances of the same size $n \times m$. As we can observe, both algorithms yield solutions of similar quality regardless of the size of the instances, with only small variations that may be due to their stochastic nature. The two genetic algorithms are very effective at solving the smallest instances (up to size 20×5), obtaining solutions close to the best known ones, although the error grows for larger (and more challenging) instances, exceeding 10% on average for the largest ones of size 50×20.

However, GA_{BS} is faster than GA_{LS} at completing the 250 generations. On average GA_{BS} takes less than 75% of the time taken by GA_{LS}. This indicates that the binary search phase included in the solution builder used by GA_{BS} is effective at saving feasibility tests, what enables reducing the overall computation time. Greater improvements

Table 4: Summary of results from GA_{BS} and GA_{LS} after evolving a population of 100 individuals over 250 generations. Best and average results (in terms of error in percentage) from 20 independent runs are reported. Running times are given in seconds.

	GA_{LS}			GA_{BS}		
$n \times m$	Best	Avg.	T.(s)	Best	Avg.	T.(s)
10×5	0.00	1.09	0.58	0.00	1.12	0.48
15×5	0.99	1.12	1.78	0.99	1.07	1.27
20×5	0.68	1.49	3.66	1.01	1.38	2.49
10×10	1.77	4.82	0.85	2.35	4.75	0.80
15×10	3.37	7.31	2.51	3.97	6.74	2.21
20×10	4.83	8.04	5.44	4.73	7.88	4.50
30×10	3.79	6.18	16.80	3.79	6.28	11.72
15×15	3.07	7.63	2.98	3.91	7.83	2.78
50×15	6.08	8.42	80.72	5.91	8.36	57.09
50×20	7.32	10.28	92.34	7.14	10.20	70.95
All	3.43	5.97	26.62	3.63	5.89	19.76



(a) Up to 30 jobs

(b) 50 jobs

Figure 3: Scatter plots of the time taken (seconds) by GA_{LS} and GA_{BS} to complete 250 generations on the instances with (a) up to 30 jobs and (b) 50 jobs.

are observed as the number of jobs grows, which does not come as a surprise since in these cases a larger number of tests can be saved by applying binary search.

Figure 3 shows two scatter plots depicting the time taken by GA_{LS} and GA_{BS} to complete 250 generations for the instances with up to 30 jobs (Figure 3(a)) and for the largest instances with 50 jobs (Figure 3(b)). Each point represents the running time in seconds of GA_{LS} (horizontal axis) compared to GA_{BS} (vertical axis) on a given instance. Points below the diagonal indicate that GA_{BS} is faster than GA_{BS}. As can be observed, the difference in favor of GA_{BS} is greater with longer computation times. Noticeably, for the instances with 50 jobs, every point is far from the diagonal, which means that for these large instances, GA_{BS} is much faster than GA_{LS}.

Table 5: Summary of average results (error in percentage) from the configurations of GA* after evolving a population of 100 individuals over 250 generations.

	$P_{LWGA} = 0.15$				$P_{LWGA} = 0.5$				$P_{LWGA} = 1$			
$\#c/\#g$	10/10	10/20	10/30	20/20	10/10	10/20	10/30	20/20	10/10	10/20	10/30	20/20
10×5	0.33	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**
15×5	0.99	0.99	0.96	0.94	0.97	0.89	0.81	0.79	0.99	0.82	0.59	**0.18**
20×5	0.83	0.76	0.73	0.60	0.76	0.65	0.59	0.57	0.65	0.63	0.61	**0.47**
10×10	1.57	1.10	0.83	0.69	1.06	0.51	0.30	0.17	0.81	0.23	**0.06**	0.12
15×10	4.49	3.79	3.92	3.49	3.64	3.49	3.29	2.86	3.12	2.90	2.52	**2.25**
20×10	5.36	4.95	4.75	4.45	4.50	4.22	3.71	3.22	4.42	3.67	3.24	**2.81**
30×10	3.43	3.31	3.36	3.12	3.22	2.91	2.70	2.23	2.95	2.59	2.22	**1.92**
15×15	3.19	2.76	2.49	2.05	2.53	1.63	1.38	1.13	1.71	1.19	1.16	**1.03**
50×15	3.89	3.51	3.41	2.96	3.05	2.67	2.45	**2.27**	2.61	2.30	2.62	2.62
50×20	4.73	4.05	3.85	3.41	3.44	**2.81**	2.91	2.83	2.94	2.92	3.40	3.68
All	2.85	2.46	2.33	2.06	2.23	1.84	1.69	**1.51**	1.92	1.61	1.58	**1.51**

7.2. Analyzing GA*

The second part of the experimental study is aimed at analyzing GA*. As mentioned above, GA* uses the procedure given in Algorithm 5 to test the feasibility of subsets of jobs. This procedure issues a light-weight genetic algorithm (LWGA) with a given probability P_{LWGA} whenever the G&T algorithm fails at proving feasibility. So, it is expected to be (much) more time consuming than simply using the G&T algorithm for this purpose (as GA_{LS} and GA_{BS} do). As a consequence, and given the results in the previous subsection, GA* uses the solution builder that integrates the binary search phase in order to save time.

In these experiments we evaluate different values of the parameter P_{LWGA}; concretely we consider values for P_{LWGA} in $\{0.15, 0.5, 1\}$. In addition, we evaluate different configurations of LWGA in terms of population size and number of generations. In this respect, we consider four configurations: evolving a population of 10 individuals over 10, 20 and 30 generations, and a population of 20 individuals over 20 generations. Throughout we will use the notation $GA^{P_{LWGA}}_{\#c/\#g}$ to refer to GA* using a given probability P_{LWGA} and setting the underlying LWGA with a population size of $\#c$ individuals and $\#g$ generations. Taking all into account, we consider 12 configurations of GA*.

We first evaluate the different versions of GA* with the termination criterion of completing 250 generations. Besides, since some configurations may be too time consuming, we set a timeout of 3600 seconds in these experiments. Table 5 shows, for each configuration, the error on average of the solutions obtained over the 20 independent runs, averaged for groups of instances of the same size. On the other hand, Table 6 shows the time taken by each configuration in seconds. In both cases, the last row averages the results over all the instances.

As we can observe in Table 5, the quality of the solutions improves with larger values of P_{LWGA} in most cases. For most groups of instances, using $P_{LWGA} = 1$ yields the best results. However, the largest instances of size 50×15 and 50×20 are the exception to this trend. In these cases, options with $P_{LWGA} = 0.5$ achieved the best results. This can be explained by the fact that for these large instances, some configurations with $P_{LWGA} = 1$ were only able to complete a fraction of the 250 iterations

Table 6: Summary of the average time (in seconds) that each configuration of GA* took to evolve a population of 100 individuals over 250 generations.

	$P_{LWGA} = 0.15$				$P_{LWGA} = 0.5$				$P_{LWGA} = 1$			
$\#c/\#g$	10/10	10/20	10/30	20/20	10/10	10/20	10/30	20/20	10/10	10/20	10/30	20/20
10×5	**2.2**	3.0	3.8	5.7	6.2	9.2	11.8	18.7	12.4	19.0	24.4	40.0
15×5	**5.6**	8.0	9.9	15.2	16.3	25.1	32.1	51.8	33.1	52.1	67.6	109.7
20×5	**12.1**	18.0	23.1	34.9	35.6	56.3	74.3	117.5	71.4	115.4	153.8	245.1
10×10	**4.5**	6.5	8.3	12.6	13.0	20.3	26.2	41.4	26.0	41.1	53.6	86.2
15×10	**12.3**	18.3	23.2	36.2	36.2	56.7	73.9	119.3	72.5	116.0	151.2	248.0
20×10	**26.4**	40.3	51.5	79.5	79.0	126.6	165.7	262.7	157.2	257.2	339.5	546.4
30×10	**67.4**	106.3	140.2	205.1	202.9	337.8	457.0	682.6	404.5	688.9	941.6	1411.4
15×15	**16.8**	25.5	33.1	50.5	50.2	80.0	105.8	167.4	100.1	162.5	219.1	349.3
50×15	**397.1**	658.4	912.4	1258.7	1206.6	2107.5	2988.6	3567.5	2388.6	3581.2	3600	3600
50×20	**530.3**	885.5	1229.5	1707.3	1619.9	2849.5	3600	3600	3212.4	3600	3600	3600
All	**139.9**	231.4	319.4	444.9	425.6	741.8	986.5	1106.5	843.8	1107.7	1149.8	1233.0

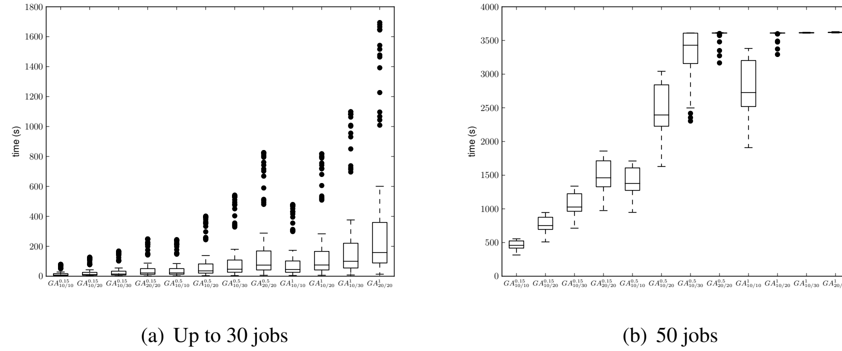
by the time limit of 3600 seconds. In the rest of the experiments, they completed them and produced the best results. Also, configurations with $P_{LWGA} = 0.5$ yielded better results than those with $P_{LWGA} = 0.15$ for all the instance sets. On the other hand, assigning the LWGA larger values of population size and number of generations leads to better results as well. In most cases, the configurations that produce the least average error are those with $\text{GA}_{20/20}^{P_{LWGA}}$, followed by $\text{GA}_{10/30}^{P_{LWGA}}$, then $\text{GA}_{10/20}^{P_{LWGA}}$ and the configuration in the last position is normally $\text{GA}_{10/10}^{P_{LWGA}}$. This ranking does not hold for $P_{LWGA} \in \{0.5, 1.0\}$ on the largest instances, due to the reason mentioned above.

These results indicate that the use of LWGA is effective at testing feasibility, and that issuing it more often and with larger population size and number of generations leads to better results in most cases. However, these improvements do not come without a cost. As shown in Table 6, running times grow significantly with P_{LWGA}, as well as with the population size and number of generations of LWGA, being $\text{GA}_{10/10}^{0.15}$ the most efficient approach, as could be expected. Note that for some configurations with $P_{LWGA} \in \{0.5, 1\}$ running times are rather long, especially for the largest instances, where some of these configurations timed out before completing the 250 generations.

Figure 4 shows two boxplots of the time taken (seconds) to complete 250 generations by the different configurations of GA* on the instances with up to 30 jobs (Figure 4(a)) and 50 jobs (Figure 4(b)). We can see that, the larger the values of P_{LWGA}, $\#c$ and $\#g$ are, the more time consuming the resulting algorithm is.

Figure 5(a) shows a boxplot of the average errors for the 50×20 set, which illustrates the behaviour of the different configurations on the largest instances. As we can observe, the best configurations are those in the middle in terms of resource consumption, which reach a proper balance between the effort spent at each individual and the global search process that the genetic algorithm is able to conduct under the given conditions.

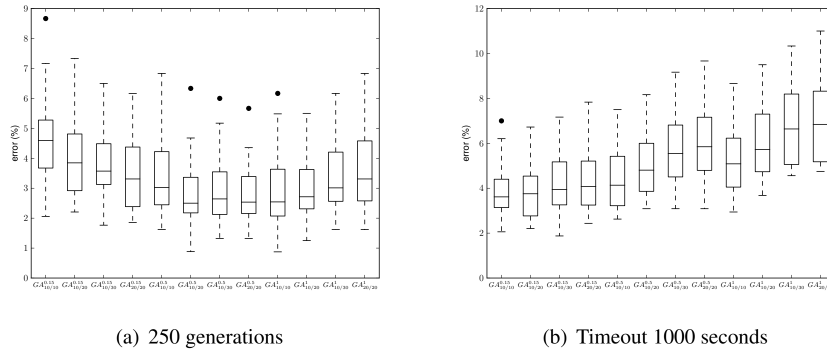
In order to do a fair comparison, we conducted a series of experiments giving the algorithms the same time, letting the different configurations to perform as many generations as possible by the given time limit. Concretely, for an instance of size $n \times m$,



(a) Up to 30 jobs

(b) 50 jobs

Figure 4: Boxplots of the time taken (seconds) by the different configurations of GA* to complete 250 generations on the instances with (a) up to 30 jobs and (b) 50 jobs.



(a) 250 generations

(b) Timeout 1000 seconds

Figure 5: Boxplots of the average error in percentage obtained for the instances of size 50×20, limiting the configurations of GA* to (a) 250 generations and (b) 1000 seconds.

we set the time limit to $n \times m$ seconds (which we believe is a reasonable choice).

The results are summarized in Table 7, which reports the errors on average over the 20 independent runs produced by the different configurations, averaged for each group of instances, as well as the average values over the whole benchmark set. Looking at the results for the smallest instances, i.e., 10×5 and 15×5, we can draw similar conclusions as to what we observed in the previous comparisons: the heavier configurations of GA* perform better, although with small differences in this case. However, as the size of the instances grows, this trend is inverted, favoring smaller values of P_{LWGA} as well as smaller population size and number of generations of the underlying LWGA. Figure 5(b) shows a boxplot for the 50×20 instance set. We can see how the lighter versions of GA* are now favored when comparing to the first series of experiments.

Overall, the best configuration of GA* seems to be $\text{GA}^{0.15}_{10/20}$, since it yields the least average error if all the instances are considered. So, we will use this configuration

in the next series of experiments.

7.3. Detailed comparison

The last part of the experimental study is aimed at comparing the three GAs. In this comparison GA* is configured with $P_{LWGA} = 0.15$ and the underlying LWGA with a population size of 10 individuals and 20 generations.

As before, we first compare the algorithms with a termination criterion of 250 generations. The results are summarized in Table 8, which reports the best and average errors over the 20 independent runs and the time taken in seconds, averaged for instances of the same size and the same value of $\%C_{opt}$.

It can be observed that GA* reaches (much) better results than the other two methods for all instances sizes considered in terms of both best and average solutions, and that the difference grows in favor of GA* with the size of the instances. As was expected, GA* takes significantly more time than GA_{LS} and GA_{BS}, especially for the large instances (which require a larger number of feasibility tests).

On the other hand, the imposed value of C (w.r.t $\%C_{opt}$) seems to affect the running times of the algorithms. In general, GA_{LS} and GA_{BS} take more time when the value of C is large, although GA_{BS} is more stable. This may be due to the fact that the under-approximations of maxFSJs computed are larger with high values of C, and so the feasibility tests are expected to be more expensive, since more jobs must be scheduled by the G&T algorithm. Interestingly, GA* shows a different trend, often taking more time for low values of C. This may be due to the fact that with lower values of C, more infeasible sets of jobs would need to be tested, increasing the number of invocations to the underlying LWGA, and so running time.

Figures 6(a), 6(c) and 6(e) show the evolution of the average error of the solutions computed by each algorithm for the 30×10, 50×15 and 50×20 instance sets along the 250 generations. It can be appreciated that GA_{BS} and GA_{LS} have similar results at

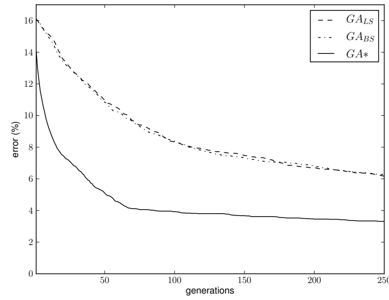
Table 7: Summary of average results (error in percentage) from the configurations of GA* after $n \times m$ seconds.

	$P_{LWGA} = 0.15$				$P_{LWGA} = 0.5$				$P_{LWGA} = 1$			
$\#c/\#g$	10/10	10/20	10/30	20/20	10/10	10/20	10/30	20/20	10/10	10/20	10/30	20/20
10×5	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**
15×5	0.99	0.97	0.97	0.87	0.99	0.92	0.79	0.62	0.94	0.74	0.47	**0.35**
20×5	0.62	0.55	**0.46**	0.51	0.66	0.55	0.57	0.57	0.68	0.60	0.63	0.60
10×10	0.38	0.14	0.07	0.10	0.36	0.10	0.03	0.12	0.32	0.05	**0.02**	0.09
15×10	2.80	**2.16**	2.60	2.24	2.77	2.62	2.84	2.34	2.69	2.62	2.66	2.58
20×10	3.51	**2.99**	3.25	3.21	3.66	3.62	3.54	3.99	4.47	3.62	3.74	4.18
30×10	**2.63**	2.81	2.68	2.97	3.13	3.12	3.18	3.24	3.12	3.31	3.36	3.44
15×15	1.42	1.23	1.17	1.19	1.13	1.20	**1.11**	1.25	1.25	1.16	1.19	1.22
50×15	**3.34**	3.42	3.58	3.64	3.65	4.02	4.53	4.75	4.18	4.79	5.35	5.67
50×20	**3.88**	3.91	4.13	4.36	4.34	5.02	5.64	6.08	5.20	6.11	6.91	7.12
All	1.93	**1.80**	1.86	1.90	2.06	2.12	2.26	2.37	2.30	2.37	2.55	2.67

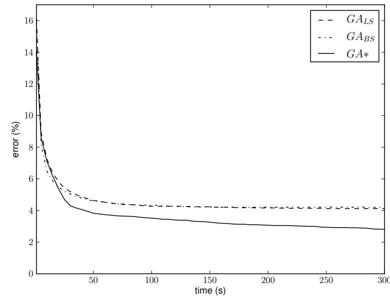
Table 8: Summary of results from GA_{LS}, GA_{BS} and GA^* after evolving a population of 100 individuals over 250 generations.

		GA_{LS}			GA_{BS}			GA^*		
$n \times m$	$\%C_{opt}$	Best	Avg.	T.(s)	Best	Avg.	T.(s)	Best	Avg.	T.(s)
10×5	70	**0.00**	**0.00**	0.51	**0.00**	**0.00**	0.48	**0.00**	**0.00**	3.76
	80	**0.00**	0.38	0.61	**0.00**	0.12	0.48	**0.00**	**0.00**	3.42
	90	**0.00**	2.89	0.63	**0.00**	3.22	0.47	**0.00**	**0.00**	1.94
	Avg.	**0.00**	1.09	0.58	**0.00**	1.12	0.48	**0.00**	**0.00**	3.04
15×5	70	**0.00**	0.08	1.64	**0.00**	**0.00**	1.33	**0.00**	**0.00**	10.09
	80	**1.54**	1.69	1.80	**1.54**	1.77	1.32	**1.54**	**1.54**	8.68
	90	**1.43**	1.57	1.90	**1.43**	**1.43**	1.17	**1.43**	**1.43**	5.16
	Avg.	**0.99**	1.12	1.78	**0.99**	1.07	1.27	**0.99**	**0.99**	7.98
20×5	70	**1.11**	2.41	3.34	**1.11**	1.93	2.58	**1.11**	**1.11**	21.67
	80	**0.00**	1.09	3.70	0.98	1.29	2.59	**0.00**	**0.25**	18.50
	90	**0.93**	0.97	3.92	**0.93**	**0.93**	2.29	**0.93**	**0.93**	13.88
	Avg.	**0.68**	1.49	3.66	1.01	1.38	2.49	**0.68**	**0.76**	18.02
10×10	70	1.25	3.60	0.62	2.29	3.57	0.64	**0.00**	**0.00**	6.34
	80	2.68	4.79	0.87	2.68	5.02	0.86	**0.00**	**1.16**	7.03
	90	1.39	6.08	1.05	2.08	5.64	0.90	**0.00**	**2.14**	6.21
	Avg.	1.77	4.82	0.85	2.35	4.75	0.80	**0.00**	**1.10**	6.53
15×10	70	2.00	6.90	2.01	2.00	6.03	1.95	**0.00**	**2.82**	18.11
	80	3.48	7.71	2.53	5.30	7.18	2.26	**1.67**	**3.85**	19.18
	90	4.62	7.31	2.98	4.62	7.00	2.42	**3.08**	**4.69**	17.47
	Avg.	3.37	7.31	2.51	3.97	6.74	2.21	**1.58**	**3.79**	18.25
20×10	70	5.93	9.36	4.58	**4.40**	9.28	4.08	**4.40**	**6.07**	41.13
	80	3.92	8.31	5.50	5.17	7.96	4.63	**1.25**	**4.97**	42.25
	90	4.64	6.45	6.25	4.64	6.39	4.80	**2.29**	**3.82**	37.48
	Avg.	4.83	8.04	5.44	4.73	7.88	4.50	**2.64**	**4.95**	40.28
30×10	70	4.55	7.67	14.66	4.55	7.47	11.88	**1.78**	**3.75**	124.77
	80	3.23	5.33	16.95	3.23	5.88	12.06	**0.00**	**2.58**	112.19
	90	**3.60**	5.54	18.78	**3.60**	5.47	11.22	**3.60**	**3.60**	82.02
	Avg.	3.79	6.18	16.80	3.79	6.28	11.72	**1.79**	**3.31**	106.33
15×15	70	**0.00**	4.76	2.07	2.50	5.59	2.10	**0.00**	**0.00**	22.26
	80	4.22	9.30	3.03	4.22	9.39	2.93	**0.00**	**3.46**	26.76
	90	5.00	8.83	3.83	5.00	8.50	3.32	**3.33**	**4.83**	27.52
	Avg.	3.07	7.63	2.98	3.91	7.83	2.78	**1.11**	**2.76**	25.52
50×15	70	7.48	9.75	68.31	6.89	9.60	55.35	**1.49**	**4.00**	716.23
	80	5.45	8.34	81.67	6.22	8.47	59.13	**2.06**	**3.64**	690.99
	90	5.32	7.18	92.18	4.63	7.02	56.81	**1.85**	**2.88**	568.07
	Avg.	6.08	8.42	80.72	5.91	8.36	57.09	**1.80**	**3.51**	658.43
50×20	70	8.63	12.52	74.26	8.95	12.32	63.37	**2.31**	**5.39**	874.16
	80	7.39	9.87	93.02	6.80	9.91	73.29	**1.71**	**3.62**	931.55
	90	5.93	8.45	109.73	5.67	8.38	76.20	**1.23**	**3.15**	850.92
	Avg.	7.32	10.28	92.34	7.14	10.20	70.95	**1.75**	**4.05**	885.54
All		3.43	5.97	26.62	3.63	5.89	19.76	**1.11**	**2.46**	231.38

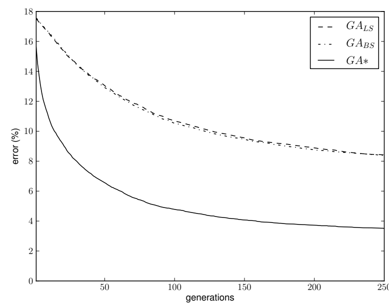
each generation, while GA^* yields solutions with much less error along the whole process. Although the improvement in terms of the errors is less significant during the last



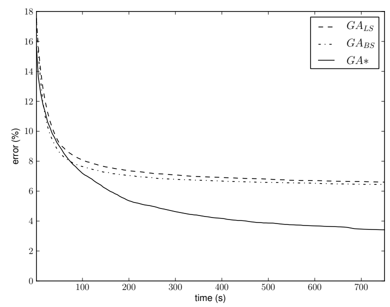
(a) Generations (30×10)



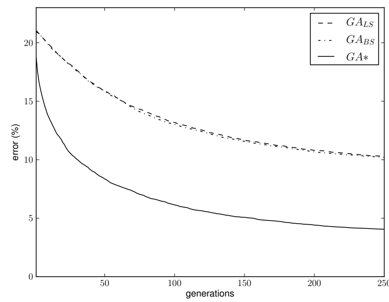
(b) Time (30×10)



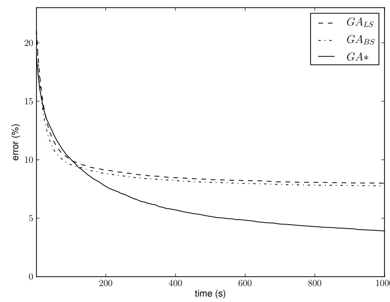
(c) Generations (50×15)



(d) Time (50×15)



(e) Generations (50×20)



(f) Time (50×20)

Figure 6: Evolution of the average error in percentage over generations and time obtained by the genetic algorithms for the instances of sizes 30×10, 50×15 and 50×20.

generations, it seems in the three sets that the results could improve if the algorithms were given more generations, which we do in the following series of experiments.

The previous series of experiments shows that GA* reaches better solutions than the other two methods, but also that it takes more time. To perform a fair comparison

Table 9: Summary of results from GA_{LS}, GA_{BS} and GA^* after evolving a population of 100 individuals for the same running time of $n \times m$ seconds.

		GA_{LS}		GA_{BS}		GA^*	
$n \times m$	$\%C_{opt}$	Best	Avg.	Best	Avg.	Best	Avg.
10×5	70	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**
	80	**0.00**	0.12	**0.00**	**0.00**	**0.00**	**0.00**
	90	**0.00**	3.11	**0.00**	3.00	**0.00**	**0.00**
	Avg.	**0.00**	1.08	**0.00**	1.00	**0.00**	**0.00**
15×5	70	**0.00**	0.08	**0.00**	0.08	**0.00**	**0.00**
	80	**1.54**	**1.54**	**1.54**	**1.54**	**1.54**	**1.54**
	90	1.43	1.57	1.43	1.43	**0.00**	**1.36**
	Avg.	0.99	1.06	0.99	1.02	**0.51**	**0.97**
20×5	70	1.11	1.50	1.11	1.50	**0.00**	**1.06**
	80	**0.00**	0.78	**0.00**	0.83	**0.00**	**0.00**
	90	**0.00**	0.88	0.93	0.93	**0.00**	**0.60**
	Avg.	0.37	1.05	0.68	1.09	**0.00**	**0.55**
10×10	70	1.25	3.71	**0.00**	3.35	**0.00**	**0.00**
	80	0.89	4.29	2.68	4.11	**0.00**	**0.04**
	90	2.08	4.12	2.08	3.85	**0.00**	**0.38**
	Avg.	1.41	4.04	1.59	3.77	**0.00**	**0.14**
15×10	70	**0.00**	5.23	**0.00**	5.25	**0.00**	**1.31**
	80	3.48	5.57	3.48	4.75	**0.00**	**2.86**
	90	3.08	5.31	3.08	5.46	**0.00**	**2.31**
	Avg.	2.19	5.37	2.19	5.15	**0.00**	**2.16**
20×10	70	**2.97**	6.82	**2.97**	7.09	**2.97**	**4.33**
	80	**1.25**	5.76	2.58	5.75	**1.25**	**2.99**
	90	1.11	4.40	1.11	4.35	**0.00**	**1.65**
	Avg.	1.78	5.66	2.22	5.73	**1.41**	**2.99**
30×10	70	0.91	5.13	2.69	5.31	**0.00**	**2.43**
	80	0.80	3.35	2.40	3.51	**0.00**	**2.39**
	90	**3.60**	3.88	**3.60**	3.78	**3.60**	**3.60**
	Avg.	1.77	4.12	2.90	4.20	**1.20**	**2.81**
15×15	70	**0.00**	3.20	**0.00**	4.26	**0.00**	**0.00**
	80	2.22	7.62	2.00	7.48	**0.00**	**0.44**
	90	3.33	5.50	**1.67**	5.58	**1.67**	**3.25**
	Avg.	1.85	5.44	1.22	5.77	**0.56**	**1.23**
50×15	70	4.79	7.38	4.49	7.37	**2.08**	**3.88**
	80	4.66	6.86	4.14	6.62	**1.80**	**3.55**
	90	3.47	5.56	3.46	5.36	**1.15**	**2.82**
	Avg.	4.30	6.60	4.03	6.45	**1.68**	**3.42**
50×20	70	5.97	9.70	5.64	9.33	**1.98**	**5.05**
	80	4.55	7.57	4.52	7.65	**1.42**	**3.51**
	90	4.45	6.73	3.45	6.29	**0.98**	**3.17**
	Avg.	4.99	8.00	4.54	7.76	**1.46**	**3.91**
All		2.23	4.59	2.26	4.49	**0.69**	**1.80**

between the three algorithms, we carried a new series of experiments, having a time limit of $n \times m$ seconds as termination condition. Table 9 shows the results of these

experiments. The results show that although GA_{LS} and GA_{BS} are able to improve, their results still lay behind those achieved by GA^* significantly, especially for the largest instances.

Figures 6(b), 6(d) and 6(f) show the evolution of the average error over time (first generation and each 5 seconds) of each method for the 30×10, 50×15 and 50×20 instance sets. Even though the errors are different, we can detect some tendencies shared among the methods across the three instance sets. We can observe that GA_{BS} takes the lead early, followed by GA_{LS}. This can be explained by the efficiency of the binary search integrated in the solution builder used by GA_{BS}. During a brief time, both GA_{BS} computes marginally better solutions than GA^*. In the case of the 50×20 set, GA_{LS} is ahead of GA^* as well. However, right after that, GA^* takes the lead, and from that moment on, the difference in favor of GA^* grows over time. By the time GA_{LS} and GA_{BS} converge, GA^* is able to compute much better solutions, still having room for improvement.

In order to make the experimental study more robust, we have made a series of statistical inference tests to the results in Table 9[2]. These tests aim at detecting statistically significant differences among the algorithms, in terms of the average error of the solutions they compute over the whole set of instances.

Following (García et al., 2010; Gallardo and Cotta, 2015), we start performing an Aligned Friedman Rank Test. This is a multiple-comparison non-parametric test that detects whether there are significant differences among the results from a collection of algorithms and creates a ranking of them from the best to the worst performing one. In this test, the null hypothesis states that the rankings between the algorithms are equal. If it is rejected, we then perform a series of post-hoc procedures to compare the control algorithm (the one that is ranked first) with the other algorithms. We considered all the post-hoc procedures discussed in (García et al., 2010): Bonferroni-Dunn, Holm, Hochberg, Hommel, Holland, Rom, Finn, Finner and Li.

Table 10: Average rankings of the algorithms calculated by the Aligned Friedman Rank Test.

Position	Algorithm	Ranking
1	GA^*	132.75
2	GA_{BS}	411.43
3	GA_{LS}	429.33

Table 10 shows the average ranking computed by the Aligned Friedman Rank Test (distributed according to χ^2 with 2 degrees of freedom: 149.18). The test yielded a p-value of 1.16E-10. This means that there are significant differences among the results returned by the algorithms. Also, we can see in the ranking that the best algorithm is GA^*, followed by GA_{BS} in second place and GA_{LS} in the last position.

We carried out post-hoc procedures in order to compare the algorithm that ranked first (GA^*) with the other algorithms (GA_{LS} and GA_{BS}). In all cases, the null hypothesis states that the distributions obtained by GA^* and the other two algorithms are equal.

[2]For this purpose, we used the software available at http://sci2s.ugr.es/sicidm.

Table 11: Adjusted p-values of the post-hoc procedures, comparing GA* with GA$_{\text{LS}}$ and GA$_{\text{BS}}$.

Algorithm	p_{Bonf}	p_{Holm}	p_{Hoch}	p_{Homm}	p_{Holl}	p_{Rom}	p_{Finn}	p_{Li}
GA$_{\text{LS}}$	1.33E-60	1.33E-60	1.33E-60	1.33E-60	0.0	1.33E-60	0.0	6.65E-61
GA$_{\text{BS}}$	1.10E-53	5.49E-54	5.49E-54	5.49E-54	0.0	5.49E-54	0.0	5.49E-54

The adjusted p-values obtained by each procedure, p_{Bonf} (Bonferroni-Dunn), p_{Holm} (Holm), p_{Hoch} (Hochberg), p_{Homm} (Hommel), p_{Holl} (Holland), p_{Rom} (Rom), p_{Finn} (Finner) and p_{Li} (Li) are shown in Table 11. As can be observed, all these values are very close to 0, which means that the null hypothesis is rejected in all cases, so we can conclude that the differences are statistically significant in favor of GA*.

To further strengthen the experimental analysis, we performed a Wilcoxon's paired test to compare GA$_{\text{LS}}$ and GA$_{\text{BS}}$, with a significance level of $\alpha = 0.05$. The null hypothesis states that there are not significant differences between both algorithms, and the alternative hypothesis states that GA$_{\text{BS}}$ produces solutions with less error in average than those computed by GA$_{\text{LS}}$. The obtained p-value is 0.0048, showing statistically significant differences in favor of GA$_{\text{BS}}$.

From the experimental results, we can conclude that the proposed methods are fit for their respective purposes. The results confirm that the new solution builder improves efficiency. So, if there are tight time requirements, the best approach would be using GA$_{\text{BS}}$, since it is the fastest algorithm while computing solutions of similar quality than GA$_{\text{LS}}$ along generations. On the other hand, the results also indicate that considering more effective implementations of the procedure Feasible leads to improving the quality of the solutions computed, at the cost of longer running times. This way, if more computation time is available, GA* should be the preferred approach. In both cases, the proposed approaches outperform the genetic algorithm proposed in (Mencía et al., 2019).

8. Conclusions

Repairing infeasibility in scheduling constitutes a useful (and challenging) task. In this paper, we target infeasible job shop problems with a hard constraint on makespan and focus on the task of computing the largest subset of jobs that can be scheduled under such constraint.

This problem was tackled in (Mencía et al., 2019) by means of an efficient genetic algorithm. This algorithm relies on a solution builder that defines the search space. Building on this approach, we provide a formal analysis of the search space and the solution builder and make key enhancements to the genetic algorithm: a new solution builder aimed at improving efficiency by exploiting a binary search phase and a new procedure for testing the feasibility of different subsets of jobs aimed at improving effectiveness, which relies on a light-weight genetic algorithm. Experimental results confirm that the proposed methods bring substantial benefits in practice, allowing the genetic algorithm to compute higher quality solutions.

The results also encourage further research, opening a wide space of promising possibilities for the future, as devising new solution builders or new procedures for testing

feasibility. Another interesting line for future research is the development of alternative algorithms, and their combination with the genetic algorithms proposed herein. In this respect, devising a local search approach, or even an exact method, seems promising. Finally, future efforts will target the task of repairing infeasibility considering other scheduling problems and hard constraints defined on metrics different from the makespan.

Acknowledgements

This research is supported by the Spanish Government under projects TIN2016-79190-R and PID2019-106263RB-I00, and by the Principality of Asturias under grant IDI/2018/000176.

References

Akbari, M., Rashidi, H., Alizadeh, S. H., 2017. An enhanced genetic algorithm with new operators for task scheduling in heterogeneous computing systems. Eng. Appl. Artif. Intell. 61, 35–46.

Allahverdi, A., Aydilek, H., 2014. Total completion time with makespan constraint in no-wait flowshops with setup times. European Journal of Operational Research 238 (3), 724 – 734.

Andresen, M., Brsel, H., Mrig, M., Tusch, J., Werner, F., Willenius, P., 2008. Simulated annealing and genetic algorithms for minimizing mean flow time in an open shop. Mathematical and Computer Modelling 48 (7), 1279–1293.

Artigues, C., Lopez, P., Ayache, P., 2005. 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.

Bailey, J., Stuckey, P. J., 2005. Discovery of minimal unsatisfiable subsets of constraints using hitting set dualization. In: PADL. pp. 174–186.

Beasley, J. E., 1990. Or-library: Distributing test problems by electronic mail. J Oper Res Soc 41 (11), 1069–1072.

Beck, J. C., 2007. Solution-guided multi-point constructive search for job shop scheduling. J. Artif. Intell. Res. 29, 49–77.

Bierwirth, C., 1995. A generalized permutation approach to job shop scheduling with genetic algorithms. OR Spectrum 17, 87–92.

Branda, A., Castellano, D., Guizzi, G., Popolo, V., 2021. Metaheuristics for the flow shop scheduling problem with maintenance activities integrated. Computers & Industrial Engineering 151, 106989.

Brucker, P., Jurisch, B., Sievers, B., 1994. A branch and bound algorithm for the job-shop scheduling problem. Discret. Appl. Math. 49 (1-3), 107–127.

Brucker, P., Knust, S., 2006. Complex Scheduling. Springer.

Choi, J. Y., 2015. Minimizing total weighted completion time under makespan constraint for two-agent scheduling with job-dependent aging effects. Computers & Industrial Engineering 83, 237 – 243.

Davis, L., 1985. Applying adaptive algorithms to epistatic domains. In: Joshi, A. K. (Ed.), IJCAI. Morgan Kaufmann, pp. 162–164.

Dawande, M., Gavirneni, S., Rachamadugu, R., 2006. Scheduling a two-stage flowshop under makespan constraint. Mathematical and Computer Modelling 44 (1), 73 – 84.

Della Croce, F., Gupta, J. N., Tadei, R., 2000. Minimizing tardy jobs in a flowshop with common due date. European Journal of Operational Research 120 (2), 375 – 381.

Della Croce, F., Koulamas, C., T'Kindt, V., 2017. A constraint generation approach for two-machine shop problems with jobs selection. Eur. J. Oper. Res. 259 (3), 898–905.

Della Croce, F., Tadei, R., Volta, G., 1995. A genetic algorithm for the job shop problem. Comput. Oper. Res. 22 (1), 15–24.

Deng, G., Su, Q., Zhang, Z., Liu, H., Zhang, S., Jiang, T., 2020. A population-based iterated greedy algorithm for no-wait job shop scheduling with total flow time criterion. Engineering Applications of Artificial Intelligence 88, 103369.

Gallardo, J. E., Cotta, C., 2015. A grasp-based memetic algorithm with path relinking for the far from most string problem. Eng. Appl. Artif. Intell. 41, 183–194.

García, S., Fernández, A., Luengo, J., Herrera, F., 2010. Advanced nonparametric tests for multiple comparisons in the design of experiments in computational intelligence and data mining: Experimental analysis of power. Inf. Sci. 180 (10), 2044–2064.

Garey, M., Johnson, D., Sethi, R., 1976. The complexity of flowshop and jobshop scheduling. Mathematics of Operations Research 1 (2), 117 – 129.

Giffler, B., Thompson, G. L., 1960. Algorithms for solving production scheduling problems. Operations Research 8, 487–503.

Gonalves, J., Mendes, J., Resende, M., 2008. A genetic algorithm for the resource constrained multi-project scheduling problem. European Journal of Operational Research 189 (3), 1171–1190.

Gonçalves, J. F., Resende, M. G. C., 2014. An extended akers graphical method with a biased random-key genetic algorithm for job-shop scheduling. Int. Trans. Oper. Res. 21 (2), 215–246.

Gong, G., Deng, Q., Chiong, R., Gong, X., Huang, H., 2019. An effective memetic algorithm for multi-objective job-shop scheduling. Knowledge-Based Systems 182, 104840.

Graham, R., Lawler, E., Lenstra, J., Kan, A., 1979. Optimization and approximation in deterministic sequencing and scheduling: a survey. Annals of Discrete Mathematics 5, 287 – 326.

Guyon, O., Lemaire, P., Pinson, E., Rivreau, D., 2014. Solving an integrated job-shop problem with human resource constraints. Annals OR 213 (1), 147–171.

Holland, J., 1975. Adaptation in Natural and Artificial Systems. University of Michigan Press, Ann Arbor.

Hosseinabadi, A. A. R., Vahidi, J., Saemi, B., Sangaiah, A. K., Elhoseny, M., 2019. Extended genetic algorithm for solving open-shop scheduling problem. Soft Comput. 23 (13), 5099–5116.

Kolisch, R., 1996. Serial and parallel resource-constrained project scheduling methods revisited: Theory and computation. European Journal of Operational Research 90 (2), 320 – 333.

Kundakc, N., Kulak, O., 2016. Hybrid genetic algorithms for minimizing makespan in dynamic job shop scheduling problem. Computers & Industrial Engineering 96, 31–51.

Kurdi, M., 2015. A new hybrid island model genetic algorithm for job shop scheduling problem. Computers & Industrial Engineering 88, 273 – 283.

Lee, C., 2018. A review of applications of genetic algorithms in operations management. Engineering Applications of Artificial Intelligence 76, 1 – 12.

Liao, X., Zhang, H., Koshimura, M., Huang, R., Yu, W., 2019. Maximum satisfiability formulation for optimal scheduling in overloaded real-time systems. In: Nayak, A. C., Sharma, A. (Eds.), PRICAI. Vol. 11670 of Lecture Notes in Computer Science. Springer, pp. 618–631.

Marques-Silva, J., Heras, F., Janota, M., Previti, A., Belov, A., 2013. On computing minimal correction subsets. In: IJCAI. pp. 615–622.

Marques-Silva, J., Mencía, C., 2020. Reasoning about inconsistent formulas. In: IJCAI. pp. 4899–4906.

Meeran, S., Morshed, M. S., 2012. A hybrid genetic tabu search algorithm for solving job shop scheduling problems: a case study. J. Intell. Manuf. 23 (4), 1063–1078.

Mencía, C., Sierra, M. R., Mencía, R., Varela, R., 2019. Evolutionary one-machine scheduling in the context of electric vehicles charging. Integrated Computer-Aided Engineering 26, 49–63.

Mencía, C., Sierra, M. R., Varela, R., 2014. Intensified iterative deepening a* with application to job shop scheduling. J. Intelligent Manufacturing 25 (6), 1245–1255.

Mencía, R., Mencía, C., Varela, R., 2019. Repairing infeasibility in scheduling via genetic algorithms. In: de Vicente, J. M. F., Sánchez, J. R. Á., de la Paz López, F., Toledo-Moreo, F. J., Adeli, H. (Eds.), 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. Vol. 11487 of Lecture Notes in Computer Science. Springer, pp. 254–263.

Mencía, R., Mencía, C., Varela, R., 2020. A memetic algorithm for restoring feasibility in scheduling with limited makespan. Natural Computing (Online First).

Mencía, R., Sierra, M. R., Mencía, C., Varela, R., 2015. Memetic algorithms for the job shop scheduling problem with operators. Appl. Soft Comput. 34, 94–105.

Mencía, R., Sierra, M. R., Mencía, C., Varela, R., 2015. Schedule generation schemes and genetic algorithm for the scheduling problem with skilled operators and arbitrary precedence relations. In: Proc. of ICAPS. AAAI Press, pp. 165–173.

Moore, J. M., 1968. An n job, one machine sequencing algorithm for minimizing the number of late jobs. Management Science 15 (1), 102–109.

Mustu, S., Eren, T., 2018. The single machine scheduling problem with sequence-dependent setup times and a learning effect on processing times. Applied Soft Computing 71, 291–306.

Nowicki, E., Smutnicki, C., 2005. An advanced tabu search algorithm for the job shop problem. Journal of Scheduling 8, 145–159.

Ono, I., Yamamura, M., Kobayashi, S., 1996. A genetic algorithm for job-shop scheduling problems using job-based order crossover. In: Proceedings of 1996 IEEE International Conference on Evolutionary Computation, Nayoya University, Japan, May 20-22, 1996. pp. 547–552.

Palacios, J. J., Vela, C. R., Rodríguez, I. G., Puente, J., 2014. Schedule generation schemes for job shop problems with fuzziness. In: Proc. of ECAI. pp. 687–692.

Peng, B., L, Z., Cheng, T., 2015. A tabu search/path relinking algorithm to solve the job shop scheduling problem. Computers & Operations Research 53, 154 – 164.

Taillard, E., 1993. Benchmarks for basic scheduling problems. European Journal of Operational Research 64 (2), 278–285.

Talbi, E., 2009. Metaheuristics - From Design to Implementation. Wiley.

Tan, M., Yang, H. L., Su, Y. X., 2019. Genetic algorithms with greedy strategy for green batch scheduling on non-identical parallel machines. Memetic Comp. 11, 439–452.

Vallada, E., Ruiz, R., 2011. A genetic algorithm for the unrelated parallel machine scheduling problem with sequence dependent setup times. Eur. J. Oper. Res. 211 (3), 612–622.

Vilím, P., Laborie, P., Shaw, P., 2015. Failure-directed search for constraint-based scheduling. In: CPAIOR. pp. 437–453.

Yamada, T., Nakano, R., 1995. A genetic algorithm with multi-step crossover for job-shop scheduling problems. In: First International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications. pp. 146–151.

Zhang, C. Y., Li, P., Rao, Y., Guan, Z., 2008. A very fast TS/SA algorithm for the job shop scheduling problem. Computers and Operations Research 35, 282–294.

Zhang, G., Hu, Y., Sun, J., Zhang, W., 2020. An improved genetic algorithm for the flexible job shop scheduling problem with multiple time constraints. Swarm and Evolutionary Computation 54, 100664.