Available online at www.sciencedirect.com

ScienceDirect

Computers & Operations Research 35 (2008) 3202–3212

computers & operations research

www.elsevier.com/locate/cor

A genetic algorithm for the Flexible Job-shop Scheduling Problem

F. Pezzella[a,*], G. Morganti[a], G. Ciaschetti[b]

[a]*Dipartimento di Ingegneria Informatica, Gestionale e dell'Automazione, Università Politecnica delle Marche, via Brecce Bianche, 60131 Ancona, Italy*
[b]*Università Politecnica delle Marche, Italy*

Available online 1 March 2007

Abstract

In this paper, we present a genetic algorithm for the Flexible Job-shop Scheduling Problem (FJSP). The algorithm integrates different strategies for generating the initial population, selecting the individuals for reproduction and reproducing new individuals. Computational result shows that the integration of more strategies in a genetic framework leads to better results, with respect to other genetic algorithms. Moreover, results are quite comparable to those obtained by the best-known algorithm, based on tabu search. These two results, together with the flexibility of genetic paradigm, prove that genetic algorithms are effective for solving FJSP.
© 2007 Elsevier Ltd. All rights reserved.

Keywords: Job-shop Scheduling; Genetic algorithms; Flexible manufacturing systems

Scheduling of operations is one of the most critical issues in the planning and managing of manufacturing processes. To find the best schedule can be very easy or very difficult, depending on the shop environment, the process constraints and the performance indicator [1]. One of the most difficult problems in this area is the Job-shop Scheduling Problem (JSP), where a set of jobs must be processed on a set of machines, each job is formed by a sequence of consecutive operations, each operation requires exactly one machine, machines are continuously available and can process one operation at a time without interruption. The decision concerns how to sequence the operations on the machines, such as a given performance indicator is optimized. A typical performance indicator for JSP is the makespan, i.e., the time needed to complete all the jobs. JSP is a well-known NP-hard problem [2].

The Flexible Job-shop Scheduling problem (FJSP) is a generalization of the classical JSP, where operations are allowed to be processed on *any* among a set of available machines. Then, FJSP is more difficult than the classical JSP, since it introduces a further decision level beside the sequencing one, i.e., the job routes. To determine the job routes means to decide, for each operation, what machine must process it, among the available ones.

The difficulty of FJSP suggests the adoption of heuristic methods producing reasonably good schedules in a reasonable time, instead of looking for an exact solution, also for small instances. Heuristics generally do not give solutions whose objective function value have a guaranteed distance from the optimum, but they can be effective for most problem instances. In recent years, the adoption of meta-heuristics such as simulated annealing, tabu search and genetic algorithms (GAS) has led to better results than classical dispatching or greedy heuristic algorithms [3–5].

* Corresponding author. Tel.: +39 071 2204826.
E-mail addresses: pezzella@diiga.univpm.it (F. Pezzella), morganti@diiga.univpm.it (G. Morganti), ciaschetti@diiga.univpm.it (G. Ciaschetti).

0305-0548/$ - see front matter © 2007 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cor.2007.02.014

In this paper we present a new GA for FJSP, which improves some strategies already known in literature, and mixes them to find the best criteria at each algorithm step. In particular, we adopt the *approach by localization* of Kacem et al. [6] to find initial assignments, improving it by reordering the jobs and the machines, and by searching for the global minimum in the instance table. The Most Work Remaining (MWR), the Most Operation Remaining (MOR) and random selection of the next job dispatching rules are then adopted to sequence the operations, generating the initial population. The selection criterion is chosen among binary tournament, n-size tournament and linear ranking. Different crossover and mutation genetic operators are adopted, both for the assignment and for the sequencing. In particular, we introduce in this context an *intelligent mutation* assignment operator. Computation shows that our algorithm outperforms the best-known GAs for FJSP, and that solution quality is comparable with the best-known algorithm for FJSP, i.e., the tabu search of Mastrolilli and Gambardella [3]. Our algorithm proves that when a mix of different rules for generating the initial population, selection of individuals and reproduction operators is adopted, the GA framework is suitable to develop efficient algorithms for FJSP.

The paper is organized as follows. In Section 1 we introduce the problem formulation and show an illustrative example of input data. In Section 2 we give relevant literature review on the subject. In Section 3 we present the algorithm, by detailing the strategies to generate initial solutions, the coding scheme, the fitness evaluation function, the selection criteria and the GA operators adopted to generate the offsprings. In Section 4 we present an extensive computational study on 178 benchmark problems, comparing our results with the best-known approaches. Some final remarks and future research directions are given in Section 5.

1. The FJSP

The FJSP can be stated as follows. It is given a set $J = \{J_1, J_2, \ldots, J_n\}$ of independent jobs. A job J_i is formed by a sequence $O_{i_1}, O_{i_2}, \ldots, O_{i_{n_i}}$ of operations to be performed one after the other according to the given sequence. It is given a set $U = \{M_1, M_2, \ldots, M_m\}$ of machines. Each operation O_{ij} can be executed on any among a subset $U_{ij} \subseteq U$ of compatible machines. We have partial flexibility if there exists a proper subset U_{ij} of U, for at least one operation O_{ij}, while we have $U_{ij} = U$ for each operation O_{ij} in the case of total flexibility. The processing time of each operation is machine-dependent. We denote with d_{ijk} the processing time of operation O_{ij} when executed on machine M_k. Pre-emption is not allowed, i.e., each operation must be completed without interruption once started. Furthermore, machines cannot perform more than one operation at a time. All jobs and machines are available at time 0.

The problem is to assign each operation to an appropriate machine (routing problem), and to sequence the operations on the machines (sequencing problem) in order to minimize the makespan, i.e., the time needed to complete all the jobs, which is defined as $MK = \max_i\{C_i\}$, where C_i is the completion time of job J_i. According to the evolution paradigm used in GAs, we will refer to any solution of FJSP as an *individual* or *chromosome*.

Problem data can be organized in a table, where rows correspond to operations and columns correspond to machines. The entries of the input table are the processing times, as in the example given in Table 1. In this example, we have total flexibility. In a partial flexibility scenario, a ∞ entry in the table means that a machine cannot execute the corresponding operation, i.e., it does not belong to the subset of compatible machines for that operation.

Table 1
Processing time table

	M_1	M_2	M_3	M_4
O_{11}	7	6	4	5
O_{12}	4	8	5	6
O_{13}	9	5	4	7
O_{21}	2	5	1	3
O_{22}	4	6	8	4
O_{23}	9	7	2	2
O_{31}	8	6	3	5
O_{32}	3	5	8	3

2. Literature review

To start our literature review, we have to recall that exact algorithms are not effective for solving FJSP also for small instances. In fact, exact methods based on a disjunctive graph representation of the problem have been developed, but they are not applicable for instances with more than 20 jobs and 10 machines [1]. Our review will then focus only on heuristic approaches. First, we have to claim that, to our knowledge, no approximation algorithms are known for FJSP producing solutions with a guaranteed distance from the optimal solution. Some known results on the approximation of JSP [7] are not easily expandable to FJSP. Only when the number of machines and the maximum number of operations per job are fixed, Jansen et al. [8] give a linear time approximation scheme.

Several heuristic procedures such as dispatching rules, local search and meta-heuristics such as tabu search, simulated annealing and GAs have been developed in recent years for FJSP. They can be classified into two main categories: hierarchical approach and integrated approach. The hierarchical approach attempts to solve the problem by decomposing it into a sequence of subproblems, with reduced difficulty. A typical decomposition is *assign-then-sequence*, coming from the trivial observation that once the assignment is done, the resulting sequencing problem is a JSP. This approach is followed by Brandimarte [9], Paulli [10], Chambers and Barnes [4], among the others. They all solve the assignment problem using some dispatching rules, and then solve the resulting JSP using different tabu search heuristics.

Integrated approach is much more difficult to solve, but in general achieves better results, as reported in Vaessens et al. [11], Dauzére-Pérés and Paulli [12], Hurink et al. [13] and Mastrolilli and Gambardella [3]. They all adopt an integrated approach, proposing different tabu search to solve the problem. Among them, Mastrolilli and Gambardella show computational results proving that their tabu search performs better than any other heuristic developed so far, both in terms of computation time and solution quality.

Recently, GAs have been successfully adopted to solve FJSP, as proven by the growing number of papers on the topic. The most relevant works are those of Chen et al. [5], Jia et al. [14], Ho and Tay [15] and, in particular, Kacem et al. [6] that give the basis of our work. They are all integrated approaches, and differ from each other for different coding scheme, initial population generation, chromosome selection and offspring generation strategies. Chen et al. [5] split the chromosome representation into two parts, the first defining the routing policy, and the second the sequence of operations on each machine. Jia et al. [14] propose a modified GA able to solve distributed scheduling problems and can be adapted for FJSP. Ho and Tay [15] propose an efficient methodology called GENACE based on a cultural evolutionary architecture for solving FJSP with recirculation. Finally, Kacem et al. [6] use a chromosome representation that combines both routing and sequencing information, and develop an *approach by localization* to find promising initial assignments. Dispatching rules are then applied to sequence the operations. Once this initial population is found, they apply crossover and mutation operators to jointly modify assignments and sequences, producing better individual as the generations go by. We remark that in our work we adopt many of the choices of Kacem et al. [6] giving some improvements on their method. Furthermore, they conduct experimental results only on some sample instances, and we complete their work in this paper with our extensive computational study.

Concluding our literature review, we want to mention the important work of Tay and Wibowo [16] who compare four different chromosome representations, testing their performance on some 10×10 (10 jobs, 10 machines) problem instances. Their result shows that the more the chromosome representation is formed by concatenating meaningful partial strings, the more appropriate genetic operators can be developed, and better results can be obtained. They also report some remarks on the best rates of crossover operators and mutation operators to generate the offspring.

3. A GA for FJSP

GA is a local search algorithm that follows the evolution paradigm. Starting from an initial population, the algorithm applies genetic operators in order to produce offsprings (in the local search terminology, it corresponds to exploring the neighborhood), which are presumably more fit than their ancestors. At each generation (iteration), every new individual (chromosome) corresponds to a solution, i.e., a schedule of the given FJSP instance. The strength of GA with respect to other local search algorithms is due to the fact that in a GA framework more strategies can be adopted together to find individuals to add to the mating pool, both in the initial population phase and in the dynamic generation phase. Then, a more variable search space can be explored at each algorithm step. The overall structure of our GA can be

Table 2
Approach by localization (machine workload updates in bold)

	M_1	M_2	M_3	M_4	M_1	M_2	M_3	M_4	M_1	M_2	M_3	M_4	
O_{11}	7	6	4	5	7	6	$\boxed{4}$	5	7	6	$\boxed{4}$	5	
O_{12}	4	8	5	6	4	8	**9**	6	$\boxed{4}$	8	9	6	
O_{13}	9	5	4	7	9	5	**8**	7	**13**	5	8	7	
O_{21}	2	5	1	3	2	5	**5**	3	**6**	5	5	3	
O_{22}	4	6	8	4	4	6	**12**	4	**8**	6	12	4	. . .
O_{23}	9	7	2	2	9	7	**6**	2	**13**	7	6	2	
O_{31}	8	6	3	5	8	6	**7**	5	**12**	6	7	5	
O_{32}	3	5	8	3	3	5	**12**	3	**7**	5	12	3	

described as follows:

1. *Coding*: The genes of the chromosomes describe the assignment of operations to the machines, and the order in which they appear in the chromosome describes the sequence of operations. Each chromosome represents a solution for the problem.
2. *Initial population*: The initial chromosomes are obtained by a mix of two assignment procedures (global minimum and random permutation of jobs and machines) and a mix of three dispatching rules (Random, MWR, MOR) for sequencing.
3. *Fitness evaluation*: The makespan is computed for each chromosome in the current generation.
4. *Selection*: At each iteration, the best chromosomes are chosen for reproduction by one among three different methods, i.e., binary tournament, n-size tournament and linear ranking.
5. *Offspring generation*: The new generation is obtained by changing the assignment of the operations to the machines (assignment crossover, assignment mutation, intelligent mutation) and by changing the sequencing of operations (POX crossover and PPS mutation). These rules preserve feasibility of new individuals. New individuals are generated until a fixed maximum number of individuals is reached. In our approach, only the new individuals form the mating pool for the next generation, at each algorithm step.
6. *Stop criterion*: Fixed number of generations is reached. If the stop criterion is satisfied, the algorithm ends and the best chromosome, together with the corresponding schedule, is given as output. Otherwise, the algorithm iterates again steps 3–5.

To describe our GA, we detail in the following the different steps of the algorithm. In particular, we describe in Section 3.1 how we generate the initial population; in Section 3.2 we present the coding scheme adopted; in Section 3.3 we describe the fitness evaluation function; in Section 3.4 we report the strategies to select individuals for reproduction and, finally, in Section 3.5 we describe the mutation and crossover operators, both for changing the assignment and the sequencing of operations.

3.1. Initial population

The initial population is generated following the *approach by localization* of Kacem et al. [6]. This approach takes into account both the processing times and the workload of the machines, i.e., the sum of the processing times of the operations assigned to each machine. The procedure consists in finding, for each operation, the machine with the minimum processing time, fixing that assignment, and then to add this time to every subsequent entry in the same column (machine workload update), as shown in Table 2, where bold values correspond to workload updates.

Since this approach is strongly dependent on the order in which operations and machines are given in the table, we slightly modify it in two ways:

- AssignmentRule1: search for the global minimum in the processing time table.
- AssignmentRule2: randomly permute jobs and machines in the table.

Table 3
Initial assignments by AssignmentRule1 (machine workload updates in bold)

	M_1	M_2	M_3	M_4	M_1	M_2	M_3	M_4	M_1	M_2	M_3	M_4		M_1	M_2	M_3	M_4
O_{11}	7	6	4	5	7	6	**5**	5	7	6	5	**7**		7	6	$\boxed{4}$	5
O_{12}	4	8	5	6	4	8	**6**	6	4	8	6	**8**		$\boxed{4}$	8	5	6
O_{13}	9	5	4	7	9	5	**5**	7	9	5	5	**9**		9	$\boxed{5}$	4	7
O_{21}	2	5	1	3	2	5	$\boxed{1}$	3	2	5	1	**5**		2	5	$\boxed{1}$	3
O_{22}	4	6	8	4	4	6	**9**	4	4	6	9	**6**	…	4	6	8	$\boxed{4}$
O_{23}	9	7	2	2	9	7	**3**	2	9	7	3	$\boxed{2}$		9	7	2	$\boxed{2}$
O_{31}	8	6	3	5	8	6	**4**	5	8	6	4	**7**		8	6	$\boxed{3}$	5
O_{32}	3	5	8	3	3	5	**9**	3	3	5	9	**5**		$\boxed{3}$	5	8	3

Table 4
Initial assignments by AssignmentRule2 (machine workload updates in bold)

	M_4	M_1	M_3	M_2	M_4	M_1	M_3	M_2	M_4	M_1	M_3	M_2		M_4	M_1	M_3	M_2
O_{31}	5	8	3	6	5	8	$\boxed{3}$	6	5	8	3	6		5	8	$\boxed{3}$	6
O_{32}	3	3	8	5	3	3	**11**	5	$\boxed{3}$	3	11	5		$\boxed{3}$	3	8	5
O_{11}	5	7	4	6	5	7	**7**	6	**8**	7	7	6		5	7	4	$\boxed{6}$
O_{12}	6	4	5	8	6	4	**8**	8	**9**	4	8	8		6	$\boxed{4}$	5	8
O_{13}	7	9	4	5	7	9	**7**	5	**10**	9	7	5	…	7	9	$\boxed{4}$	5
O_{21}	3	2	1	5	3	2	**4**	5	**6**	2	4	5		$\boxed{3}$	2	1	5
O_{22}	4	4	8	6	4	4	**11**	6	**7**	4	11	6		4	$\boxed{4}$	8	6
O_{23}	2	9	2	7	2	9	**5**	7	**5**	9	5	7		$\boxed{2}$	9	2	7

AssignmentRule1 foresees to start from the operation that corresponds to the global minimum in the table. As a consequence, the machine workload update is performed on every other operation, i.e., every other entry in the same column. AssignmentRule2 foresees to permute randomly the jobs and the machines before to apply the approach by localization. The advantage in using AssignmentRule2 is that it finds different initial assignments in different runs of the algorithm, better exploring the search space. The adoption of a mix of these two rules produces the initial set of assignments. For example, 10% of initial population can be generated by rule 1 and 90% by rule 2. One of the goals of our research is, in fact, to find a robust tuning of this mix for different classes of problem instances. Examples showing how AssignmentRule1 and AssignmentRule2 work are illustrated in Tables 3 and 4, respectively (bold values report machine workload updates). In both examples, the last table shown represents the final assignments obtained.

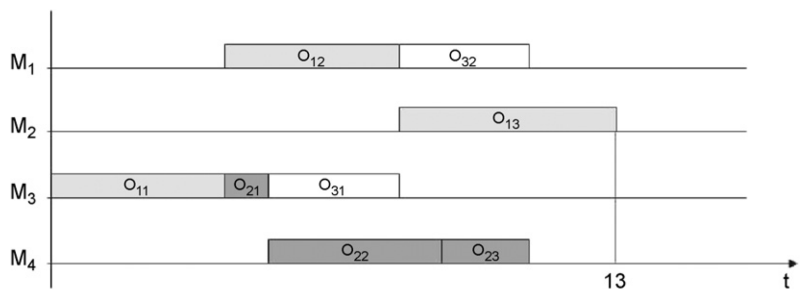

Fig. 1. Gantt chart.

Once the assignments are settled up, we have to determine how to sequence the operations on the machines. Obviously, the sequencing is feasible if it respects the precedence constraints among operations of the same job, i.e., operation $O_{i,j+1}$ cannot be processed before operation $O_{i,j}$.

In our GA, the sequencing of the initial assignments is obtained by a mix of three known dispatching rules:

(a) Randomly select a job (Random);
(b) Most Work Remaining (MWR);
(c) Most number of Operations Remaining (MOR).

For example, 20% of initial chromosomes can be generated by rule (a), 40% by rule (b) and 40% by rule (c). Again, finding a robust tuning for the mix of these three rules for different classes of problem instances is one of the goals of our research. As an example, by applying the MWR rule to the assignments of Table 3, we obtain the following initial solution, which has makespan equal to 13, and is drawn using a Gantt chart in Fig. 1.

$$S = (O_{11}, M_3), (O_{12}, M_1), (O_{21}, M_3), (O_{22}, M_4), (O_{31}, M_3), (O_{13}, M_2), (O_{32}, M_1), (O_{23}, M_4).$$

3.2. Coding

In order to implement our GA, we need to represent a schedule symbolically, e.g., by a string. We use the *task sequencing list* representation proposed by Kacem et al. [6], in which a string is formed by triples (i, j, k), one for each operation, where

- i is the job that operation belongs to;
- j is the progressive number of that operation within job i;
- k is the machine assigned to that operation.

The length of the string is equal to the total number of operations. In the example shown in Fig. 1, the solution

$$S = (O_{11}, M_3), (O_{12}, M_1), (O_{21}, M_3), (O_{22}, M_4), (O_{31}, M_3), (O_{13}, M_2), (O_{32}, M_1), (O_{23}, M_4)$$

is represented by the string

(1,1,3)	(1,2,1)	(2,1,3)	(2,2,4)	(3,1,3)	(1,3,2)	(3,2,1)	(2,3,4)

3.3. Fitness evaluation

The fitness evaluation function for the chromosomes coincides with the makespan of the solution they represent. Therefore, since we are searching for solutions with lower values of the makespan, the genetic evolution will prefer chromosomes with a lower fitness. For each generation, all the chromosomes are evaluated, and the best individual is recorded.

3.4. Selection

The selection phase is in charge to choose the chromosomes for reproduction. In our approach, the criterion used to select the chromosomes to be included in the mating pool can be chosen among three selection methods well known in the GA literature: binary tournament, *n*-size tournament and linear ranking.

- *Binary tournament*: Two individuals are randomly chosen from the population and the best of them is selected for reproduction.
- *n-Size tournament*: The individual for reproduction is chosen among a random number of individuals.
- *Linear ranking*: Individuals are sorted according to their fitness and a rank $r_i \in \{1, \ldots, N\}$ is assigned to each, where N is the population size. The best individual gets rank N while the worst gets rank 1. Then,

$$p_i = \frac{2r_i}{N(N+1)}, \quad i = 1, \ldots, N$$

  is the probability of choosing the *i*th individual in the rank ordering.

From the mating pool, pairs of chromosomes are then randomly selected for reproduction. In our approach, the mating pool is completely renewed at each iteration, then the chosen criterion has to be repeated until the number of individuals in the mating pool equals the population size. Computational experience proves that binary tournament gives better results among the three different methods.

3.5. Offspring generation

Once the chromosomes for reproduction have been selected, the *crossover* and *mutation* genetic operators are applied to produce the offspring. Crossover operator applies to pairs of chromosomes, while mutation operator applies to single individuals. We distinguish between two kinds of operators:

- assignment operators;
- sequencing operators.

Assignment operators only change the assignment property of the chromosomes, i.e., the sequencing of operations is preserved in the offspring. Assignment crossover generates the offspring by exchanging the assignment of a subset of operations between the two parents. Assignment mutation only exchanges the assignment of a single operation in a single parent. In both cases the operations to be exchanged are chosen arbitrarily. In the *intelligent mutation*, we select an operation on the machine with the maximum workload, and assign it to the machine with the minimum workload, if compatible.

Sequencing operators only change the sequence of the operations in the parent chromosomes, i.e., the assignment of operations to machines is preserved in the offspring. In applying the sequencing operators, we must respect the precedence constraints among operations of the same job. Adopting a correcting algorithm to modify unfeasible offspring is very time-consuming, and hence it is preferable to design operators such that precedence constraints are not violated. As in Kacem et al. [6], the Precedence Preserving Order-based crossover (POX) and Precedence Preserving Shift mutation (PPS) operators of Lee et al. [17] are adopted.

POX generates two children starting from two parents. First, POX selects an operation from the first parent, copies in the first child all the operations of the job which the selected operations belong to, then complete this new individual with the remaining operations, in the same order as they appear in the second parent. The symmetric process is repeated for the second parent and the second child. Note that POX preserves the sequencing constraints. PPS selects an operation from a single parent chromosome and moves it into another position, taking care of the precedence constraints for that operation. PPS operator is applied only if the mutation improves the solution quality.

The offspring generation phase terminates when the maximum number of individuals in the mating pool is reached, i.e., a new generation is found. The algorithm ends when a maximal number of generations is reached, and the best individual, together with the corresponding schedule, is given as output.

4. Computational result

The GA procedure described in Section 3 has been implemented on a 1.8 MHz Pentium IV processor, and tested on a large amount of problem instances from literature (http://www.idsia.ch/monaldo). The best results are selected after five runs from different initial populations. Several sets of problem instances have been considered.

1. *Brandimarte* [12]: The data set consists of 10 problems with number of jobs ranging from 10 to 20, number of machines ranging from 4 to 15 and number of operations for each job ranging from 5 to 15.
2. *Dauzére-Pérés and Paulli* [12]: The data set consists of 18 test problems with number of jobs ranging from 10 to 20, number of machines ranging from 5 to 10 and number of operations for each job ranging from 5 to 25.
3. *Barnes and Chambers* [4]: The data set consists of 21 problems obtained from the classical *mt10* JSP instance of Fisher and Thompson [18] and the *la24*, *la40* instances proposed by Lawrence [19]. The number of jobs ranges from 10 to 15 and the number of machines ranges from 11 to 18.
4. *Hurink et al.* [13]: The data set consists of 129 test problems created from 43 classical JSP instances. They divide the test problems into three subsets, *EData*, *RData* and *VData*, depending on the average number of alternative machines for each operation. The number of jobs ranges from 6 to 30 and the number of machines ranges from 5 to 15.

In our experiment, we tested different values for a list of algorithm parameters, and computational experience proves that binary tournament selection and the following values are more effective:

- population size: 5000;
- number of generations: 1000;
- rate of initial assignments with global minimum method: 10%;
- rate of initial assignments with random permutation method: 90%;
- rate of initial sequences with Random rule: 20%;
- rate of initial sequences with MWR rule: 40%;
- rate of initial sequences with MOR rule: 40%;
- POX crossover probability: 45%;
- assignment crossover probability: 45%;
- PPS mutation probability: 2%;
- assignment mutation probability: 2%;
- assignment intelligent mutation probability: 6%.

Table 5 compares our GA to the algorithms proposed by Chen et al. [5], Ho and Tay [15] and Jia et al. [14] on 10 FJSP problem instances from Brandimarte [9]. The first column reports the instance name; the second and third columns report the number of jobs and the number of machines for each instance, respectively. The fourth column reports the best-known lower bound [3]. The fifth column reports our best makespan over five runs of GA. The remaining columns report the best results of the three algorithms we compare with, together with the relative deviation with respect to our algorithm. The relative deviation is defined as

$$dev = [(MK_{\text{comp}} - MK_{\text{GA}})/MK_{\text{comp}}] \times 100\%,$$

where MK_{GA} is the makespan obtained by our algorithm and MK_{comp} is the makespan of the algorithm we compare to. Result shows that our algorithm outperforms the other three GAs. In the table, the results reported in the 6th and 10th columns are obtained by our implementation of the algorithms of Chen et al. [5], and by adapting the algorithm of Jia et al. [14] to FJSP.

In Table 6 we compare our best results over five runs of GA, to the best results of the tabu search of Mastrolilli and Gambardella [3], on 10 FJSP instances from Brandimarte. The seventh column reports the relative deviation of our algorithm with respect to their. The values of LB within parenthesis are optimal. Result shows that the quality of solutions are comparable.

Table 7 shows computational results over the four instance classes. The first column reports the data set, the second column the number of instances for each class, the third column the average number of alternative machines per

Table 5
Comparison with other GAs on 10 FJSP instances from Brandimarte

Name	n	M	LB	GA	Chen	*dev* (%)	GENACE	*dev* (%)	Jia	*dev* (%)
Mk01	10	6	36	40	40	0	41	+2.44	40	0
Mk02	10	6	24	26	29	+10.34	29	+10.34	28	7.14
Mk03	15	8	204	204	204	0	204	0	204	0
Mk04	15	8	48	60	63	+4.76	67	+10.45	61	1.64
Mk05	15	4	168	173	181	+4.42	176	+1.70	176	1.70
Mk06	10	15	33	63	60	−5.0	68	+7.35	62	−1.61
Mk07	20	5	133	139	148	+6.08	148	+3.38	145	4.14
Mk08	20	10	523	523	523	0	523	0	523	0
Mk09	20	10	299	311	308	−0.97	328	+5.18	310	−0.32
Mk10	20	15	165	212	212	0	231	+8.23	216	1.85
Average improvement						+1.96		+5.18		+1.45

Table 6
Comparison with the tabu search of Mastrolilli and Gambardella on 10 FJSP instances from Brandimarte

Name	n	M	LB	GA	M.G.	*dev* (%)
Mk01	10	6	36	40	40	0
Mk02	10	6	24	26	26	0
Mk03	15	8	(204)	204	204	0
Mk04	15	8	48	60	60	0
Mk05	15	4	168	173	173	0
Mk06	10	15	33	63	58	−8.62
Mk07	20	5	133	139	144	+3.47
Mk08	20	10	(523)	523	523	0
Mk09	20	10	299	311	307	−0.64
Mk10	20	15	165	212	198	−7.07

Table 7
Mean relative error (MRE) over best-known lower bound

Data set	Num.	Alt.	GA (%)	Chen (%)	Jia (%)
Brandimarte	10	2.59	17.53	19.55	19.11
Dauzére-Pérés and Paulli	18	2.49	7.63	7.91	10.62
Barnes and Chambers	21	1.18	29.56	38.64	29.75
Hurink EData	43	1.15	6.00	5.59	9.01
Hurink RData	43	2	4.42	4.41	8.34
Hurink VData	43	4.31	2.04	2.59	3.24

operation. The next three columns report the mean relative error (MRE) of the best solution obtained by our GA, by Chen et al. [5] and by Jia et al. [14], respectively, with respect to the best-known lower bound. The relative error (RE) is defined as $RE = [(MK - LB)/LB \times 100]\%$ where MK is the best makespan obtained by the reported algorithm and LB is the best-known lower bound. The table shows that our algorithm is stronger with a higher degree of flexibility (Hurink VData). Furthermore, result shows that our algorithm outperforms the other two genetic algorithms.

In Fig. 2 we draw the decrease of the average makespan and the best makespan over five runs, for the la01 test problem class with 10 jobs and 5 machines [19]. It is possible to note that our algorithm improves the average makespan very fast, and that the best makespan, equal to 609, is reached after 35 generations and is optimal.

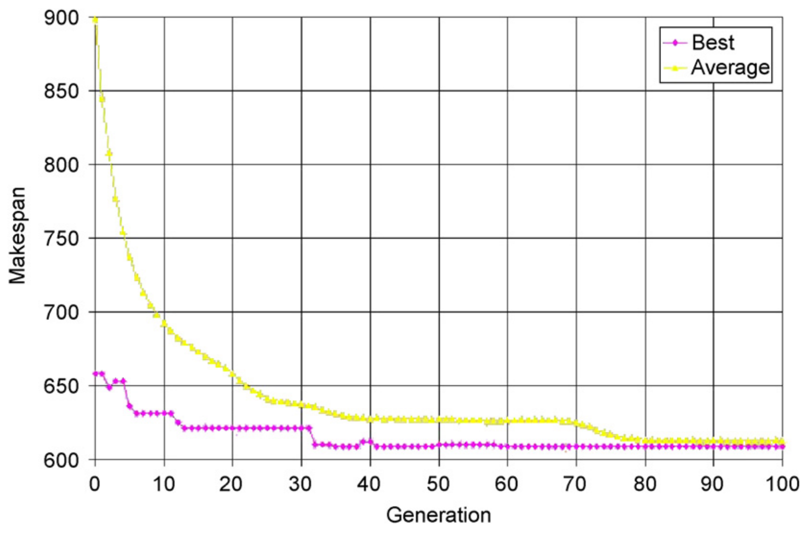

Fig. 2. Decreasing of the makespan.

5. Conclusion and future research

In this work we develop a genetic algorithm (GA) for the Flexible Job-shop Scheduling Problem (FJSP). An extensive computational study shows that our algorithm outperforms other known GA for the same problem, and gives results comparable with the best algorithm known so far. As a consequence, the GA framework is effective for developing efficient algorithms for FJSP, when integration of different strategies for selection and reproduction, as well as procedures for finding an initial population, are adopted. In the future, it will be interesting to investigate on the following issues:

- use of a mix of different selection criteria for choosing the best individual of a given generation of chromosomes;
- maintain the best individuals of the mating pool in a successive generation of chromosomes;
- improve the intelligent mutation operator by selecting a critical operation, instead of any operation, on the most critical machine;
- consider other performance indicators and process constraints.

References

[1] Pinedo M. Scheduling: theory, algorithms and systems. Englewood cliffs, NJ: Prentice-Hall; 2002.

[2] Garey MR, Johnson DS, Sethi R. The complexity of flowshop and jobshop scheduling. Mathematics of Operations Research 1976;1:117–29.

[3] Mastrolilli M, Gambardella LM. Effective neighbourhood functions for the flexible job shop problem. Journal of Scheduling 1996;3:3–20.

[4] Barnes JW, Chambers JB. Flexible Job Shop Scheduling by tabu search. Graduate program in operations research and industrial engineering. Technical Report ORP 9609, University of Texas, Austin; 1996. ⟨http://www.cs.utexas.edu/users/jbc/⟩.

[5] Chen H, Ihlow J, Lehmann C. A genetic algorithm for flexible Job-shop scheduling. In: IEEE international conference on robotics and automation, Detroit; 1999. p. 1120–5.

[6] Kacem I, Hammadi S, Borne P. Approach by localization and multiobjective evolutionary optimization for flexible job-shop scheduling problems. IEEE Transactions on Systems, Man, and Cybernetics, Part C 2002;32(1):1–13.

[7] Feige U., Scheideler C. Improved bounds for acyclic job shop scheduling. In: Proceedings of the 30th annual ACM symposium on the theory of computing (STOC '98); 1998. p. 624–33.

[8] Jansen K, Mastrolilli M, Solis-Oba R. Approximation algorithms for Flexible Job Shop Problems. In: Lecture notes in computer science, vol. 1776. Proceedings of the fourth Latin American symposium on theoretical informatics; Berlin: Springer. 2000. p. 68–77.

[9] Brandimarte P. Routing and scheduling in a flexible job shop by tabu search. Annals of Operations Research 1993;41:157–83.

[10] Paulli J. A hierarchical approach for the FMS scheduling problem. European Journal of Operational Research 1995;86(1):32–42.

[11] Vaessens RJM, Aarts EHL, Lenstra JK. Job Shop Scheduling by local search. COSOR Memorandum 94-05. Eindhoven University; 1994.

[12] Dauzére-Pérés S, Paulli J. An integrated approach for modeling and solving the general multiprocessor job-shop scheduling problem using tabu search. Annals of Operations Research 1997;70:281–306.

[13] Hurink J, Jurish B, Thole M. Tabu search for the job shop scheduling problem with multi-purpose machines. OR-Spektrum 1994;15:205–15.
[14] Jia HZ, Nee AYC, Fuh JYH, Zhang YF. A modified genetic algorithm for distributed scheduling problems. International Journal of Intelligent Manufacturing 2003;14:351–62.
[15] Ho NB, Tay JC. GENACE: an efficient cultural algorithm for solving the Flexible Job-Shop Problem. IEEE international conference on robotics and automation 2004;1759–66.
[16] Tay JC, Wibowo D. An Effective Chromosome Representation for evolving flexible job shop schedules, GECCO 2004. In: Lecture notes in computer science, vol. 3103. Berlin: Springer; 2004. p. 210–21.
[17] Lee KM, Yamakawa T, Lee KM. A genetic algorithm for general machine scheduling problems. International Journal of Knowledge-Based Electronic 1998;2:60–6.
[18] Fisher H, Thompson L. Probabilistic learning combinations of local job shop scheduling rules. In: Muth JF, Thompson GL, editors. Industrial scheduling. Englewood Cliffs, NJ: Prentice-Hall; 1968. p. 225–51.
[19] Lawrence S. An experimental investigation of heuristic scheduling techniques. In: Supplement to resource constrained project scheduling, GSIA. Pittsburgh, PA: Carnegie Mellon University; 1984.