Nat Comput (2014) 13:179–192
DOI 10.1007/s11047-013-9373-x

A genetic algorithm for job-shop scheduling with operators enhanced by weak Lamarckian evolution and search space narrowing

Raúl Mencía · María R. Sierra · Carlos Mencía · Ramiro Varela

Published online: 16 May 2013

Abstract The job-shop scheduling problem with operators is a very interesting problem that generalizes the classic job-shop problem in such a way that an operation must be algorithm to solve this problem considering makespan minimization. The genetic algorithm uses permutations with repetition to encode chromosomes and a schedule generation scheme, termed *OG&T*, as decoding algorithm. This combination guaranties that at least one of the chromosomes represents and optimal schedule and, at the samhat machines and operators are idle while an operation is available to be processed. To improve the quality of the schedules for large instances, we use Lamarckian evolution and modify the *OG&T* algorithm to further reduce the idle time of the machines and operators, in this case at the risk of leaving all optimal schedules out of the search space. We conducted a large experimental study showing that these improvements allow the genetic algorithm to reach high quality solutions in very short time, and so it is quite competitive with the current state-of-the-art methods.

Keywords Job shop scheduling problem with operators · Genetic algorithms · Lamarckian evolution · Schedule generation schemes

1 Introduction

In this paper we propose a genetic algorithm (GA) to solve the job-shop scheduling problem with operators. This problem has been recently proposed in (Agnetis et al. 2011) and it is motivated by manufacturing processes in which part of the work is done by human operators sharing the same set of tools. The problem is formalized as a classical job-shop problem in which the processing of an operation on a given machine requires the assistance of one of the p available operators. In (Agnetis et al. 2011) the problem is studied and the minimal *NP*-hard cases are established. Also, a number of exact and approximate algorithms to cope with this problem are proposed and evaluated on a set of instances generated from that minimal relevant cases. The result of their experimental study makes it clear that instances with 3 jobs, 3 machines, 2 operators and a number of 30 operations per job are hard to solve to optimality.

The proposed GA exploits the permutation with repetition schema proposed in (Bierwirth 1995) for the classic job-shop scheduling problem. In order to get feasible schedules for the problem with operators, we use the schedule generation scheme, termed *OG&T*, proposed in (Sierra et al. 2013) to design a decoding algorithm. Combining these two elements, it is guaranteed that at least one chromosome represents an optimal schedule and, at the same time, the periods of time that machines and operators are idle while they can be used to process some operation are limited. So the GA searches for solutions over a reduced space with good solutions in average.

In order to improve the performance of the GA, we introduce two more elements. On the one hand, we exploit Lamarckian evolution. This kind of evolution is often used when a local searcher is combined with a genetic algorithm as it allows for the transmission of the learned characteristics from parents to offsprings. This technique was shown efficient, for example, in (González et al. 2012) where the authors proposed a memetic algorithm for a job-shop

R. Mencía · M. R. Sierra (✉) · C. Mencía · R. Varela
Department of Computer Science, University of Oviedo, Campus de Viesques s/n, Gijón 33271, Spain
e-mail: sierramaria@uniovi.es
URL: http://www.di.uniovi.es/iscop

scheduling problem with setup times. Even though we do not use any local searcher herein, we can exploit Lamarckian evolution due to the fact that the starting times of the operations in the schedule built by the *OG&T* decoder do not keep the same order as the operations in the chromosome encoding. So, the order given by the starting times may be coded back in the chromosome and so it can be easily transmitted to the offsprings by crossover an mutations. We call weak Lamarckian evolution to this procedure due to the fact that it is expected to produce much less changes in the chromosome than a classical local searcher.

On the other hand, we modified the strategy of the *OG&T* algorithm in order to further reduce the idle times of the machines and operators. In this way, we restrict the search to a smaller space in which the schedules are better in average. However, we have the risk of leaving all optimal schedules out of the search space. This search space narrowing strategy is similar to that used in (Bierwirth and Mattfeld 1999) for the classic job-shop scheduling problem and it is implemented by introducing a parameter $\delta \in [0, 1]$ to control the idle time intervals.

To assess the performance of the proposed GA, we have conducted a thorough experimental study across instances of different sizes and characteristics. The results of this study show that the genetic algorithm outperforms the approximate state-of-the-art algorithms, that as far as we know are those proposed in (Agnetis et al. 2011), and compares favorably with the exact methods proposed in (Agnetis et al. 2011) and (Sierra et al. 2013). The study also shows that the two new elements proposed allow GA to perform much more effectively.

The remaining of the paper is organized as follows. In Sect. 2 we define the job-shop scheduling problem with operators and give a disjunctive model to represent instances and schedules. In Sect. 3 we describe the original *OG&T* algorithm and how it may be adapted to search on a restricted space. Section 4 presents the main components of the GA used and discuss the two evolution models considered. Section 5 is devoted to the experimental study. Finally, we summarize the main conclusions of the paper and propose some ideas for future research in Sect. 6.

2 Description of the problem

Formally the job-shop scheduling problem with operators can be defined as follows. We are given a set of n jobs $\{J_1, \ldots, J_n\}$, a set of m resources or machines $\{R_1, \ldots, R_m\}$ and a set of p operators $\{O_1, \ldots, O_p\}$. Each job J_i consists of a sequence of v_i operations or tasks $(\theta_{i1}, \ldots, \theta_{iv_i})$. Each task θ_{il} has a single resource requirement $R_{\theta_{il}}$, an integer duration $p_{\theta_{il}}$ and a start time $st_{\theta_{il}}$ to be determined. A feasible schedule is a complete assignment of starting times and operators to operations that satisfies the following constraints: (i) the operations of each job are sequentially scheduled, (ii) each machine can process at most one operation at any time, (iii) no preemption is allowed and (iv) each operation is assisted by one operator and one operator cannot assist more than one operation at a time. The objective is finding a feasible schedule that minimizes the completion time of all the operations, i.e. the makespan. This problem was first defined in (Agnetis et al. 2011) and is denoted as $JSO(n, p)$. The significant cases of this problem are those with $p < min(n, m)$, otherwise the problem is a standard job-shop problem denoted as $J||C_{max}$.

We use the following disjunctive model for the $JSO(n, p)$. A problem instance is represented by a directed graph $G = (V, A \cup E \cup I \cup O)$. Each node in the set V represents either an actual operation, or any of the fictitious operations introduced with the purpose of giving the graph a particular structure: starting and finishing operations for each operator i, denoted O_i^{start} and O_i^{end} respectively, and the the dummy operations *start* and *end*.

The arcs in A are called *conjunctive arcs* and represent precedence constraints among operations of the same job. The arcs in E are called *disjunctive arcs* and represent capacity constraints. E is partitioned into subsets E_i with $E = \cup_{\{i=1,\ldots,M\}} E_i$. E_i includes an arc (v, w) for each pair of operations requiring the resource R_i. The set O of *operator arcs* includes three types of arcs: one arc (u, v) for each pair of operations of the problem, and arcs (O_i^{start}, u) and (u, O_i^{end}) for each operator node and operation. The set I includes arcs connecting node *start* to each node O_i^{start} and arcs connecting each node O_i^{end} to node *end*. The arcs are weighted with the processing time of the operation at the source node.

From this representation, building a solution can be viewed as a process of fixing disjunctive and operator arcs. A disjunctive arc between operations u and v gets fixed when one of (u, v) or (v, u) is selected and consequently the other one is discarded. An operator arc between u and v is fixed when (u, v), (v, u) or none of them is selected, and fixing the arc (O_i^{start}, u) means discarding (O_i^{start}, v) for any operation v other than u. Analogously for (u, O_i^{end}).

Therefore, a feasible schedule S is represented by an acyclic subgraph of G, of the form $G_S = (V, A \cup H \cup I \cup Q)$, where H expresses the processing order of operations on the machines and Q expresses the sequences of operations that are assisted by each operator. The makespan is the cost of a *critical path* in G_S. A critical path is a longest cost path from node *start* to node *end*.

Figure 1 shows a solution graph for an instance with 3 jobs, 3 machines and 2 operators. Discontinuous arrows

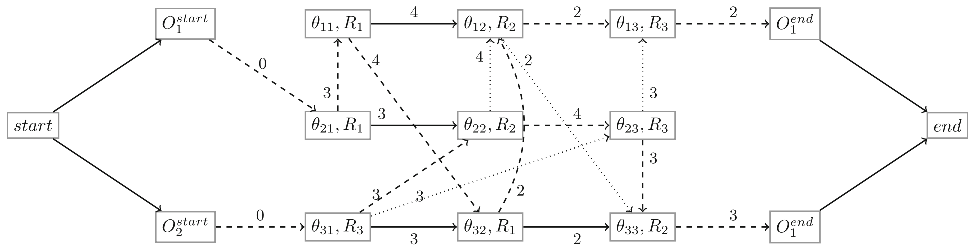

Fig. 1 A feasible schedule to a problem with 3 jobs, 3 machines and 2 operators

represent operator arcs. So, the sequences of operations assisted by operators O_1 and O_2 are (θ_{21}, θ_{11}, θ_{32}, θ_{12}, θ_{13}) and (θ_{31}, θ_{22}, θ_{23}, θ_{33}) respectively. In order to simplify the picture, only the operator arc is drawn when there are two arcs between the same pair of nodes. Continuous arrows represent conjunctive arcs and doted arrows represent disjunctive arcs; in these cases only arcs not overlapping with operator arcs are drawn. In this example, the critical path is given by the sequence (θ_{21}, θ_{11}, θ_{32}, θ_{12}, θ_{33}), so the makespan is 14.

In order to simplify expressions, we define the following notation for a feasible schedule. The *head* r_v of an operation v is the cost of the longest path from node *start* to node v, i.e. it is the value of st_v. The *tail* q_v is defined so as the value $q_v + p_v$ is the cost of the longest path from v to *end*. Hence, $r_v + p_v + q_v$ is the makespan if v is in a critical path, otherwise, it is a lower bound. PM_v and SM_v denote the predecessor and successor of v respectively on the machine sequence, PJ_v and SJ_v denote the predecessor and successor operations of v respectively on the job sequence and PO_v and SO_v denote the predecessor and successor operations of v respectively on the operator of v.

A partial schedule is given by a subgraph of G where some of the disjunctive and operator arcs are not fixed yet. In such a schedule, heads and tails can be estimated as

$$r_v = \max\left\{ \max_{J\subseteq P(v)} \left\{ \min_{j\in J} r_j + \sum_{j\in J} p_j \right\}, \max_{J\subseteq PO(v)} \left\{ \min_{j\in J} r_j + \sum_{j\in J} p_j \right\}, r_{PJ_v} + p_{PJ_v} \right\} \tag{1}$$

$$q_v = \max\left\{ \max_{J\subseteq S(v)} \left\{ \sum_{j\in J} p_j + \min_{j\in J} q_j \right\}, \max_{J\subseteq SO(v)} \left\{ \sum_{j\in J} p_j + \min_{j\in J} q_j \right\}, p_{SJ_v} + q_{SJ_v} \right\} \tag{2}$$

with $r_{start} = q_{end} = r_{O_i^{start}} = q_{O_i^{end}} = 0$ and where $P(v)$ denotes the disjunctive predecessors of v, so as for all $w \in P(v)$, $R_w = R_v$ and the disjunctive arc (w, v) is already fixed (analogously, $S(v)$ denotes the disjunctive successors of v). $PO(v)$ denotes the operator predecessors of v, i.e $w \in PO(v)$ if it is already established that $O_w = O_v$ and w is processed before v, so as the operator arc (w, v) is fixed (analogously, $SO(v)$ are the operator successors of v).

3 Schedule generation schemes

In (Giffler and Thompson 1960), the authors proposed a schedule generation scheme for the $J||C_{max}$ problem termed *G&T* algorithm. This algorithm has been used in a variety of settings for job-shop scheduling problems. For example, in (Brucker et al. 1994; Mencía et al. 2012; Mencía et al. 2013) it was used to devise a greedy algorithm, in (Bierwirth 1995) and (Mattfeld 1995) it was exploited as a decoder for genetic algorithms, and in (Sierra and Varela 2010) it was used to define the state space for a best-first search algorithm. Unfortunately, for other scheduling problems the original *G&T* algorithm is not longer suitable, but it has inspired the design of similar schedule builders such as the *EG&T* algorithm proposed in (Artigues et al. 2005) for the job-shop scheduling problems with sequence dependent setup times. In this section we propose to use a schedule generation scheme for the $JSO(n, p)$ which is also inspired in the *G&T* algorithm. This scheme is termed *OG&T* and it has been proposed in (Sierra et al. 2013). In the remainder of this section, we summarize the main characteristics of the *OG&T* algorithm.

As it is done by the *G&T* algorithm, the operations are scheduled one at a time in sequential order within each job. When an operation u is scheduled, it is assigned a starting time st_u and an operator O_i, $1 \le i \le p$. Let SC be the set of scheduled operations at a given time and G_{SC} the partial solution graph built so far. For each operation u in SC, all operations in $P(u)$ or $PO(u)$ are scheduled as well. So, there is a path in G_{SC} from the node O_i^{start} to u through all operations in $PO(u)$ and a path from *start* to u through all operations in $P(u)$. Let A be the set that includes the first unscheduled operation of each job that has at least one unscheduled operation, i.e.

$$A = \{u \notin SC; \nexists\, PJ_u \vee PJ_u \in SC\} \tag{3}$$

For each operation u in A, r_u is the starting time of u if u is selected to be scheduled next. In accordance with expression (1), r_u is greater or at least equal than both the completion time of PJ_u and the completion time of the last operation scheduled on the machine R_u. Moreover, the value of r_u also depends on the availability of operators at this time, hence r_v is not lower than the earliest time an operator is available. Let t_i, $1 \leq i \leq p$, be the time at which the operator i is available, then

$$r_u = \max\left\{ r_{PJ_u} + p_{PJ_u}, r_v + p_v, \min_{1 \leq i \leq p} t_i \right\} \tag{4}$$

where v denotes the last operation scheduled having $R_v = R_u$. In general, a number of operations in A could be scheduled simultaneously at their current heads, however it is clear that not all of them could start processing at these times due to both capacity constraints and operators availability. So, a straightforward schedule generation scheme is obtained if each one of the operations in A is considered as candidate to be scheduled next.

If the selected operation is u, it is scheduled at the time $st_u = r_u$ and all the disjunctive arcs of the form (u, v), for all $v \notin P(u)$, get fixed. Operator arcs should also be fixed from the set of scheduled operations assisted by any of the operators. In principle, any operator i with $t_i \leq r_u$ can be selected for the operation u. So, if we start from the set A containing the first operation of each job and iterate until all the operations get scheduled, no matter what operation is selected in each step, we will finally have a feasible schedule, and there is a sequence of choices eventually leading to an optimal schedule.

Let us now consider the operation v^* with the earliest completion time if it is selected from the set A, i.e.

$$v^* = \arg\min\{r_u + p_u; u \in A\} \tag{5}$$

and the set A' given by the operations in A that can start before the earliest completion of v^*, i.e.

$$A' = \{u \in A; r_u < r_{v^*} + p_{v^*}\} \tag{6}$$

If we restrict the selection, in each step of the previous scheduling algorithm, to an operation in A', at least one of the schedules that may be eventually reached is optimal. The reason for this is that for every operation u in A not considered in A', the possibility of being scheduled at a time $st_u = r_u^A$ remains in the next step.

Moreover, if the number of operators available is large enough, it is not necessary to take all the operations in the set A' as candidate selections. Let $[T, C)$ be a time interval, where $C = r_{v^*} + p_{v^*}$ and $T = min\{r_u; u \in A'\}$, and the set of machines $R_{A'} = \{R_u; u \in A'\}$. If we consider the simplified situation where $r_u = T$, for all $u \in A'$ we can do the following reasoning: if, for instance, the number of machines in $R_{A'}$ is two and there are two or more operators available along $[T, C)$, then the set A' can be reduced to the operations requiring the machine R_{v^*}. In other words, we can do the same as it is done in the *G&T* algorithm for the classical job-shop problem. The reason for this is that after selecting an operation v requiring R_{v^*} to be scheduled, every operation $u \in A'$ requiring the other machine can still be scheduled at the same starting time as if it were scheduled before v, so as this machine may not be considered in the current step. However, if there is only one operator available along $[T, C)$ then A' may not be reduced, otherwise the operations removed from A' will no longer have the possibility of being processed at their current heads.

The reasoning above can be extended to the case where p' operators are available along $[T, C)$ and the number of machines in $R_{A'}$ is $m' \geq p'$. In this case A' can be reduced to maintain the operations of only $m' - p' + 1$ machines in order to guarantee that all the operations in A' have the opportunity to get scheduled at their heads in G_{SC}.

In general, the situation is more complex as the heads of operations in A' are distributed along the interval $[T, C)$ as well as the times at which the operators get available. Let $\tau_0 < \ldots < \tau_{p'}$ be the times given by the head of some operation in A' or the time at which some operator becomes available along the interval $[T, C)$. It is clear that $\tau_0 = T$ and $\tau_{p'} < C$. Let $NO_{\tau_i}, i \leq p'$, be the number of operators available in the interval $[\tau_i, \tau_{i+1})$, with $\tau_{p'+1} = C, R_{\tau_i}$ the set of machines that are required before τ_{i+1}, i.e $R_{\tau_i} = \{R_u; r_u \leq \tau_i\}$ and NR_{τ_i} be the number of machines in R_{τ_i}.

We now consider the time intervals from $[\tau_{p'}, C)$ backwards to $[T, \tau_1)$. If $NO_{\tau_i} \geq NR_{\tau_i}$ then only the operations of just one of the machines in R_{τ_i} should be maintained in A' due to the interval $[\tau_i, \tau_{i+1})$. Otherwise, i.e. if $NO_{\tau_i} < NR_{\tau_i}$, then the operations from at least $NR_{\tau_i} - NO_{\tau_i} + 1$ machines must be maintained from A' in order to guarantee that any operation requiring a machine in R_{τ_i} could be eventually processed along the interval $[\tau_i, \tau_{i+1})$. In other words, in this way we guarantee that any combination of NO_{τ_i} operations requiring different machines can be processed along the interval $[\tau_i, \tau_{i+1})$, as at least one operation requiring each of the $NR_{\tau_i} - NO_{\tau_i} + 1$ machines will appear in such combination of operations.

Here, it is important the following remark. As it was pointed, in principle any operator available at time r_u is suitable for u. However, we now have to be aware of leaving available operators for the operations that are removed from A'. In order to do that we select an operator i available at the latest time t_i such that $t_i \leq r_u$. In this way, we maximize the availability of operators for the operations to be scheduled in the next steps. To be more precise, we guarantee that any operation removed from A' can be scheduled at its current head in a subsequent step.

From the interval $[\tau_{p'}, C)$ at least the operation v^* is maintained and then some new operations are added from the remaining intervals. The set of operations obtained in this way is termed B and it is clear that $|B| \leq |A'| \leq |A|$. An important property of this schedule generation scheme is that if the number of operators is large enough, in particular if $p \geq \min(n, m)$ so as $JSO(n, p)$ becomes $J||C_{max}$, it is equivalent to the *G&T* algorithm. So, we call this new algorithm *OG&T* (Operators *G&T*). From the reasoning above the following result can be established.

Theorem 1 *Let $\mathcal{P}$ be a JSO(n, p) instance. The set $\mathcal{S}$ of schedules that can be obtained by the OG&T algorithm to $\mathcal{P}$ is dominant, i.e. $\mathcal{S}$ contains at least one optimal schedule.*

3.1 Narrowing of the search space

In its original formulation, the *G&T* algorithm was shown to be able to generate all the possible active schedules for the classic job-shop problem. In these schedules, at least one operation has to be delayed in order to start the processing of another operation earlier, i.e., there are no machines idle along a whole period where an operation could be completely processed. Active schedules constitute a subset of the feasible schedules that contains at least one optimal solution.

The concept of active schedule has not yet been formalized for the $JSO(n, p)$ problem, and it is not trivial due to the limited number of assisting operators. Nevertheless, the reduction made by the *OG&T* algorithm from the set A to A', taking only the operations that can start before the earliest possible completion time of v^*, guarantees that no operation can be processed earlier in a schedule without delaying at least other operation.

Although the set of schedules defined by the *OG&T* algorithm is smaller than the set of the feasible schedules, it may be still huge for large instances. For this reason, we propose extending this algorithm in order to achieve further reductions, yet loosing the possibility of finding an optimal solution. Concretely, we reduce B to the B' defined as

$$B' = \{u \in B; r_u \leq \min\{r_w; w \in B\} + \delta \times (r_{v^*} + p_{v^*} - \min\{r_w; w \in B\})\} \quad (7)$$

where δ is a real parameter such that $0 \leq \delta \leq 1$.

Hence, the parameter δ establishes the upper limit for the starting time of the operation to be scheduled next, and so it controls the maximum time that a machine can be idle when an operation is ready to be processed on that machine and an operator is available to assist that processing. It is clear that the smaller value of δ the lower idle time periods of the machines. So, it can be expected that the average makespan is lower for the schedules that remain in the search space after this reduction than it is for the whole set of schedules in the original search space. At the same time, it is clear that for $\delta < 1$ we can no longer guarantee that at least one of the chromosomes represent an optimal schedule. From all of this, it is our hypothesis that for large instances where the search space is huge it may be worthwhile to restrict the search to a reduced subspace with good solutions in average by setting δ to a small value, even though this subspace may not contain any optimal schedule. However, for small instances it may be better to set $\delta = 1$ so as the genetic algorithm can search over the whole space.

4 Genetic algorithm for the JSO(n,p)

The GA used here is inspired in that proposed in (González et al. 2008). The coding schema is based on permutations with repetition as it was proposed in (Bierwirth 1995). A chromosome is a permutation of the set of operations that represents a tentative ordering to schedule them, each one being represented by its job number. For example, the sequence (2 1 1 3 2 3 1 2 3) is a valid chromosome for a problem with 3 jobs and 3 machines. As it was shown in (Varela et al. 2005), this encoding has a number of interesting properties for the classic job-shop scheduling problem; for example, it tends to represent orders of operations as they appear in good solutions. So, it is expected that these characteristics are to be good for the $JSO(n, p)$ as well.

For chromosome mating, the GA uses the Job-based Order Crossover (JOX) described in (Bierwirth 1995). Given two parents, JOX selects a random subset of jobs and copies their genes to the offspring in the same positions as they are in the first parent, then the remaining genes are taken from the second parent so as they maintain their relative ordering. We clarify how JOX works in the next example. Let us consider the following two parents

Parent1 (**2** 1 1 3 **2** 3 1 **2** 3) Parent2 (3 3 1 2 1 3 2 2 1)

If the selected subset of jobs from the first parent just includes the job 2, the generated offspring is

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

Hence, operator JOX maintains for each machine a subsequence of operations in the same order as they are in Parent1 and the remaining in the same order as they are in Parent2.

To evaluate chromosomes, the GA uses the algorithm *OG&T* described in the previous section, so that the non-deterministic choice is done by looking at the chromosome: the operation in B which is in the leftmost position in the chromosome sequence is selected to be scheduled next.

The remaining elements of GA are rather conventional. To create a new generation, all chromosomes from the current one are organized into couples which are mated and then mutated to produce two offsprings in accordance with the crossover and mutation probabilities respectively. Finally, tournament replacement among every couple of parents and their offsprings is done to obtain the next generation.

4.1 Evolution models

Regarding the evolution model of a genetic algorithm, we can consider either Baldwinian or Lamarckian evolution. The genetic algorithm described above follows the Baldwinian model, as no information but the fitness from the phenotypes is translated back to the genotypes after the decoding algorithm is applied to the chromosomes. According to this evolution model, no characteristics learned during the life time of an individual are passed down to its offspring, as it happens in natural evolution. In this model, a trait would remain in the population as a result of selection pressure focusing towards individuals more capable of acquiring the trait.

On the contrary, in Lamarckian evolution the changes of an individual during its life are inherited by its offspring. This would happen if the characteristics acquired by the individual were coded back into the chromosome structure. It is well known that this effect does not occur in natural evolution, but it is usually implemented in artificial evolution as it allows the characteristics of the parents to be immediately transmitted to the offspring. So, a genetic algorithm exploiting this model would exhibit more selection pressure, so that the search may be biased towards better regions of the solution space. As a drawback, Lamarckian evolution may have negative effects on the diversity of the genetic material in the population, and thus causing premature convergence.

We propose a Lamarckian version of our genetic algorithm, in which the characteristics of the solutions are coded back into the chromosomes. To do so, after a schedule is built by the *OG&T* algorithm, the original chromosome is replaced in the population by a new one in which the operations appear in a compatible order with their starting times in the schedule, i.e., for any two operations u and v requiring the same machine, if $st_u < st_v$ then the gene representing u is before than that representing v in the chromosome sequence. This way, the new chromosome will hold all the information about the solution it represents, and not only the fitness value. Clearly, the differences between the new and the old chromosomes are not likely to be large, or at least they are expected to be much lower than the differences between them if a conventional local search were applied from the original chromosome. For this reason, as we have pointed out, we consider this as a weak Lamarckian evolution model. In the next section, both evolution models are empirically evaluated.

5 Experimental study

We have conducted a thorough experimental study aimed at evaluating our proposals and making a comparison with the state-of-the-art methods which, to the best of our knowledge, are the dynamic programming and the heuristic algorithms given in (Agnetis et al. 2011), and the A* search algorithm proposed in (Sierra et al. 2013).

For this purpose, we have experimented with both the Baldwinian (GA) and the Lamarckian (LGA) versions of the genetic algorithm, and have considered five different values of δ (0, 0.25, 0.5, 0.75, 1) for each of them. Among the 10 possible configurations, we will consider GA and $\delta = 1$ as a base method, as it does not present any difference with the original genetic algorithm proposed in (Mencía et al. 2011).

The experiments were carried out across two sets of instances. The first one was proposed in (Agnetis et al. 2011); all these instances have $n = 3$ and $p = 2$ and are characterized by the number of machines (m), the number of operations per job (v_{max}) and the range of processing times (p_i). A set of small instances was generated combining the values $m = 3, 5, 7$; $v_{max} = 5, 7, 10$ and $p_i = [1, 10], [1, 50], [1, 100]$ and a set of larger instances was generated with $m = 3$, combining $v_{max} = 20, 25, 30$ and $p_i = [1, 50], [1, 100], [1, 200]$; 10 instances were considered from each combination. The sets of small instances are identified by numbers from 1 to 27: set 1 corresponds to triplet $3 - 5 - 10$, the second is $3 - 5 - 50$ and so on. The large instances are identified analogously by labels from $L1$ to $L9$. There are 360 instances in all.

The second benchmark set is derived from a number of instances taken from the *OR*-library (Beasley 1990). Firstly, small instances with $m = 5$ and different number of jobs: $LA01 - 05$ with $n = 10$, $LA06 - 10$ with $n = 15$ and $LA11 - 15$ and $FT20$ with $n = 20$. Then larger instances with $m = 10$ and different number of jobs: $FT10$, $LA16 - 20$, $ABZ5, 6$ and $ORB01 - 10$ with $n = 10$, $LA21 - 25$ with $n = 15$, $LA26 - 30$ with $n = 20$ and $LA31 - 35$ with $n = 30$. Finally, the instances $LA36 - 40$ with $n = m = 15$. For each instance, all values of p in $[1, \min(n, m)]$ are considered. So, this set contains 480 instances in all.

In all the experiments, the genetic algorithms were parameterized with a population of 100 chromosomes, a number of 140 generations, crossover probability of 0.7 and mutation probability of 0.2, so that about 10,000 chromosomes are evaluated in each run. Each algorithm was run 30 times for each instance. The algorithms were coded in C++ and the target machine was Intel Xeon 2.26 GHz. 24 GB RAM.

In the remainder of this section, we first provide a comparison between the base GA and the state-of-the-art methods. Then, we analyse the effects that reducing the search space and incorporating a Lamarkian evolution scheme have on the genetic algorithm.

5.1 Comparison between GA and the state-of-the-art methods

In the first part of the experimental study, we have compared the original genetic algorithm (GA and $\delta = 1$) with the methods proposed in (Agnetis et al. 2011) and (Sierra et al. 2013).

For all the experiments, we report the mean relative error in percentage terms of the best and average solutions (*Ebest* and *Eavg* respectively) reached by GA, w.r.t. the best lower bounds given by the A^* algorithm proposed in (Sierra et al. 2013). This algorithm was able to solve optimally all the instances taken from (Agnetis et al. 2011) and a number of the instances from the second set. For the remaining ones it has given a suboptimal solution (as it combines best-first search with a greedy algorithm that is issued from a number of expanded states to obtain upper bounds) and a lower bound in a time limit of 3,600 s, or after the memory of the target machine got exhausted. We also report the time taken in average ($T(s)$) by GA and the Pearson coefficient of variation in percentage terms of the solutions (CV) computed as the ratio of standard deviation and the average of the solutions across the 30 runs.

Table 1 reports the results obtained by GA for the first set, together with the results from the best exact and approximate algorithms given in (Agnetis et al. 2011), namely the dynamic programming algorithm (DP) and the heuristic algorithms (*Heur* and *Heur+*). DP was able to solve optimally all but a number of instances from sets L4-9. In these cases, DP was stopped after 3 h, and the incumbent solution is taken as reference for calculating the gap of the heuristic algorithms. In the other cases the gap is defined as the error in percentage w.r.t. the optimal solution, averaged for all instances. For the sets L4-9, the instances that were not solved by DP are not considered in the average error computed for the methods proposed in (Agnetis et al. 2011). For GA, we report *Ebest*, *Eavg*, *CV* and $T(s)$. In all cases the results are averaged for subsets of instances with the same number of operations per job v_{max}.

The first conclusion we can draw from these experiments is that GA clearly outperforms the heuristic methods *Heur* and *Heur+* ; not only the error is much lower for GA than it is for both heuristic methods, but also the time taken is much lower for GA, in particular for the largest instances. A comparison between GA and DP is not so straightforward. As we can see, GA is not able to solve optimally all these instances in the time given whereas DP solves small instances easily. At the same time, GA is able to produce high-quality solutions in very short time regardless of the size of the instances, while DP performs

Table 1 Summary of results from instances with 3 jobs and 2 operators

		GA				T.(s)			Gap	
Instances		*Ebest*	*Eavg*	*CV*	T.(s)	*DP*	*Heur*	*Heur+*	*Heur*	*Heur+*
Small	1–9	4.50	4.51	0.03	0.18	0.03	0.01	0.04	15.33	11.22
	10–18	1.55	1.88	0.12	0.18	1.29	0.04	0.47	14.81	10.98
	19–27	1.07	1.28	0.23	0.26	14.93	0.47	4.86	15.31	11.12
Large	L1-L3	1.43	2.37	0.50	0.50	654.10	9.40	91.47	17.80	12.90
	L4-L6	1.82	2.97	0.58	0.64	1683.60	35.77	366.73	15.40	11.97
	L7-L9	2.53	3.96	0.68	0.73	4351.00	97.00	823.97	16.50	11.57

Table 2 Summary of results from instances with 5 machines and 10, 15 and 20 jobs

	10 *jobs*			15 *jobs*			20*jobs*		
	(T=0.4s, CV=0.2)			(T=0.8s, CV=0.1)			(T=1.1s, CV=0.1)		
#Op.	*Ebest*	*Eavg*	*EA**	*Ebest*	*Eavg*	*EA**	*Ebest*	*Eavg*	*EA**
1	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
2	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
3	0.00	0.18	0.00	0.00	0.07	0.00	0.01	0.12	0.05
4	1.20	2.20	0.24	0.20	0.71	0.00	0.98	1.49	1.27
5	0.30	1.27	0.00	0.00	0.00	0.00	0.70	1.14	0.00
Avg.	0.30	0.73	0.05	0.04	0.16	0.00	0.34	0.55	0.26

poorly on large instances, either taking a long time to solve them or not even being able to come up with a single solution by a time limit of 3 h. So, DP is well suited for solving small instances and GA may be a better option for large ones. It is also remarkable that GA seems to be very stable, as the value of *CV* is very low in all cases and increases very slightly with the size of the problem.

The second set of instances is more interesting than the first one, as it allows us to analyze the behavior of the algorithms in a broader range of settings regarding the numbers of machines, jobs and available operators. Here, we will only consider GA and A*, as there are no results available in the literature from Agnetis et al's methods. Besides, these methods were shown to be clearly outperformed by the A* algorithm in (Sierra et al. 2013). Tables 2, 3, and 4 are devoted to this set, and share a similar structure: they show the errors in percentage obtained by GA (*Ebest* and *Eavg*) and by A* (*EA**) averaged for subsets of instances with the same size and number of operators. They also report, for GA, the time taken ($T(s)$) and *CV* averaged for subsets of instances with the same size.

One important remark about these experiments is that A* was given more time and space than GA, so a strict comparison between the two methods is not possible. Nevertheless, these results provide an insight into the effectiveness that can be achieved by GA with respect to the best known solutions to the problem.

Table 2 reports the results across instances with 5 machines and 10, 15 and 20 jobs. These are small instances and so they were almost all solved to optimality by the A* algorithm. As we can see, GA is able to reach very good solutions, finding an optimum in many cases. Even for some of the largest instances with 20 jobs that A* did not solve optimally, GA reached better solutions in less than one second.

Table 3 shows the results from larger instances with 10 machines and 10, 15, 20 and 30 jobs, which are really hard to solve. First of all, we can note that both GA and A* solved to optimality all the instances with 1 or 2 operators, regardless of the number of jobs. In the remaining cases, there seems to be a trend that GA performs better than A* as long as the size of the instances grows. A* reaches better

Table 4 Summary of results from instances with 15 jobs and 15 machines

	GA		
	(T=4.3s, CV=0.4)		
#Op.	*Ebest*	*Eavg*	EA^*
1	0.00	0.00	0.00
2	0.00	0.00	0.00
3	0.00	0.03	0.00
4	0.07	0.17	0.00
5	0.25	0.45	1.59
6	0.67	1.01	2.11
7	1.42	2.09	3.11
8	3.34	4.58	4.89
9	7.80	9.66	8.46
10	10.02	13.17	13.02
11	10.20	12.94	10.63
12	9.09	12.65	9.76
13	9.96	12.62	9.75
14	9.98	12.77	9.68
15	9.61	12.64	9.64
Avg.	4.83	6.32	5.51

Table 3 Summary of results from instances with 10 machines and 10, 15, 20 and 30 jobs

	10 *jobs*			15 *jobs*			20 *jobs*			30 *jobs*		
	(T=1.1s, CV=0.4)			(T=2.4s, CV=0.3)			(T=3.8s, CV=0.3)			(T=7.1s, CV=0.1)		
#Op.	*Ebest*	*Eavg*	EA^*	*Ebest*	*Eavg*	EA^*	*Ebest*	*Eavg*	EA^*	*Ebest*	*Eavg*	EA^*
1	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
2	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
3	0.03	0.16	0.21	0.00	0.04	0.00	0.00	0.02	0.00	0.00	0.01	0.00
4	0.69	1.24	0.65	0.07	0.24	0.00	0.04	0.14	0.26	0.01	0.06	0.32
5	3.01	4.20	3.26	0.36	0.73	1.41	0.21	0.43	1.63	0.10	0.20	0.80
6	6.69	8.92	6.40	1.17	1.89	5.71	0.60	1.01	3.45	0.26	0.45	3.25
7	5.17	7.46	4.14	3.30	4.76	3.12	1.71	2.57	2.92	0.67	1.01	1.96
8	2.54	4.94	1.05	7.70	9.95	7.58	4.82	6.52	4.83	1.76	2.55	2.94
9	2.52	4.80	0.80	7.21	10.19	9.04	8.14	10.40	9.45	1.91	3.43	4.48
10	2.67	4.89	0.89	6.74	9.48	6.91	7.37	9.62	7.44	0.49	1.46	2.48
Avg.	2.33	3.66	1.74	2.66	3.73	3.38	2.29	3.07	3.00	0.52	0.92	1.62

solutions than GA for most of the smallest instances with 10 jobs. For instances with 15 and 20 jobs, the best solutions reached by GA are better in average than those obtained by A*. Moreover, regarding the average makespan, GA outperforms A* in most of the cases having an intermediate numbers of available operators: instances with 5, 6 and 7 available operators and 15 jobs and instances with 4, 5, 6 and 7 operators and 20 jobs. For the largest and hardest instances in the whole experimental study, with 30 jobs and 10 machines, GA outperforms A* in all cases, reducing the average error by about 43.21 % in average and 67.90 % in the best case.

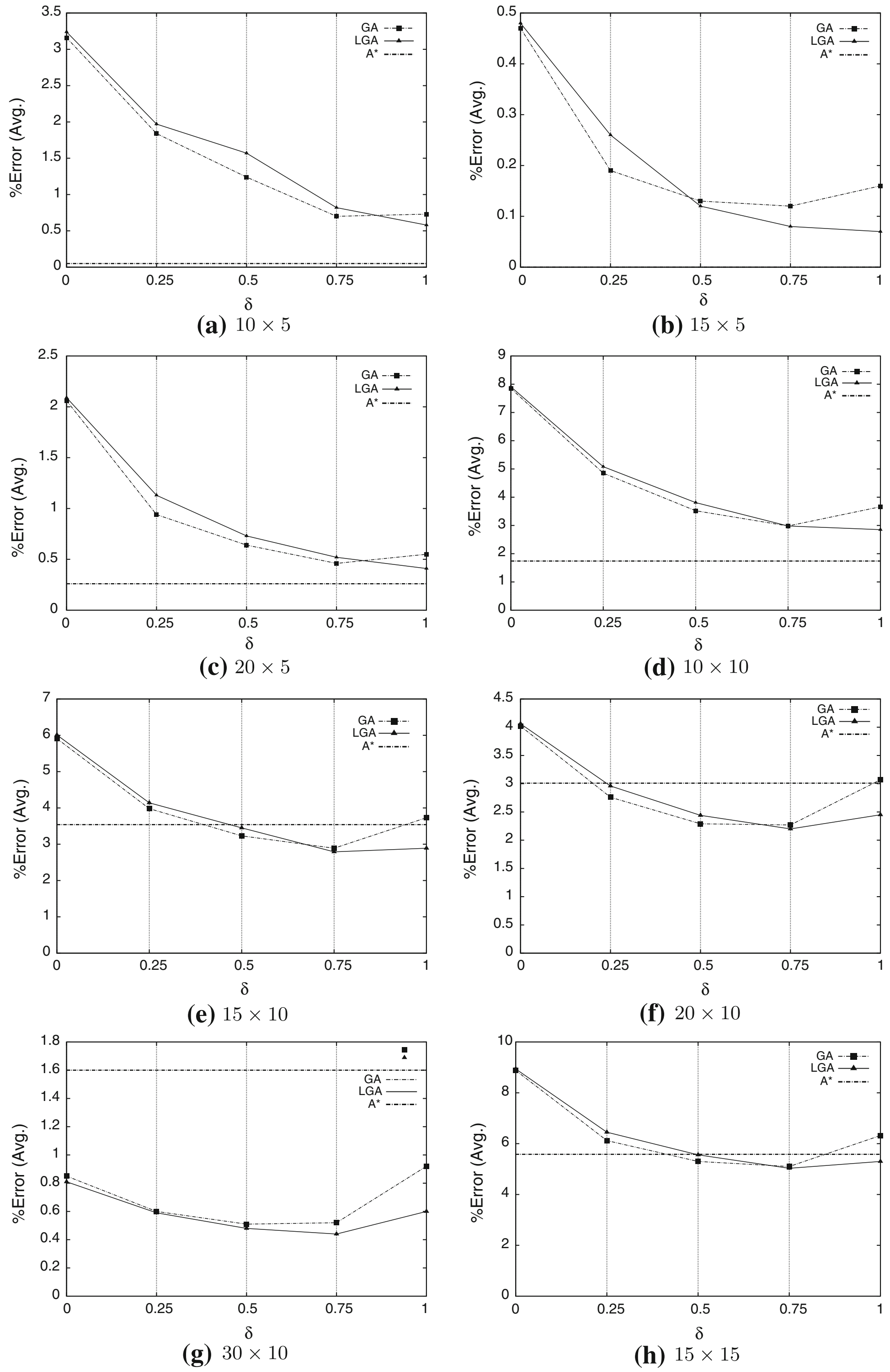

Fig. 2 Errors in percentage from GA, LGA and A* across the second set averaged for instances with the same size and number of operators

Finally, Table 4 shows results from instances with 15 jobs and 15 machines. The results are fairly similar to those from instances with 20 jobs and 10 machines. Overall, the best solutions computed by GA are better than those returned by A*. Also, GA outperforms A* in average solving instances with an intermediate number of operators (5, 6, 7 and 8). In the remaining cases, A* reaches the best solutions.

We can also observe that GA is able to solve all the instances very quickly. Indeed, it only takes an average time of 7.1 seconds to solve the largest ones. GA is also very stable, as shown by the very low values of *CV* in all cases.

5.2 Evaluation of the search space narrowing and the Lamarckian evolution

This part of the experimental study is intended to evaluate the two new elements used in the genetic algorithm with the aim of improving its effectiveness: the strategy for narrowing the search space and the Lamarckian evolution model. It has been carried out over the second set of instances presented at the beginning of the section.

Figure 2 shows the average error in percentage terms obtained by the different versions of the genetic algorithms and the A* algorithm, w.r.t. the best lower bounds computed by A*. The genetic algorithms were run 30 times for each instance and value of δ. GA and LGA denote the Baldwinian and the Lamarckian genetic algorithms respectively. In the experiments, we have considered five values for the parameter δ (0, 0.25, 0.5, 0.75, 1); which are represented in the x-axis of the graphs. The y-axis represent the errors of the algorithms, which are averaged for all the instances with the same number of jobs and machines.

Let us firstly focus on the reduction of the search space. Looking at GA in Figure 2, we can observe the expected behavior. Overall, the reductions achieved with $\delta \in \{0.5, 0.75\}$ seem to be the best options to reach a tradeoff between the size of the search space and the quality of the solutions that can be computed by GA. Note that the average solutions obtained by these versions of GA are better than those obtained by A* in all the subsets with the largest instances (Figure 2d, e, f, g) whereas the original GA ($\delta = 1$) is only able to do so for one of these subsets. At the same time, reducing the space to a great extent with $\delta \in \{0, 0.25\}$ often turns out in poor results. For the smallest instances (Figure 2a, b and c), using $\delta = 1$ is usually better than $\delta = 0.5$, and a small reduction of $\delta = 0.75$ is always the best option. This is quite reasonable as the search spaces for these instances are not too large. In the other cases, a value of $\delta = 0.5$ produces either fairly similar results as $\delta = 0.75$ or it is even better, as it happens for the largest instances with 30 jobs and 10 machines (Figure 2g). It is also clear that as the size of the instances grows, not reducing the search space at all ($\delta = 1$) yields worse results compared to those obtained by other options, being even outperformed by $\delta = 0$ (the greatest possible reduction) on the largest instances (Figure 2g).

Regarding the Lamarckian version of the genetic algorithm (LGA), we can observe that the results showed in Figure 2 do not contradict the hypotheses. Comparing both evolution models over the original search space ($\delta = 1$) LGA clearly outperforms GA in all cases, regardless of the size of the instances. In this case, LGA looks for solutions in an unconstrained and (often) large search space, so increasing the selection pressure is worthwhile. Note that the Lamarckian model itself allows LGA to outperform A* in all the subsets of largest instances (cases e, f, g, h). Combining LGA with the strategy for reducing the search space does not always produce good results, as it was expected. In the first four subsets with the smallest instances (cases a, b, c and d), LGA reaches better solutions with $\delta = 1$ than with $\delta < 1$; getting especially worse with $\delta < 0.75$. As the search space for these instances is not too large, LGA is able to reach a near-optimal solution. On the other hand, for instances with very large solution spaces, it seems that LGA is not able to explore the whole search space and so the best option is to perform a small reduction using $\delta = 0.75$. This combination reaches a good tradeoff and yields the best results in the whole experimental study for the cases e, f, g and h. It also seems that for values of $\delta < 0.75$, GA usually performs better than LGA, with the exception of the largest instances with 30 jobs and 10 machines (case g). This might be due to a premature convergence of LGA.

One important note here is that we have not observed significant differences regarding the time taken and the variation coefficient among the different versions of the genetic algorithm.

Table 5 Average ranking of the algorithms for instances with up to 100 operations

Algorithm	Ranking
LGA($\delta = 0.75$)	3.44
LGA($\delta = 1$)	3.48
GA($\delta = 0.75$)	4.44
LGA($\delta = 0.5$)	5.04
GA($\delta = 0.5$)	5.37
GA($\delta = 1$)	5.99
GA($\delta = 0.25$)	6.15
LGA($\delta = 0.25$)	6.35
GA($\delta = 0$)	7.31
LGA($\delta = 0$)	7.43

Table 6 Adjusted *p*-values. Instances with up to 100 operations

i	Hypothesis	Unadjusted *p*	p_{Holm}	p_{Shaf}
1	LGA(δ = 0) vs .LGA(δ = 0.75)	4.36E−51	1.96E−49	1.96E−49
2	LGA(δ = 0) vs .LGA(δ = 1)	4.29E−50	1.89E−48	1.54E−48
3	GA(δ = 0) vs .LGA(δ = 0.75)	3.86E−48	1.66E−46	1.39E−46
4	GA(δ = 0) vs .LGA(δ = 1)	3.54E−47	1.49E−45	1.27E−45
5	GA(δ = 0.75) vs .LGA(δ = 0)	1.46E−29	6.00E−28	5.27E−28
6	LGA(δ = 0.25) vs .LGA(δ = 0.75)	7.80E−28	3.12E−26	2.81E−26
7	GA(δ = 0) vs .GA(δ = 0.75)	2.37E−27	9.25E−26	8.54E−26
8	LGA(δ = 0.25) vs .LGA(δ = 1)	4.12E−27	1.57E−25	1.48E−25
9	GA(δ = 0.25) vs .LGA(δ = 0.75)	1.90E−24	7.03E−23	6.84E−23
10	GA(δ = 0.25) vs .LGA(δ = 1)	9.00E−24	3.24E−22	3.24E−22
11	GA(δ = 1) vs .LGA(δ = 0.75)	9.59E−22	3.36E−20	2.78E−20
12	GA(δ = 1) vs .LGA(δ = 1)	4.13E−21	1.41E−19	1.20E−19
13	LGA(δ = 0) vs .LGA(δ = 0.5)	1.94E−19	6.42E−18	5.64E−18
14	GA(δ = 0) vs .LGA(δ = 0.5)	1.13E−17	3.61E−16	3.27E−16
15	GA(δ = 0.5) vs .LGA(δ = 0)	6.57E−15	2.04E−13	1.91E−13
16	GA(δ = 0) vs .GA(δ = 0.5)	2.20E−13	6.59E−12	6.37E−12
17	GA(δ = 0.5) vs .LGA(δ = 0.75)	4.42E−13	1.28E−11	1.28E−11
18	GA(δ = 0.75) vs .LGA(δ = 0.25)	6.41E−13	1.80E−11	1.80E−11
19	GA(δ = 0.5) vs .LGA(δ = 1)	1.34E−12	3.62E−11	3.22E−11
20	GA(δ = 0.25) vs .GA(δ = 0.75)	1.05E−10	2.72E−09	2.51E−09
21	LGA(δ = 0.5) vs .LGA(δ = 0.75)	1.76E−09	4.41E−08	4.23E−08
22	LGA(δ = 0.5) vs .LGA(δ = 1)	4.46E−09	1.07E−07	1.07E−07
23	GA(δ = 0.75) vs .GA(δ = 1)	5.31E−09	1.22E−−07	1.17E−07
24	GA(δ = 1) vs .LGA(δ = 0)	4.94E−08	1.09E−06	1.09E−06
25	GA(δ = 0) vs .GA(δ = 1)	5.82E−07	1.22E−05	1.22E−05
26	LGA(δ = 0.25) vs .LGA(δ = 0.5)	8.77E−07	1.75E−05	1.75E−05
27	GA(δ = 0.25) vs .LGA(δ = 0)	1.36E−06	2.59E−05	2.45E−05
28	GA(δ = 0) vs .GA(δ = 0.25)	1.22E−05	2.19E−04	2.19E−04
29	GA(δ = 0.25) vs .LGA(δ = 0.5)	2.84E−05	4.83E−04	4.83E−04
30	LGA(δ = 0) vs .LGA(δ = 0.25)	4.15E−05	6.64E−04	6.64E−04
31	GA(δ = 0.75) vs .LGA(δ = 0.75)	1.81E−04	2.72E−03	2.72E−03
32	GA(δ = 0.5) vs .LGA(δ = 0.25)	2.21E−04	3.10E−03	3.10E−03
33	GA(δ = 0) vs .LGA(δ = 0.25)	2.70E−04	3.51E−03	3.51E−03
34	GA(δ = 0.75) vs .LGA(δ = 1)	3.28E−04	3.94E−03	3.94E−03
35	GA(δ = 1) vs .LGA(δ = 0.5)	3.67E−04	4.03E−03	4.03E−03
36	GA(δ = 0.5) vs .GA(δ = 0.75)	4.69E−04	4.69E−03	4.69E−03
37	GA(δ = 0.25) vs .GA(δ = 0.5)	3.06E−03	2.75E−02	2.75E−02
38	GA(δ = 0.5) vs .GA(δ = 1)	1.93E−02	1.55E−01	1.55E−01
39	GA(δ = 0.75) vs .LGA(δ = 0.5)	2.30E−02	1.61E−01	1.61E−01
40	GA(δ = 1) vs .LGA(δ = 0.25)	1.76E−01	1.05E+00	1.05E+00
41	GA(δ = 0.5) vs .LGA(δ = 0.5)	2.21E−01	1.10E+00	1.10E+00
42	GA(δ = 0.25) vs .LGA(δ = 0.25)	4.65E−01	1.86E+00	1.86E+00
43	GA(δ = 0.25) vs .GA(δ = 1)	5.33E−01	1.86E+00	1.86E+00
44	GA(δ = 0) vs .LGA(δ = 0)	6.48E−01	1.86E+00	1.86E+00
45	LGA(δ = 0.75) vs .LGA(δ = 1)	8.79E−01	1.86E+00	1.86E+00

5.3 Statistical analysis

The results reported so far rely on descriptive statistics, so they are only suitable for comparing the algorithms' performance over the actual instances considered in the experimental study. In order to infer conclusions for a broader class of problem instances we have made a series of statistical inference tests. Concretely, we have divided the second set of instances in two subsets: One containing instances with up to 100 operations (sizes 10×5, 15×5, 20×5 and 10×10) and another with larger instances (sizes 15×10, 20×10, 30×10 and 15×15). In a preliminary analysis, the Shapiro-Wilk test rejected the hypotheses of normality so, following (García et al. 2010), we have used non-parametric statistical techniques[1]. Concretely, for each subset we made Friedman and Iman-Davenport tests, so establishing a ranking among the algorithms and deducing whether there are significant differences among the algorithms or not. Then, as recommended in (Trawiński et al. 2012) for multiple comparisons of regression algorithms, we carried out Holm's and Shaffer's post-hoc procedures for multiple comparisons so as to establish solid pairwise comparisons between the algorithms, and determine the soundness of the ranking.

Table 5 shows the average ranking computed by the Friedman test for the subset of instances with up to 100 operations. In this case, Friedman and Iman-Davenport tests yielded p-values of 1.72E-10 and 2.22E-16 respectively, meaning there are significant differences among some of the algorithms. Table 6 reports, for each hypothesis comparing any two methods, the unadjusted *p*-value, and the adjusted *p*-values given by Holm's (*pHolm*) and Shaffer's (*pShaf*) post-hoc procedures. Roughly speaking, the hypothesis *method1 versus method2* states that there are no significant differences between the two methods. If *pHolm* <0.05 (resp. *pShaf* < 0.05) allows us to reject that hypothesis via Holm's (resp. Shaffer's) procedure, concluding that the method that appears first in the ranking (*method1* or *method2*) is more effective than the other one. As we can observe, 37 out of the 45 hypotheses (i = 1..37) are rejected by both Holm's and Shaffer's procedures. It is remarkable that LGA($\delta = 0.75$) and LGA ($\delta = 1$) outperform all the other algorithms (although nothing can be said about a comparison between both). On the other hand, GA($\delta = 0$) and LGA($\delta = 0$) are outperformed by any other configuration of the genetic algorithm. Regarding the original GA ($\delta = 1$), the only conclusion we can draw is that it outperforms GA($\delta = 0$) and LGA($\delta = 0$).

The average ranking given by the Friedman test for the subset of largest instances is reported in Table 7. For these instances, Friedman and Iman-Davenport tests returned *p*-values of 1.27E-10 and 3.33E-16 respectively, so there are differences among some algorithms. We computed Holm's and Shaffer's post-hoc procedures as well, whose adjusted p-values are shown in Table 8. Both procedures allow us to reject the same 28 hypotheses (i = 1..28). It is clear that LGA($\delta = 0.75$) is the best algorithm, as outperforms all the other methods. This algorithm is followed by LGA ($\delta = 1$), that performs better than every other method under it in the ranking with the exception of LGA($\delta = 0.5$) and LGA($\delta = 0.75$). In these two cases there are not significant statistical differences among them. Finally, it is quite remarkable that LGA($\delta = 0$), GA($\delta = 0$) and the original GA($\delta = 1$) are worse than all the other genetic algorithms. This last result regarding the base genetic algorithm strongly supports the idea that the elements introduced in GA can improve its effectiveness to a great extent.

Table 7 Average rankings of the algorithms for large instances

Algorithm	Ranking
LGA($\delta = 0.75$)	3.42
LGA($\delta = 1$)	4.44
GA($\delta = 0.75$)	4.97
LGA($\delta = 0.5$)	5.22
GA($\delta = 0.5$)	5.46
GA($\delta = 0.25$)	5.60
LGA($\delta = 0.25$)	6.02
GA($\delta = 0$)	6.53
GA($\delta = 1$)	6.55
LGA($\delta = 0$)	6.79

Summing up, the results have shown that a genetic algorithm can take profit from the proposed improvements, but we have to be aware of the importance of reaching a good tradeoff between the selection pressure and the diversity of the population. Overall, for instances up to 100 operations the best algorithm is LGA using $\delta \in \{0.75, 1\}$ whereas for larger instances LGA using $\delta = 0.75$ reaches the best solutions.

6 Conclusions

We have seen that the job-shop scheduling problem with operators proposed in (Agnetis et al. 2011) can be efficiently solved by means of a genetic algorithm. Two key points of this algorithm are the permutations with repetition encoding and the decoding algorithm designed from the *OG&T* schedule builder proposed in (Sierra et al. 2013). Also, the two enhancements proposed in this paper, namely the weak Lamarckian evolution and the narrowing of the search space, contribute greatly to improve the performance of the algorithm, as shown in the experimental evaluation.

As future work, we plan to combine the proposed genetic algorithm with local search and to apply other metaheuristics such as scatter search and path relinking to the same problem. To do that, we need to devise efficient

[1] To this aim, we have used the software developed by the Research Group on Soft Computing and Intelligent Information Systems at the University of Granada, which is available at http://sci2s.ugr.es/sicidm.

Table 8 Adjusted p-values. Large instances

i	Hypothesis	Unadjusted p	p_{Holm}	p_{Shaf}
1	LGA(δ = 0) vs .LGA(δ = 0.75)	3.17E-32	1.42E-30	1.42E-30
2	GA(δ = 1) vs .LGA(δ = 0.75)	4.53E-28	1.99E-26	1.63E-26
3	GA(δ = 0) vs .LGA(δ = 0.75)	1.17E-27	5.01E-26	4.20E-26
4	LGA(δ = 0.25) vs .LGA(δ = 0.75)	6.74E-20	2.83E-18	2.43E-18
5	LGA(δ = 0) vs .LGA(δ = 1)	1.77E-16	7.27E-15	6.38E-15
6	GA(δ = 0.25) vs .LGA(δ = 0.75)	2.36E-14	9.44E-13	8.50E-13
7	GA(δ = 1) vs .LGA(δ = 1)	1.33E-13	5.17E-12	4.77E-12
8	GA(δ = 0) vs .LGA(δ = 1)	2.52E-13	9.57E-12	9.07E-12
9	GA(δ = 0.5) vs .LGA(δ = 0.75)	9.41E-13	3.48E-11	3.39E-11
10	GA(δ = 0.75) vs .LGA(δ = 0)	1.91E-10	6.89E-09	6.89E-09
11	LGA(δ = 0.5) vs .LGA(δ = 0.75)	2.73E-10	9.54E-09	7.90E-09
12	LGA(δ = 0.25) vs .LGA(δ = 1)	2.85E-08	9.67E-07	8.25E-07
13	GA(δ = 0.75) vs .GA(δ = 1)	3.11E-08	1.03E-06	9.02E-07
14	LGA(δ = 0) vs .LGA(δ = 0.5)	3.71E-08	1.19E-06	1.08E-06
15	GA(δ = 0) vs .GA(δ = 0.75)	5.05E-08	1.57E-06	1.46E-06
16	GA(δ = 0.75) vs .LGA(δ = 0.75)	5.05E-08	1.57E-06	1.46E-06
17	GA(δ = 0.5) vs .LGA(δ = 0)	2.89E-06	8.37E-05	8.37E-05
18	GA(δ = 1) vs .LGA(δ = 0.5)	3.00E-06	8.39E-05	8.39E-05
19	GA(δ = 0) vs .LGA(δ = 0.5)	4.53E-06	1.22E-04	1.09E-04
20	GA(δ = 0.25) vs .LGA(δ = 0)	2.81E-05	7.31E-04	6.74E-04
21	GA(δ = 0.25) vs .LGA(δ = 1)	5.16E-05	1.29E-03	1.24E-03
22	GA(δ = 0.5) vs .GA(δ = 1)	1.20E-04	2.88E-03	2.88E-03
23	GA(δ = 0) vs .GA(δ = 0.5)	1.70E-04	3.91E-03	3.74E-03
24	GA(δ = 0.75) vs .LGA(δ = 0.25)	2.31E-04	5.09E-03	5.09E-03
25	LGA(δ = 0.75) vs .LGA(δ = 1)	3.42E-04	7.19E-03	7.19E-03
26	GA(δ = 0.5) vs .LGA(δ = 1)	3.74E-04	7.48E-03	7.48E-03
27	GA(δ = 0.25) vs .GA(δ = 1)	7.93E-04	1.51E-02	1.43E-02
28	GA(δ = 0) vs .GA(δ = 0.25)	1.08E-03	1.94E-02	1.94E-02
29	LGA(δ = 0.25) vs .LGA(δ = 0.5)	4.83E-03	8.21E-02	8.21E-02
30	LGA(δ = 0.5) vs .LGA(δ = 1)	6.28E-03	1.01E-01	1.01E-01
31	LGA(δ = 0) vs .LGA(δ = 0.25)	7.24E-03	1.09E-01	1.09E-01
32	GA(δ = 0.25) vs .GA(δ = 0.75)	2.93E-02	4.10E-01	4.10E-01
33	GA(δ = 0.5) vs .LGA(δ = 0.25)	4.63E-02	6.01E-01	6.01E-01
34	GA(δ = 0.75) vs .LGA(δ = 1)	6.17E-02	7.40E-01	7.40E-01
35	GA(δ = 1) vs .LGA(δ = 0.25)	6.39E-02	7.40E-01	7.40E-01
36	GA(δ = 0) vs .LGA(δ = 0.25)	7.72E-02	7.72E-01	7.72E-01
37	GA(δ = 0.5) vs .GA(δ = 0.75)	9.12E-02	8.20E-01	8.20E-01
38	GA(δ = 0.25) vs .LGA(δ = 0.25)	1.33E-01	1.06E+00	1.06E+00
39	GA(δ = 0.25) vs .LGA(δ = 0.5)	1.88E-01	1.32E+00	1.32E+00
40	GA(δ = 0) vs .LGA(δ = 0)	3.58E-01	2.15E+00	2.15E+00
41	GA(δ = 0.75) vs .LGA(δ = 0.5)	3.88E-01	2.15E+00	2.15E+00
42	GA(δ = 1) vs .LGA(δ = 0)	4.05E-01	2.15E+00	2.15E+00
43	GA(δ = 0.5) vs .LGA(δ = 0.5)	4.09E-01	2.15E+00	2.15E+00
44	GA(δ = 0.25) vs .GA(δ = 0.5)	6.24E-01	2.15E+00	2.15E+00
45	GA(δ = 0) vs .GA(δ = 1)	9.32E-01	2.15E+00	2.15E+00

neighborhood structures. As it was done, for example, in (González et al. 2008) for the job-shop scheduling problem with sequence-dependent setup times or in (González Rodríguez et al. 2008) for the job-shop scheduling with uncertain durations, we will look for inspiration in the structures proposed in (Van Laarhoven et al. 1992, Dell' Amico and Trubian 1993) for the classical job-shop scheduling problem.

Acknowledgments This work has been supported by the Spanish Ministry of Science and Innovation under research project MICINN-FEDER TIN2010-20976-C02-02 and by FICYT under Grant BP09105.

References

Agnetis A, Flamini M, Nicosia G, Pacifici A (2011) A job-shop problem with one additional resource type. J Sched 14(3):225–237

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. Ann Oper Res 138:21–52

Beasley JE (1990) Or-library: distributing test problems by electronic mail. J Oper Res Soc 41(11):1069–1072, http://www.jstor.org/stable/2582903

Bierwirth C (1995) A generalized permutation approach to jobshop scheduling with genetic algorithms. OR Spectrum 17:87–92

Bierwirth C, Mattfeld DC (1999) Production scheduling and rescheduling with genetic algoritms. Evol Comput 7:1–17

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

Dell' Amico M., Trubian M. (1993) Applying tabu search to the job-shop scheduling problem. Ann Oper Res 41:231–252

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 180:2044–2064

Giffler B, Thompson GL (1960) Algorithms for solving production scheduling problems. Oper Res 8:487–503

González MA, Vela CR, Varela R (2008) A new hybrid genetic algorithm for the job shop scheduling problem with setup times. In: Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling (ICAPS-2008). AAAI Press, Sidney

González MA, Vela CR, Varela R (2012) A competent memetic algorithm for complex scheduling. Nat Comput 11:151–160

González Rodríguez I, Vela CR, Puente J, Varela R (2008) A new local search for the job shop problem with uncertain durations. In: Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling (ICAPS-2008). AAAI Press, Sidney

Mattfeld DC (1995) Evolutionary search and the job shop investigations on genetic algorithms for production scheduling. Springer, Berlin

Mencía C, Sierra MR, Varela R (2012) Depth-first heuristic search for the job shop scheduling problem. Ann Oper Res. doi:10.1007/s10479-012-1296-x

Mencía C, Sierra MR, Varela R (2013) Intensified iterative deepening A* with application to job shop scheduling. J Intell Manuf. doi:10.1007/s10845-012-0726-6

Mencía R, Sierra M, Mencía C, Varela R (2011) Genetic algorithm for job-shop scheduling with operators. Lect Notes Comput Sci 6687(2):305–314

Sierra MR, Mencía C, Varela R (2013) Searching for optimal schedules to the job-shop problem with operators. Technical report. iScOp Research Group. University of Oviedo, Oviedo

Sierra MR, Varela R (2010) Pruning by dominance in best-first search for the job shop scheduling problem with total flow time. J Intell Manuf 21(1):111–119

Trawiński B, Smetek M, Telec Z, Lasota T (2012) Nonparametric statistical analysis for multiple comparison of machine learning regression algorithms. Int J Appl Math Comput Sci 22(4):867–881

Van Laarhoven P, Aarts E, Lenstra K (1992) Job shop scheduling by simulated annealing. Oper Res 40:113–125

Varela R, Serrano D, Sierra M (2005) New codification schemas for scheduling with genetic algorithms. Proceedings of IWINAC 2005. Lect Notes Comput Sci 3562:11–20