Genetic Algorithm and Local Search for Just-in-Time Job–Shop Scheduling

Rodolfo Pereira Araujo, André Gustavo dos Santos, José Elias Cláudio Arroyo

Abstract— **This paper describes a successful combination of genetic algorithm and local search procedure to find good solutions for just-in-time job-shop scheduling problem with earliness and tardiness penalties. For each job is given a specific order of machines in which its operations must be processed, and each operation has a due date, a processing time, and earliness and tardiness penalties, which are paid if the operation is completed before or after its due date. The problem is very hard to solve to optimality even for small instances, but the proposed genetic algorithm found good solutions for some problem instances, even improving its performance when a local search procedure is invoked as an additional phase. The quality of the solutions is evaluated and compared to a set of instances from the literature, with up to 20 jobs and 10 machines. The proposed algorithm improved the solution value for most of the instances.**

I. Introduction

JUST-IN-TIME scheduling problems are very common in industry. In the just-in-time (JIT) scheduling environment, the job should be finished as close to the due date as possible. An early job completion results in inventory carrying costs, such as storage and insurance costs. On the other hand, a tardy job completion results in penalties, such as loss of customer goodwill and damaged reputation. A lot of research effort has been devoted to such problems considering both earliness and tardiness [1]. The main focus has been the single machine problems [2], moreover only the tardiness is considered for the multiple machine problems [3], due to the intractable nature of the JIT job shop scheduling problems. There are some few papers considering multiple machines and both earliness and tardiness penalties: in [4] and [5], for example, all operations, except the last ones of each job, have null penalties for earliness and tardiness. As the earliness penalty is paid only for the last operation, no storage cost is considered for intermediary operations: the jobs may be finished on time, but nothing is done about the intermediary operations, which are processed as early as possible, maybe leading in waiting lines (and then storage costs) between those operations and the last operation of the respective job, which contradicts the just-in-time philosophy.

The definition of the just-in-time job shop scheduling problem (JITJSSP) considered in this paper is based on [6]. There is a set of n jobs, $J = \{J_1, J_2, \dots, J_n\}$ and a set of m machines, $M = \{M_1, M_2, \dots, M_m\}$. Each job J_i is a sequence of m ordered operations, $O_i = \{o_i^1, o_i^2, \dots, o_i^m\}$, where o_i^k is the k-th operation of job J_i. Each operation o_i^k has a due date d_i^k, a processing time p_i^k, and a specific machine $M(o_i^k) \in M$ where the given job must be processed. In order to evaluate the cost of a solution, each operation o_i^k has also two penalty coefficients, α_i^k and β_i^k, penalizing its early and tardy completion. Let C_i^k be the completion time of operation o_i^k and $E_i^k = \max(0,\ d_i^k - C_i^k)$ and $T_i^k = \max(0, C_i^k - d_i^k)$ its earliness and tardiness. An operation o_i^k is early if $E_i^k > 0$ and tardy if $T_i^k > 0$.

The objective is to compute a feasible schedule, i.e., completion times C_i^k for all operations o_i^k, minimizing the total earliness and tardiness costs of all operations (also named total penalty), given by

$$min \sum_{i=1}^{n} \sum_{k=1}^{m} (\alpha_i^k E_i^k + \beta_i^k T_i^k). \quad (1)$$

A feasible schedule must obey the precedence constraints: for each pair of consecutive operations o_i^{k-1} and o_i^k of the same job, operation o_i^k cannot be started before operation o_i^{k-1} is completed, i.e., for $i = 1, \dots, n$ and $k = 1, \dots, m$,

$$C_i^k \geq C_i^{k-1} + p_i^k. \quad (2)$$

For this constraint to make sense even for the first operation, consider $C_i^0 = 0$ for $i = 1, \dots, n$, which means that the first operation of each job starts on or after time 0.

It must also obey the resource constraints: two operations of two distinct jobs o_i^k and o_j^h that must be processed in the same machine cannot be processed simultaneously, i.e., for $i, j = 1, \dots, n$, $i \neq j$ and for $k = 1, \dots, m$ and $h = 1, \dots, m$ where $M(o_i^k) = M(o_j^h)$,

$$C_i^k \geq C_j^h + p_i^k \ \text{ or } \ C_j^h \geq C_i^k + p_j^h. \quad (3)$$

For the JITJSSP, Baptiste, Flamini and Sourd [6] proposed two types of Lagrangian relaxation of a mixed linear programming problem, and obtained lower bounds for a set of instances of different characteristics and sizes. They also reported upper bounds found by heuristics over the

Manuscript received November 13, 2008. This work was supported by the FAPEMIG – Fundação de Amparo à Pesquisa do Estado de Minas Gerais, under Grant APQ3768-6.01/07.

R. P. Araujo, A. G. Santos and J. E. C. Arroyo are with the Computer Science Department, Viçosa Federal University, Campus UFV, 36570-000, Viçosa, MG, Brazil (phone: +55 (31) 3899-2396; fax +55 (31) 3899-2394; e-mail: rodolfo.araujo@ufv.br, andre@dpi.ufv.br, jarroyo@dpi.ufv.br)

978-1-4244-2959-2/09/$25.00 ©

relaxed solutions and by the ILOG CPLEX package. Their method could even prove optimality for one of the instances, with 10 jobs and 2 machines.

The job shop scheduling problem (JSSP) is one of the best-known machine scheduling problems, and is among the hardest combinatorial optimization problems. The JITJSSP addressed in this paper is an extension of the JSSP which is known to be NP-hard [7][8].

In the past few years, the genetic algorithm (GA) has received a rapidly growing interest in the combinatorial optimization community and has shown great power with very promising results from experimentation and practice of many engineering areas. Therefore, some researchers proposed several GA-based approaches to solve the JSSP [9].

Local search procedures to improve solutions are widely used in GAs just as in other metaheuristic approaches for the JSSP [10][11]. This is especially important in GAs for scheduling where genetic operators are unable to carry out the fine improvement that a simple local search method can.

In this paper we present an efficient genetic algorithm with a local search procedure to find good solutions for the JITJSSP. The quality of the obtained solutions are evaluated and compared to a set of problem instances with up to 20 jobs and 10 machines, generated in [6].

This article is organized as follows. In Section II the details of proposed GA are presented. Section III describes the local search procedure. The computational experiments are presented in Section IV. Then Section V summarizes the conclusions of the work.

II. GENETIC ALGORITHM

In the following subsections we describe the characteristics of the proposed genetic algorithm for the JITJSSP.

A. Chromosome representation

In our genetic algorithm each chromosome represents a feasible solution, encoded on a vector of integers that contains numbers from 1 to *n* (representing the jobs), each one *m* times (representing the operations of each job). These numbers represents the order in which the jobs are to be scheduled on the machines. Each operation is scheduled as early as possible, according to the precedence and resource constraints. Consider for example one possible solution for a 4 jobs 2 machines problem, represented in Fig. 1

1	2	4	1	4	3	2	3

Fig. 1. Sequence of job operations as it is represented in a chromosome

As the first gene has the value 1, the first operation of job 1 is scheduled on the machine starting at time 0. The second gene has the value 2, so the first operation of job 2 is scheduled at time 0 if machine is empty, or at time if this machine is busy, i.e., if it has to be processed in the same machine as job 1 (which will be free at time).

The third gene has the value 4, and it will be scheduled to start at machine on time 0, , , or , depending on when this machine will be free. This depends on which operations were scheduled to this machine: none, one, or two of the previous job operations scheduled.

Now the forth gene is again 1, which means that the second operation of job 1 is the next to be scheduled (as this is the second gene 1). It will be scheduled as early as possible on machine at time (because of the precedence constraint) or when the operation currently on this machine is completed (because of the resource constraint), whichever time comes after.

The same idea is applied to all other operations, in the order 4, 3, 2 and 3, as it appears on the chromosome. If a gene has value *i*, and this is the *k*-th occurrence of this value, it will be scheduled on machine at time , where is the completion time of the last operation scheduled in machine , or 0 if no operation is scheduled to that machine yet.

For example, consider a 4 jobs 2 machines problem, where jobs and must be processed first in the machine 1 and jobs and must be processed first in the machine 2, i.e.:

= = = = 1

= = = = 2

Fig. 2 shows the schedule of the operations in the order they are given in the chromosome of Fig. 1. The size of the rectangle indicates the relative amount of processing time. Although machine 1 is free at time t_a after operation of is completed, the operation of of cannot be started before time t_b, because operation is still being processed in machine 2 (precedence constraint). The first occurrence of value 3 in the chromosome indicates the operation of , but it cannot be processed before time t_c because the machine is busy processing (resource constraint). At time t_d operation is completed in machine 1 then starts in machine 2.

Note that, in this case, other orders of genes could lead the same schedule, for example 1 2 4 4 1 3 3 2.

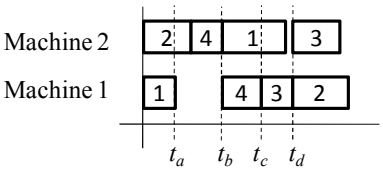

Fig. 2. Schedule of operations in the order given in Fig.1.

There is a total of / ! possible chromosomes, all representing feasible solutions. It is a very huge number, even for small number of jobs and machines. As the due dates are generally very tight, almost not allowing any idle time, we may consider that any possible solution can be represented by some sequence.

The fitness of a chromosome is the total earliness and tardiness costs total penalty of the represented solution, and can be calculated while the operations are being scheduled by the above procedure, as all completion times are set deterministically. The genetic algorithm should search for the best chromosome, i.e., the one who gives the best scheduling sequence, whose total earliness and tardiness costs of all operations is minimal.

B. Initial population

The initial population contains 200 chromosomes generated randomly. Although a constructive heuristic can be used, very good results were obtained by this simple idea, so for now we do not use any other more complex schema.

C. Selection phase

On each iteration the whole population is substituted by a new one, of the same size, preserving only the best chromosome. The chromosome with the best fitness value is always kept for the next generation (elitism), to avoid losing a good solution. Each other chromosome of the new population is selected by a ternary tournament: three chromosomes are randomly chosen, and the one with best fitness (lowest total penalty) is selected.

D. Crossover operator

The chromosomes of the selected population are combined by a crossover operation. A whole new population is created by crossover; only the best chromosome of the selected population is directly copied, all others are generated by the following crossover operator.

Two chromosomes are randomly selected; their genes are combined yielding two new chromosomes, and only one survives for the next iteration, the one that has the best fitness. The crossover operator used to combine the genes was proposed in [12]. The set of n jobs is randomly partitioned in two non-empty subsets S_1 and S_2, not necessary having equal cardinality. The first gene of parent 1 goes to child 1 if this gene belongs to the subset S_1 or to child 2 if it belongs to subset S_2. Then, the first gene of parent 2 goes to child 2 if this gene belongs to the subset S_1 or to child 1 if it belongs to the subset S_2. The same process is repeated to the second gene of parent 1 and second gene of parent 2, and so on, scanning the genes of both parents from left to right. The child 1 is made by the genes of parent 1 that belongs to set S_1 and by the genes of the parent 2 that belongs to set S_2, in the order they appear on each parent. The remaining genes go to child 2. This process is illustrated in Fig. 3.

This procedure produces two feasible solutions whose fitness are calculated, and only the best child takes part in the new population.

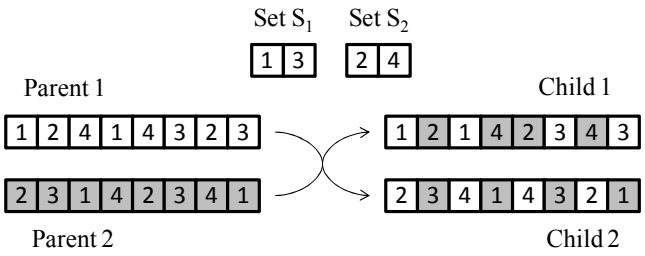

Fig. 3. Crossover operator using two subsets randomly chosen.

E. Mutation operator

The mutation operator is used to diversify the population and consists in swapping randomly two genes in the chromosome, i.e., swapping the order of two operations in the sequence, as shown in Fig. 4. The best chromosome is copied without mutation, and each other chromosome has a probability of p_m = 5% to be changed by the mutation operator. This value was obtained experimentally.

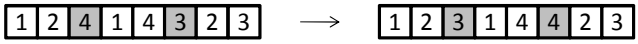

Fig. 4. Mutation operator swapping two randomly chosen operations.

F. Stop criterion

The population goes iteratively through the selection phase, then the crossover operator is applied and finally the mutation operator. This is done $numIter$ times, and the best solution in the last population is returned, as shown in Fig. 5.

The results presented in section IV were obtained with $numIter$ = 100. As the genetic algorithm lasts for only few seconds, it is not worth to use less iterations than that. Experimentally we noted that beyond this value the solution is rarely improved. Then, a local search is called to work on the best solution obtained, which is a better attempt to improve the solution than to increase the number of iterations in the genetic algorithm.

Genetic Algorithm – GA
1. $P \leftarrow$ RandomPopulation
2. **repeat** $numIter$ **times**
3. $P' \leftarrow$ Selection(P)
4. $P'' \leftarrow$ Crossover(P')
5. $P''' \leftarrow$ Mutation(P'')
6. $P \leftarrow P'''$
7. **end repeat**
8. **return** Best Solution in P

end GA

Fig. 5. A pseudocode for the proposed Genetic Algorithm

III. Local Search

The solution returned by the Genetic Algorithm is submitted to a local search procedure that explores its neighborhood by swapping consecutive integers of the solution vector (genes in the chromosome). As a solution contains n integers, each solution has $n - 1$ neighbors, which corresponds to

swapping each pair of consecutive integers. At first we tried two known strategies: expand the whole neighborhood and restart the local search with the best neighbor if it is better than the current solution; or restart the local search as soon as a better solution is found among the neighbors. In both cases, the local search stops only when no neighbor is better than the current solution.

However, we also propose a new strategy and better results were obtained by this new Local Search: instead of keeping only one current solution there is a limited size list L of potential solutions. The list is always sorted by solution value, and the best solutions are expanded first (like the best first strategy, but avoiding to explore the whole feasible space). Once a solution is expanded, the list is updated keeping the best solutions among the solutions already in the list and the neighbors of the expanded solution.

There is not an early termination criterion. The local search stops when all solutions in the list have already been expanded, i.e., when the neighborhood of all solutions in the list has been considered and no new solution is included in the list. This local search procedure always stops, because each time a solution is expanded a neighbor is included in the list only if it is better than all solutions in the list.

The local search is presented in Fig. 6 in pseudo-code. The union operator ∪ "merges" two lists, returning an ordered list containing the best $|L|$ elements among the elements of the two lists. Note that it may be the case that no solution is added in the list (when all neighbors are worse than the worst solution in the list), or the list are completely updated (when all neighbors are better than the best solution in the list), or the list is only partially updated. Note also that the list may contain both kinds of solutions, expanded and not expanded, and the currently expanding solution may remain in the list if it is still among the best $|L|$ so far generated. The great advantage is that even if no neighbor is better than the expanded solution, they may still be added to the list, depending of the other solutions in the list, and may be expanded in future iterations, thus avoiding some local minima.

Local Search – LS
1. L ← solution returned by GA
2. **while** there is solution in L not expanded **do**
3. ⬚ ← First solution in L not expanded
4. Mark ⬚ as expanded
5. N ← list of neighbors of ⬚
6. L ← $L \cup N$
7. **end while**
8. **return** First solution in L

end LS

Fig. 6. The pseudocode for the Local Search procedure

The memory requirement is ⬚⬚ ⬚⬚L⬚⬚⬚ ⬚ ⬚⬚⬚ ⬚ because the list L has at most ⬚ ⬚⬚L solutions and list N has ⬚⬚⬚ ⬚ ⬚⬚solutions, and each of these solutions has size ⬚⬚ .

IV. Experimental Results

This section shows the experimental results of the genetic algorithm, and of the genetic algorithm with local search. In the local search procedure we use ⬚ ⬚⬚L = 10 for the size of list L of potential solutions. Using a shorter list the local search may become too greedy and lose its ability to escape from some local minima. We tried longer lists, but the execution time is increased with no significant improvement in solution quality.

The algorithms were implemented and tested with JITJSSP instances available for comparison in the literature.

A. Instances

The instances were created by Baptiste, Flamini and Sourd [6]. There are eight instances for each combination of ⬚⬚⬚⬚ ⬚⬚ ⬚⬚⬚⬚⬚⬚5⬚2⬚⬚× ⬚2⬚5⬚⬚⬚⬚ In all instances the jobs are processed exactly once on each machine, and the machine where each the operation is to be performed is chosen randomly, with processing times chosen in [⬚⬚⬚3⬚]. The instances are named `I-n-m-DD-W-ID`, where:

- `DD` is the distance between due dates of consecutive operations, which could be exactly equal to the processing time of the last operation (`DD = tight`), or this processing time plus a random number in [⬚⬚⬚⬚] (`DD = loose`).
- `W` is the relative value of earliness and tardiness costs ⬚⬚⬚⬚, which could be both chosen randomly in [⬚⬚⬚⬚] (`W = equal`) or ⬚ in [⬚⬚⬚⬚3] and ⬚ in [⬚⬚⬚⬚] (`W = tard`).
- Two instances are available for each combination of parameters (`ID = 1`) and (`ID = 2`).

This leads to a total of 72 instances, all available on-line [13]. For these instances only one proved optimal solution is currently known.

B. Results

The algorithms were implemented in the C++ programming language, and the results were obtained on a desktop Intel Pentium IV 3.0 GHz machine, with 2 GB RAM.

The quality of the obtained results is evaluated and compared to those in [6]. Although the main focus on that paper is derive good lower bounds, they also find upper bounds heuristically, based on the lagrangian lower bounds. As our algorithm produces feasible solutions, we evaluate the quality of the solutions comparing them to their feasible solutions, i.e., to the upper bounds. The solutions are evaluated according to the percentual improvement of the known upper bound. So, the evaluation criterion is given by:

$$\%I⬚ ⬚_{GA} ⬚ \frac{⬚_{GA} ⬚ UB}{UB} × ⬚⬚⬚$$

$$\%I⬚ ⬚_{GA+LS} ⬚ \frac{S_{GA+LS}⬚ UB}{UB} × ⬚⬚⬚,$$

where z_{GA} and z_{GA+LS} are the solutions values obtained by the Genetic Algorithm without and with Local Search, and UB is the best already known upper bound. Note that a negative percentual means that the genetic algorithm found a better solution, thus improving the upper bound. The results are listed on Tables I-III, and contain the following data.

LB and UB: best Lower Bound and Upper Bound published in [6]. The Lower Bounds are calculated by Lagrangian relaxation of precedence constraints, Lagrangian relaxation of the machine constraints, and the one returned by ILOG CPLEX after 600 seconds. The Upper Bound is the best of those found by ILOG CPLEX after 600s, or by a local search procedure applied on a solution build heuristically over the Lagrangian solutions.

Δ_{GA} and Δ_{GA+LS}: the percentual improvement (if negative) over the known Upper Bound obtained respectively by the Genetic Algorithm alone, and the Genetic Algorithm with Local Search.

The three columns named (s) give the computational time, in seconds, spent respectively by the algorithm in [6] that finds the best Lower Bound, the Genetic Algorithm and the Genetic Algorithm with Local Search.

Tables I, II and III show the results for instances with 10, 15 and 20 jobs, respectively. In each Table, the instances are grouped in four groups, according to the values of due dates (ti = tight and lo = loose) and penalty costs (eq = equal and ta = tard).

TABLE I
RESULTS FOR INSTANCES WITH 10 JOBS

Instance	LB	UB	(s)	Δ_{GA}	(s)	Δ_{GA+LS}	(s)
I-10-2-ti-eq-1	433	453	8	3.7	1	3.2	1
I-10-2-ti-eq-2	357	458	7	**-1.3**	0	**-1.3**	0
I-10-5-ti-eq-1	536	826	61	20.6	1	17.6	1
I-10-5-ti-eq-2	592	848	23	**-5.5**	1	**-6.1**	2
I-10-10-ti-eq-1	812	1439	240	5.9	1	0.6	4
I-10-10-ti-eq-2	819	2006	560	42.4	2	29.2	4
I-10-2-lo-eq-1	218	225	6	**-0.1**	1	**-0.1**	1
I-10-2-lo-eq-2	298	324	7	3.9	0	3.9	0
I-10-5-lo-eq-1	1205	1905	61	1.3	1	0.6	2
I-10-5-lo-eq-2	780	1010	104	15.6	1	14.6	1
I-10-10-lo-eq-1	294	376	222	241.9	2	211.5	5
I-10-10-lo-eq-2	211	260	170	168.8	2	107.3	6
I-10-2-ti-ta-1	168	195	7	**-7.8**	1	**-7.8**	1
I-10-2-ti-ta-2	138	147	35	3.0	0	3.0	0
I-10-5-ti-ta-1	322	405	230	16.1	1	15.6	1
I-10-5-ti-ta-2	420	708	26	**-7.9**	1	**-8.0**	1
I-10-10-ti-ta-1	408	855	551	7.9	2	1.2	4
I-10-10-ti-ta-2	469	800	588	60.2	2	48.3	5
I-10-2-lo-ta-1	416	416	12	4.6	1	3.3	1
I-10-2-lo-ta-2	135	138	9	14.4	0	14.4	0
I-10-5-lo-ta-1	159	188	79	83.7	1	62.1	2
I-10-5-lo-ta-2	313	572	40	**-4.7**	0	**-7.2**	1
I-10-10-lo-ta-1	314	409	345	82.1	1	56.8	4
I-10-10-lo-ta-2	119	152	152	116.9	2	98.6	4

The negative values in columns Δ_{GA} and Δ_{GA+LS} are written in bold face to point out the instances in which the algorithms improved the known Upper Bound.

The results in Table I show that the Genetic Algorithm was able to improve the Upper Bound of 6 out of 24 instances, and for several instances with "loose" dual date the solutions are very far from the known Upper Bound. The Local Search could not improve the results. In fact, in 3 of them the Genetic Algorithm already found a local minimum, and for others the improvement of the Local Search was not significant, so the Local Search was almost helpless.

Better results were obtained for instances with 15 jobs, as shown in Table II. The Genetic Algorithm alone could improve the Upper Bound for 17 out of 24 instances, half of them with more than 10% of improvement. Using the Local Search we have improved the Upper Bound of one more instance, and for 10 instances the improvement was by more than 10%, reaching more than 20% in 3 instances.

TABLE II
RESULTS FOR INSTANCES WITH 15 JOBS

Instance	LB	UB	(s)	Δ_{GA}	(s)	Δ_{GA+LS}	(s)
I-15-2-ti-eq-1	2902	3559	136	**-6.0**	0	**-6.0**	0
I-15-2-ti-eq-2	1253	1579	17	**-6.3**	1	**-6.3**	1
I-15-5-ti-eq-1	964	1663	26	**-11.5**	1	**-13.2**	2
I-15-5-ti-eq-2	1630	2989	79	**-7.6**	2	**-9.5**	3
I-15-10-ti-eq-1	3662	8381	1138	**-4.8**	2	**-8.9**	13
I-15-10-ti-eq-2	2564	7039	2454	**-19.9**	3	**-24.3**	15
I-15-2-lo-eq-1	1014	1142	13	**-7.8**	1	**-7.8**	1
I-15-2-lo-eq-2	472	520	56	0.9	0	0.7	0
I-15-5-lo-eq-1	2449	4408	218	**-20.5**	2	**-21.5**	2
I-15-5-lo-eq-2	2773	4023	676	**-12.3**	2	**-14.5**	2
I-15-10-lo-eq-1	628	1109	1836	54.2	3	36.4	17
I-15-10-lo-eq-2	979	2256	2880	45.9	3	17.0	22
I-15-2-ti-ta-1	720	913	16	**-10.6**	0	**-10.6**	0
I-15-2-ti-ta-2	843	956	11	**-5.3**	1	**-5.3**	1
I-15-5-ti-ta-1	1008	1538	39	**-8.6**	1	**-9.0**	2
I-15-5-ti-ta-2	547	843	10280	**-11.5**	2	**-13.9**	2
I-15-10-ti-ta-1	467	972	141283	15.0	3	9.8	18
I-15-10-ti-ta-2	761	1656	81347	**-5.9**	2	**-12.6**	19
I-15-2-lo-ta-1	616	730	13	**-9.9**	1	**-10.3**	1
I-15-2-lo-ta-2	277	310	148	**-5.6**	1	**-8.4**	1
I-15-5-lo-ta-1	1005	1723	3538	**-18.3**	2	**-20.2**	4
I-15-5-lo-ta-2	313	374	3384	38.2	2	31.4	4
I-15-10-lo-ta-1	233	312	1295	95.9	3	66.5	17
I-15-10-lo-ta-2	454	855	3586	3.3	4	**-13.8**	21

In Table II, it is worth to note that in 4 instances the Local Search does not improve the solution found by the Genetic Algorithm, but the solution was already better than the known Upper Bound, which shows that the Genetic Algorithm perform very well for the JITJSSP. And the algorithm is very fast, spending at most 4 seconds for each instance and up to 4 seconds even with the local search for

the instances with 2 or 5 machines. For the 10 machines instances, the time was from 13 to 22 seconds, which is very fast, if compared to the Upper Bound obtained by CPLEX or to the heuristics over lagrangian relaxations of [6], which spend minutes or even hours of computation for some instances.

The results for the instances with 20 jobs are reported in Table III. Despite that there are more jobs on these instances, increasing substantially the number of feasible solutions, and then the feasible space, the Genetic Algorithm still works very well, and in few seconds can reach much better solutions in average than CPLEX or the heuristics based on Lagrangian relaxations. The Genetic Algorithm alone improve the Upper Bound in 20 out of 24 instances, and the local search procedure can improve even more these solutions, leaving only 1 instance with solution worst than the known Upper Bound, but in this case, the known Upper Bound was only 0.4% better. It is also worth to point out that the execution time was minimal, generally seconds, and above one minute in only 6 instances, while the other methods spend much more time.

TABLE III
RESULTS FOR INSTANCES WITH 20 JOBS

Instance	LB	UB	(s)		(s)		(s)
I-20-2-ti-eq-1	1747	2008	16	**-2.7**	1	**-3.2**	1
I-20-2-ti-eq-2	858	1014	14	**-5.5**	1	**-6.0**	2
I-20-5-ti-eq-1	2244	3090	418	3.9	2	**-0.6**	9
I-20-5-ti-eq-2	4923	7537	329	**-8.3**	3	**-9.4**	7
I-20-10-ti-eq-1	6656	12951	1226	**-10.5**	4	**-17.9**	60
I-20-10-ti-eq-2	5705	9435	1190	**-12.5**	4	**-16.5**	50
I-20-2-lo-eq-1	2388	2708	25	**-5.0**	2	**-5.2**	2
I-20-2-lo-eq-2	2970	3318	63	**-7.0**	1	**-7.1**	1
I-20-5-lo-eq-1	5571	9697	639	**-16.3**	3	**-20.0**	8
I-20-5-lo-eq-2	5496	8152	518	**-6.4**	2	**-10.5**	9
I-20-10-lo-eq-1	3099	6732	5083	**-3.9**	5	**-24.2**	76
I-20-10-lo-eq-2	1150	2516	2268	**-3.9**	4	**-26.7**	69
I-20-2-ti-ta-1	1515	1913	26	**-12.3**	2	**-12.5**	2
I-20-2-ti-ta-2	1327	1594	805	**-9.0**	1	**-9.0**	1
I-20-5-ti-ta-1	2507	4147	142	**-9.4**	3	**-11.1**	7
I-20-5-ti-ta-2	1633	1916	154	0.4	2	**-3.9**	7
I-20-10-ti-ta-1	2764	5968	9704	**-13.8**	5	**-22.0**	74
I-20-10-ti-ta-2	2740	3788	865	7.2	5	**-9.0**	104
I-20-2-lo-ta-1	1189	1271	447	**-5.0**	1	**-5.0**	1
I-20-2-lo-ta-2	734	857	13	**-9.3**	1	**-10.2**	1
I-20-5-lo-ta-1	2177	3377	249	**-7.5**	2	**-12.0**	7
I-20-5-lo-ta-2	2643	5014	147	**-21.5**	2	**-25.0**	5
I-20-10-lo-ta-1	2436	6237	22196	**-0.5**	4	**-10.8**	56
I-20-10-lo-ta-2	1226	1830	392	26.9	4	0.4	82

The percentage of improvement obtained by the Local Search compared to the Genetic Algorithm itself, can be measured by:

$$\frac{\square_{\square\square} - \square_{\square\square\square\square\square}}{\square_{\square\square}} \times \square\square\square$$

and it was of 4.9% in average. It was higher for instances with "loose" due dates, 6.8% in average, whilst for instances with "tight" due dates it was 3.1% in average. It also increases with the number of jobs, going from 4.3% for instances with 10 jobs to 6.1% for instances with 20 jobs.

V. CONCLUSIONS

In this paper, a genetic algorithm is presented to solve the JITJSSP. The instances consider penalty costs for both earliness and tardiness for all operations, not only for the jobs. The GA is combined with a local search procedure.

The algorithms were tested with 72 instances and the results compared to upper bounds recently published, obtained by heuristics upon Lagrangian relaxation and mixed integer programming.

The GA is very effective and efficient. It can find good solutions for most of the test instances, and the running time is much less than the other algorithms for all instances.

The genetic algorithm alone can find very good results, improving the upper bounds in 43 out of 72 instances, with minimal computational effort. The local search improves even more some of these solutions, and also finds better upper bounds for some others, in a total of 47 out of 72 instances.

An interesting observation is that the improvement becomes even more significant when the instances have more jobs. For the last set, with 20 jobs for 2, 5 and 10 machines, it performed better comparing to other methods in 23 out of 24 instances, being worse by just 0.4% in the remaining instance. This indicates that although the performance of mixed integer programming and lagrangian relaxation methods tend to decrease in performance as the instances grow in size, the performance of our genetic algorithm with local search remains similar as it was in small instances, and not even the execution time has increased significantly.

As continuation of this work, we intend apply Local Search procedure to improve the quality of the solutions during genetic algorithm, not only of the final solution.. Also, apply an intensification procedure based in the Path Relinking technique [14] exploring a path between two elite solutions.

REFERENCES

[1] K. R. Baker, and G. D. Scudder, "Sequencing with earliness and tardiness penalties: A review", *Operations Research*, vol. 38(1), pp.22–36, 1990.

[2] F. Sourd and S. Kedad-Sidhoum, "The one machine problem with earliness and tardiness penalties", *Journal of Scheduling*, vol. 6, pp.533–49, 2003.

[3] D. C. Mattfeld and C. Bierwirth, "An efficient genetic algorithm for job shop scheduling with tardiness objectives", *European Journal of Operational Research*, vol. 155, pp.616–630, 2004.

[4] J. C. Beck and P. Refalo, ‘‘A hybrid approach to scheduling with earliness and tardiness costs’’, *Annals of Operations Research* vol. 118, pp.49–71, 2003
[5] J. Kelbel and Z. Hanzálek. “Constraint Programming Search Procedure for Earliness/Tardiness Job Shop Scheduling Problem” in *Proc. of the 26th Workshop of the UK Planning and Scheduling Special Interest Group*. pp. 67–70, 2007.
[6] P. Baptiste, M. Flamini and F. Sourd, “Lagrangian bounds for just-in-time job-shop scheduling”, *Computers & Operations Research*, vol. 35, pp.906–915, 2008.
[7] M. R. Garey and D. S. Johnson, *“Computers and Intractability: A Guide to the Theory of NP-Completeness”*, Freeman, New York, 1979.
[8] J. Du, J. Y. T. Leung, “Minimizing total tardiness on one machine is NP-hard”, *Mathematics of Operations Research*, vol. 15, pp.483–495, 1990.
[9] A. S. Jain and S. Meeran, “A state-of-the-art review of job-shop scheduling techniques”, *European Journal of Operational Research*, vol. 113, pp.390–434, 1999.
[10] J. F. Gonçalves, J. J. M. Mendes and M. G. C. Resende, “A hybrid genetic algorithm for the job shop scheduling problem”, *European Journal of Operational Research*, vol. 167, pp.77–95, 2005.
[11] J. Gao, L. Sun and M. Gen, “A hybrid genetic and variable neighborhood descent algorithm for flexible job shop scheduling problems”, *Computers & Operations Research*, vol. 35(9), pp.2892–2907, 2007.
[12] H. IIma, “Proposition of selection operation in a genetic algorithm for a job shop rescheduling problem”, in *Proc. EMO – Evolutionary Multi-Criterion Optimization*, Guanajuato, Mexico, pp.721–735, 2005
[13] P. Baptiste, M. Flamini and F. Sourd, “Job-shop scheduling with earliness-tardiness penalties” [Online]. Available: http://www-poleia.lip6.fr/~sourd/project/etjssp/
[14] F. Glover, M. Laguna and R. Martí. “Fundamentals of scatter search and path relinking”, *Control and Cybernetics*, vol. 39, pp.653–684, 2000.