ELSEVIER

European Journal of Operational Research 132 (2001) 224–242

EUROPEAN JOURNAL OF OPERATIONAL RESEARCH

www.elsevier.com/locate/dsw

Theory and Methodology

A memetic algorithm for the total tardiness single machine scheduling problem

Paulo M. França *, Alexandre Mendes, Pablo Moscato

Faculdade de Engenharia Elétrica e de Computação – FEEC, Universidade Estadual de Campinas – UNICAMP, C.P. 6101, 13081-970 Campinas-SP, Brazil

Received 19 May 1999; accepted 18 April 2000

Abstract

In this paper, a new memetic algorithm (MA) for the total tardiness single machine scheduling (SMS) problem with due dates and sequence-dependent setup times is proposed. The main contributions with respect to the implementation of the hybrid population approach are a hierarchically structured population conceived as a ternary tree and the evaluation of three recombination operators. Concerning the local improvement procedure, several neighborhood reduction schemes are developed and proved to be effective when compared to the complete neighborhood. Results of computational experiments are reported for a set of randomly generated test problems. The memetic approach and a pure genetic algorithm (GA) version are compared with a multiple start algorithm that employs the all-pairs neighborhood as well as two constructive heuristics. © 2001 Elsevier Science B.V. All rights reserved.

Keywords: Scheduling; Memetic algorithms; Hybrid genetic algorithms; Single machine scheduling; Sequence-dependent setup times

1. Introduction

Under the generic denomination of single machine scheduling (SMS), we understand one of the first studied class of problems in the scheduling area, as the survey papers by Graham et al. [12] and Graves [13] show. There are many different kinds of SMS problems (generally due to different input data and objective functions). One of the simplest to state, but not easy to solve, is the problem of sequencing n jobs, given its processing times and due dates (distinct for each job), and with the objective function being to minimize the total tardiness. Tardiness is a *regular* performance measure, which means that an optimal schedule cannot have idle times between jobs [1,15]. Due to this fact, a valid solution of the problem can be defined as a permutation of the jobs. Using the permutation space representation, the solution

* Corresponding author. Tel.: +55-19-788-3774; fax: +55-19-289-1395.

E-mail addresses: franca@densis.fee.unicamp.br (P.M. França), smendes@densis.fee.unicamp.br (A. Mendes), moscato@densis.fee.unicamp.br (P. Moscato).

0377-2217/01/$ - see front matter © 2001 Elsevier Science B.V. All rights reserved.
PII: S 0 3 7 7 - 2 2 1 7 (0 0) 0 0 1 4 0 - 5

space has $n!$ configurations and all of them represent valid solutions of the problem.

The original SMS problem can be extended by the inclusion of sequence dependent setup times, precedence constraints for the jobs, ready times, etc. From an objective function's point of view (in a real-world scenario), we may want to minimize the makespan, the total tardiness, the mean tardiness, the number of tardy jobs, or even a combined set of these objectives, which would characterize a multi-criteria problem. Therefore, it is easy to see the large variety of problems, which we may face in practice (see Chapter 7 in Ref. [9]).

The SMS problem with sequence-dependent setup times is present in many industrial manufacturing systems, as pointed out in Ref. [24]. In this problem, the processing of a job requires a setup time, which depends on the predecessor job. In real-world situations, this setup may involve tooling change, cleaning, transportation of parts (if jobs i and j are done in different parts), etc. The extended problem inherits some characteristics of the original one (without sequence-dependent setups), such as no idle times being allowed between jobs, but the dominance rules which usually lead to effective solution methods are not valid anymore in this case.

The SMS problem we will address in this paper can be described as follows:

Input: Let n be the number of jobs to be processed in one machine. Let $\{p_1, \ldots, p_n\}$ be the list of processing times for each one of them, and $\{d_1, \ldots, d_n\}$ be the list of due dates for each job. Let $\{s_{ij}\}$ be a matrix of setup costs, where s_{ij} is the time required to setup job j after the machine has just finished processing job i.

Task: Find a permutation that minimizes the total tardiness of the schedule, which is calculated as

$$T = \sum_{k=1}^{n} \max[0, c_k - d_k],$$

where c_k represents the completion time of job k (i.e., the time when the machine finishes processing job k).

It is well known that the problem of sequencing jobs in one machine without setup times is already NP-hard [8]. Despite its application to real world settings, the SMS problem addressed in this article has received little attention in the scheduling literature. Ragatz [24] proposes a *branch-and-bound* (B&B) method but only small instances can be solved to optimality. The papers of Raman et al. [25] and Lee et al. [16] use dispatching rules based on the calculation of a priority index to build an approximate schedule, which is then improved by the application of a local search procedure. Two papers using metaheuristics have been proposed so far. Rubin and Ragatz [26] developed a new crossover operator and applied a *genetic algorithm* (GA) to a set of test problems. The results obtained by the GA approach were compared with the ones from a B&B and with a *multiple start* (MS) algorithm and they concluded that MS outperformed B&B and GA in many instances, considering running time and quality of solutions as performance measures. Of course, the instances in which MS outperformed B&B were the ones where the exact method did not find an optimal solution before a limit on the number of nodes was reached. The B&B was truncated in such cases, returning sub-optimal schedules. Given these results, Tan and Narasimhan [27] chose the MS technique as a baseline benchmark for conducting comparisons with the *simulated annealing* (SA) approach they proposed. Their conclusion was that SA outperformed MS in all but three instances, with percentage improvements not greater than 6%.

In this paper, we propose a GA and a *memetic algorithm* (MA) to solve the SMS problem with sequence-dependent setup times. The MA algorithm implemented combines the strengths of the hierarchical population approach with the intensification power of an ad hoc local search procedure. MAs are being used for several NP optimization problems such as quadratic assignment [17], graph bipartitioning [19], scheduling and timetabling [3–6,23]. Several recombination schemes (crossovers) jointly with mutation and local search procedures are evaluated and the results are compared against the constructive heuristic ATCS [16]. We also compare our results with an MS application using the all-pairs neighborhood and with a dispatching rule, which combines the earliest due date (EDD) with an *insertion*

procedure. In fact, as in Ref. [27], we use the MS algorithm as a benchmark to make comparisons. We also report computational experiments in order to evaluate the relative effectiveness of a variety of recombination operators especially tailored for this problem.

2. Generation of instances

We decided to describe the methodology used to generate the test instances in the beginning of the paper, as it is necessary in order to understand some partial results and parameter settings shown in the next sections. Instances were generated based on the rules of Tan and Narasimhan [27] and Lee et al. [16]. The size of the instances varies between 20 and 100 jobs. The generation of processing times and setup times follows a discrete uniform distribution: DU $(0, 100)$. Due dates are generated according to two parameters: *due date range* and *due date mean*. The due date range is defined according to a due date factor, R, and is described by

$$\Delta d = Rnp_{\mathrm{m}},$$

where p_{m} is the mean processing time.

The due dates mean is defined by the *tardiness factor* τ, which measures the percentage of jobs that are not expected to accomplish their due dates. The equation for the due dates mean is

$$d_{\mathrm{m}} = (1 - \tau)np_{\mathrm{m}}.$$

We chose three values for τ and R: 0.2, 0.6 and 1.0. By combining these values, we obtain nine different configurations of these parameters. Rubin and Ragatz [26] show that when using their B&B algorithm, an instance with the due dates generated over a wide range (a large value of R) is easier to solve than one with the due dates concentrated in a narrow range. Similarly, when the mean of the due dates lies near the beginning or near the end of the schedule the problem is easier to solve for their B&B. If the mean lies near the middle of the schedule, the problem becomes harder.

3. Comparison heuristics – MS, EDD insertion and ATCS

In order to compare our population approaches, we choose three other methods: MS, EDD insertion ($\mathrm{EDD}_{\mathrm{ins}}$) and ATCS. The MS algorithm implemented in this work consists of iteratively generating an initial random solution followed by a hill climbing procedure that uses an all-pairs neighborhood. Therefore, the solution found is a local minimum. The neighbors are obtained by systematically swapping all pairs of jobs in the given sequence. The process is iterated many times until a stop criterion (a time limit criterion has been used) is satisfied and then the best solution ever found is reported. It is one of the simplest methods and uses almost no adaptive procedure at all, with the exception of a local search scheme. Despite this fact, MS implementations have shown good performances for sequencing problems and are often used as benchmarks.

The $\mathrm{EDD}_{\mathrm{ins}}$ is a dispatching rule often used in problems with due dates. The version implemented consists, in a first phase, of sequencing the jobs according to the EDD rule, i.e., in a non-decreasing order of the due dates. In the second phase, the schedule is constructed by inserting each job of the EDD sequence in all possible positions of the partial schedule, choosing the one that leads to the smaller total tardiness. It is a simple constructive and greedy procedure similar to the NEH algorithm first proposed for the flow-shop problem [22]. The third phase consists of a local search procedure based on both all-pairs and insertion neighborhoods. In an insertion move, a job of the given sequence is systematically tested for insertion between any two jobs of the sequence. The move is performed if it reduces the tardiness.

Lee et al. [16] proposed the dispatching rule ATCS as an extension of Raman's rule [25]. It takes into account the means and dispersions of the due dates, setup times and processing times. It consists of three distinct phases. The first one is the calculation of three parameters that rely on certain characteristics of the instance such as size, due dates, setup times and processing times. In the second phase, a schedule is generated based on the parameters calculated in the previous phase.

Finally, in the third phase, a local search procedure based on swap and insertion neighborhoods is applied.

4. GA and MA implementations

GAs received recognition in the mid 1970s, particularly after John Holland's book titled *Adaptation in Natural and Artificial Systems* was published [14]. Since then, due to their simplicity and efficiency, GA became increasingly popular. In the 1980s, a new class of *knowledge-augmented* GAs, sometimes called *hybrid* GAs, started to appear in the computing literature. Recognizing important differences and similarities with other population-based approaches, some of them were categorized as MAs in 1989 [21] and this denomination has become widely accepted [7,20]. Some MAs have been understood as GAs, which *only* use, as parents, configurations that are local minima of a local search algorithm. However, this is not always the right generalization, a representative example being a class of MAs known as *scatter search*, which can be traced back to the year 1977 [10]. Below, we show a pseudo-code representation of a *local search*-based MA.

The initialization phase begins at **initializePopulation** and ends just before the **repeat** command. It is responsible for generating an initial population (*Pop*) of approximate solutions, requiring the creation, optimization and evaluation of each initial individual performed by FirstPop, Local-Search-Engine(·) and EvaluateFitness(·) methods, respectively.

```
procedure Local-Search-based Memetic Algorithm;
  begin
    initializePopulation Pop using FirstPop;
    initializeOffspring offspring;
    forEach individual i ∈ Pop do i:=Local-Search-Engine(i);
    forEach individual i ∈ Pop do EvaluateFitness(i);
    repeat /*generation loop */
      for i:=1 to #recombinations do
        selectToMerge (parent_A, parent_B) ⊆ Pop;
        new_indiv:=Recombine (parent_A, parent_B);
        if (selectToMutate new_indiv) then new_indiv:=Mutate(new_indiv);
        new_indiv:=Local-Search-Engine(new_indiv);
        EvaluateFitness(new_indiv);
        addInOffspring individual new_indiv to offspring;
      endFor;
      addInPopulation offspring offspring to Pop;
    until (termination-condition = True);
  end;
```

The second phase includes the so-called *generation loop*. At each step, two parent configurations (*parent_A*, *parent_B*) are selected for recombination (**selectToMerge**) and an offspring is generated with the Recombine(·,·) method. Then, if it is selected to mutate (**selectToMutate**), a mutation procedure is applied to it. The next steps are the optimization of the individual through a local search procedure, its evaluation and insertion (**addInOffspring**) into the offspring. After all recombinations are done, the offspring is inserted into the population. Finally, a termination condition is checked for true and a new generation begins.

Next, we will survey all the main characteristics of the MAs we have implemented. We will specify the hierarchical population structure, representation, recombination schemes, mutation, selection, fitness function and the inclusion and exclusion of individuals from the population. The effectiveness of any population approach relies on suitable choices of these characteristics as well as on the settings of the parameters associated with them.

4.1. Solution representation

When using a GA or an MA, it is very important to keep in mind that the representation of a solution can greatly influence the effectiveness of the algorithm. When representing a solution as a "chromosome" or "genotype", one should avoid,

for instance, that small changes in the values of the alleles create solutions with values of the fitness function too far from the ones of the original parents. In that case, it is wise to consider other solution representation. Such a sensitive scheme may interfere with the convergence process of the population and the quality of the offspring.

The representation for the SMS we have chosen is quite intuitive with a solution mapped into a chromosome with the alleles assuming different integer values in the $[1, n]$ interval. Any solution can be mapped into this permutation representation and consequently the size of the search space is $n!$ This approach can be found in most GA articles dealing with sequencing problems.

4.2. Crossover

Three different crossover operators were implemented, one of them being hybrid. The first crossover we have implemented is well known as *order crossover* (OX) [11]. After choosing two parents through the **selectToMerge** procedure, a fragment of the chromosome from one of them is copied into the offspring. The selection of the fragment is made uniformly at random. In the second phase, the empty positions are sequentially filled according to the chromosome of the other parent and following the sequence of jobs.

Parent A	1 3 2 [7 8 6 9] 4 5	1 _ _ 7 8 6 9 _ _ (B)
Parent B	1 7 8 4 5 6 2 3 9	1 4 _ 7 8 6 9 _ _ (B)
		1 4 5 7 8 6 9 _ _ (B)
Offspring	_ _ _ 7 8 6 9 _ _ (A)	1 4 5 7 8 6 9 2 _ (B)
	1st phase	1 4 5 7 8 6 9 2 3 (B)
		2nd phase - final offspring

In the above example, the fragment is selected from parent A and consists of jobs 7–8–6–9 at positions 4–7. The child's empty positions are then filled according to the order they appear in the chromosome of parent B. This scheme tends to perpetuate what is called the *relative order* of the jobs.

The second crossover we implemented was proposed by Rubin and Ragatz [26]. It was specifically designed for this problem and since they did not name it, we will refer to it as RRX. The RRX does not only preserve the relative order of the jobs, but also preserves some jobs in their absolute positions in the sequence. It divides the jobs into two sets and mixes them according to the information in both parents, with no random decisions. Following this scheme, exactly eight individuals are generated, two of them being clones of the parents. As a fixed sized population is adopted, it is necessary to select some of the offspring to be incorporated into the population. They made comparisons with several policies and all performed similarly. For this reason, they suggested a random choice of one of the offspring.

Finally, a third recombination strategy consists of both crossovers acting together. This is a hybrid scheme in which half of the recombination in every generation is done according to the OX crossover and the other half according to the RRX crossover.

The number of new individuals created during one generation is related to the *crossover_rate* parameter and is expressed by *pop_size* $*$ *crossover_rate*, where *pop_size* is the number of individuals in the population.

4.3. Mutation

Many researchers believe that mutation does not play an important role in the evolutionary process and that when a good crossover scheme is at hand, no mutation is needed at all. As the crossovers implemented lead to a quick loss of diversity, mutation does play an important role in preserving the diversity of the population. Indeed, for some instances and representations, mutation has its relevance [18]. We implemented a traditional mutation strategy based on job swapping. According to the Mutate(.) function, two positions are randomly selected and the alleles in these positions swap their values. The function is applied to each individual with a probability of *mut_rate* and once applied it mutates only two alleles. More changes per chromosome were tested, but led to

worse results. In fact, when the number of mutated alleles increases, valuable information tends to be lost, worsening the overall performance. The values for the parameter *mut_rate* will be addressed later.

4.4. Local search

Local search algorithms rely on a neighborhood definition that establishes a relationship between solutions in the configuration space. In this work, two neighborhood definitions were chosen and they were both used in the MS and in the MA approaches. One can define the neighborhood of a solution (a job sequence) as the set of solutions that is reachable by executing any possible move allowed by the neighborhood generation scheme. One of the neighborhoods implemented was the *all-pairs*. It consists of all possible swaps of pairs of jobs in a given solution. A *hill-climbing* algorithm can be defined by reference to this neighborhood; i.e., starting with an initial permutation of all jobs, a swap is confirmed every time it causes a reduction in the total tardiness. Other cycle of swaps recurrently takes place, until no further improvement is achieved.

The second neighborhood implemented was the *insertion* neighborhood. It consists of removing a job from one position and inserting it in another one. Jobs are removed and inserted starting from the beginning of the sequence. The hill-climbing procedure is the same we used for the all-pairs scheme.

4.5. Fitness function

The fitness function is a key element to guide the evolutionary process. A function that leads to a reliable evaluation shall prevent the algorithm from discarding good solutions or selecting bad ones for reproduction with a higher probability that would be desirable.

As in our problem the objective is to minimize the total tardiness, the fitness function chosen was

$$f_i = (T_i + 1)^{-1},$$

where T_i is the total tardiness of solution i. The inversion is due to the fact that a large total tardiness means a low fitness and vice versa. The total tardiness was increased by one in order to avoid a division by zero when it equals zero. A similar fitness function was also used in Refs. [6,26].

4.6. Population structure

GAs using structured population approaches are common in implementations in which computational tests are executed in parallel computers. Usually in this case, each processor is responsible for one individual or a subset of them. Results obtained in parallel computers (with structured populations) are in general much better than the ones obtained in sequential machines (which generally employ non-structured populations). Even though our computational implementation and tests have been carried out on a single-processor computer, we decided to implement a hierarchical population structure based on a ternary tree. In contrast with the non-structured version, where all individuals can recombine with each other, the population is divided into *clusters*, restricting crossover possibilities.

The structure consists of several clusters, each one having a *leader* solution and three *supporter* solutions, as shown in Fig. 1. The leader of a cluster is always fitter than its supporters. As a consequence, top clusters tend, on an average, to have fitter individuals than bottom clusters. As new individuals are constantly generated, replacing old ones, periodic adjustments are necessary to keep this structure well ordered.

The number of individuals in the population is restricted to the numbers of nodes in a complete ternary tree, i.e. $(3^k - 1)/2$, where k is the number

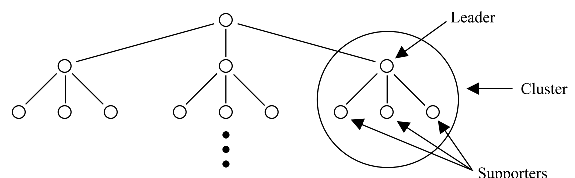


Fig. 1. Population structure.

of levels of the tree. That is, 13 individuals are necessary to construct a ternary tree with three levels, 40 to make one with four levels and so on.

In order to demonstrate the effectiveness of a structured population scheme compared to an non-structured one, we conducted tests using a small set of instances. Before doing this, the parameter settings for both implementations have to be calculated.

4.6.1. Parameter setting: Structured population

Parameter tuning was applied only to the size of the population, crossover rate and mutation rate, considering the following ranges:

Population size	13, 40 or 121 individuals (ternary tree with 3, 4 or 5 levels)
Crossover rate	from 0% to 100% (in 10% steps)
Mutation rate	from 0% to 100% (in 10% steps)

The procedure initially fixes all parameters at specific values within the ranges above. These initial values do not influence the final result of the procedure. So, we vary the population size values within the range and keep the best one, i.e., the value that led to the best total tardiness. After this phase has ended, we fix the size of the population as the best value found and start varying the crossover rate. The process continues this way and finishes when the mutation rate has been fixed. This completes the first cycle of the parameter tuning. The best total tardiness found and the best parameters are saved and another cycle begins, now with the parameters set as the best ones found in the previous cycle. At the end of each cycle, the best total tardiness value is compared to the value of the best total tardiness ever found and, if an improvement is detected, the best parameter values are updated. This strategy showed that after three cycles, there were almost no changes in the best parameter values.

Parameters were tested for GA and MA with OX crossover. The MA used an all-pairs-based local search. Values of τ and R were fixed at 0.6. Tests with other configurations of τ and R revealed no significant variations in the results. The best parameters found are presented in Table 1.

4.6.2. Parameter setting: Non-structured population

Parameter tuning followed the same methodology for the structured population approach, with the range of population size being [10, 150] in 10-individual steps. The best parameters found appear in Table 2.

Table 1
Best parameters found for the structured population

n	Genetic algorithm			Memetic algorithm		
	Population size	Crossover rate	Mutation rate	Population size	Crossover rate	Mutation rate
20	121	0.3	0.8	40	0.2	0.1
40	121	0.4	0.7	40	0.3	0
60	40	0.3	0.6	13	0.2	0.1
80	40	0.5	0.8	13	0.3	0
100	13	0.5	0.8	13	0.2	0

Table 2
Best parameters found for the non-structured population

n	Genetic algorithm			Memetic algorithm		
	Population size	Crossover rate	Mutation rate	Population size	Crossover rate	Mutation rate
20	100	0.5	0.7	80	0.3	0.1
40	70	0.4	0.8	60	0.5	0.1
60	70	0.4	0.9	50	0.4	0.0
80	50	0.5	0.8	40	0.3	0.0
100	30	0.6	0.7	20	0.3	0.1

Based on the results shown in Tables 1 and 2, there does not seem to be a correlation between parameter values and problem size (possibly due to the fixed step size of 10% chosen for the variations) except for the size of population. This can be explained by the total tardiness criterion used to determine the parameter settings. In each category we had *several configurations* with almost the same total tardiness. Therefore, one should not take these parameters as absolute values, but just as *guiding* values that should be used in the future for other comparisons. The most important results displayed by these tests are the importance of mutation and the large population size found for both GAs. One should not be puzzled by the 70–80–90% values obtained for the GA mutation rates as they are only related to the percentage of new *individuals* that are mutated. As scheduling problems represent individuals as a permutation vector, mutation must be associated with the exchange of job positions, for instance, the two jobs involved in a swap move. Usually mutation rates range from 0% to 5% at maximum, but this percentage is generally related to the percentage of *jobs* (or *genes*) that are mutated. On an instance of size $n = 100$, a 1.6% mutation rate applied to jobs would be equivalent to a 80% mutation rate applied to *individuals*. One can interpret these high GA mutation rates as an indication that GAs, when submitted to a parameter optimization procedure, tend to MAs, assuming that systematic and high mutation rates keep a close relation with a local search scheme.

With respect to the MAs, the conclusion we draw is that mutation should be set at very low rates. Crossover rates are almost the same for all approaches, with a slightly higher rate for the GA and for the non-structured population. The number of individuals decreases with the number of jobs mainly because, in a limited processing time environment, it is better to have few individuals and make more generations than to have more individuals and few generations.

4.6.3. Structured population versus non-structured population

In this section we present results for the structured and the non-structured versions of the MA. Here, we try to show the importance of a hierarchical population structure, since very few papers address this issue, with the possible exception of Refs. [2,23]. Next, we present a table comparing the results obtained by both approaches when compared to the ones obtained by the MS algorithm used as a benchmark. All methods use the same all-pairs-based local search and the same computational time: 2 minutes. Numbers are percentage improvement over the MS strategy. Values of τ and R were fixed at 0.6.

Each entry in Table 3 represents the average value for five different instances. The results indicate that the performance increases when population structure is used. For small instances, the values are similar for both structured and non-structured populations, but for instances with 80 jobs and especially 100 jobs, the gap is quite relevant. These initial computational experiments show a clear performance superiority of the MA over the GA. Notice that the GA and MA implemented shared the same characteristics, the only difference being the Local-Search-Engine($\cdot$) method that is switched off in the GA.

Table 3
Comparison between the non-structured and the structured populations

n	Genetic algorithm		Memetic algorithm	
	Non-structured population	Structured population	Non-structured population	Structured population
20	−2.3	−2.7	*1.9*	1.7
40	−2.8	−3.0	13.2	*13.4*
60	−9.2	−7.0	12.4	*14.1*
80	−11.2	−6.3	10.1	*12.6*
100	−17.0	−8.6	6.1	*11.9*
Mean	−8.5	−5.5	8.7	*10.7*

4.7. Neighborhood reductions

Given the large size of the all-pairs and insertion neighborhoods, and the computational complexity required for calculating the total tardiness for each solution, the implementation of reduction schemes is mandatory. The search within a given neighborhood requires the computation of all solutions that belong to it. In our problem, the analysis of each solution obtained through a swap or an insertion move is quite hard. For other sequencing problems, like TSP for instance, the calculation of the objective function's variation (for k-opt procedures, for example) can be done in constant time. All that is necessary is to remove the old edges and add the new ones. However, when the objective function is the total tardiness, any change in the sequence of jobs may affect the tardiness of all jobs that are placed after the ones that have been moved. Therefore, each new evaluation requires a very time-consuming loop to recalculate the total tardiness.

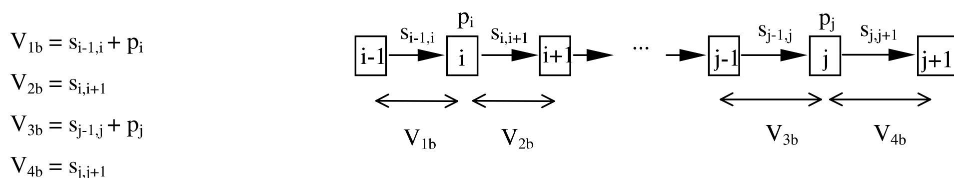
Reduced neighborhoods can be obtained from rules that specify which moves might be worth testing or not. Thus, the number of those recalculation loops may decrease dramatically, depending on how restrictive the reduced scheme is. A well-designed reduction scheme should reduce the number of evaluations as much as possible without loosing the strength of the search. That is, only movements that may return good results should belong to the reduced neighborhood and therefore deserve to be evaluated. Analogously, bad moves must be promptly identified and discarded. Some authors refer to these reduction schemes as *candidate list strategies*.

To reach this goal, we based our reduction rules on the setup times' values. We have observed that most good schedules have a common characteristic: the setup times between jobs in these solutions are very small. That is reasonable, since schedules with small setup times between jobs will be less lengthy, generating few delays.

4.7.1. Neighborhood reductions for the all-pairs and insertion schemes

We designed four reduction rules for the all-pairs neighborhood and two rules for the insertion neighborhood. They are based on comparisons of the setup times and processing times involved in the move. Consider a swap move as shown in Fig. 2.

Configuration before swap:



Configuration after swap:

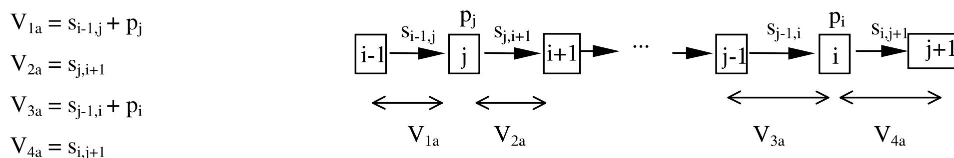


Fig. 2. Swap move.

Table 4
Reduction rules for swap moves

Reduction	Rules
A1	$(V_{1a} < V_{1b}) \vee (V_{2a} < V_{2b}) \vee (V_{3a} < V_{3b}) \vee (V_{4a} < V_{4b})$
A2	$[(V_{1a} < V_{1b}) \vee (V_{2a} < V_{2b})] \wedge [(V_{3a} < V_{3b}) \vee (V_{4a} < V_{4b})]$
A3	$[(V_{1a} < V_{1b}) \wedge (V_{2a} < V_{2b})] \vee [(V_{3a} < V_{3b}) \wedge (V_{4a} < V_{4b})]$
A4	$(V_{1a} + V_{2a} < V_{1b} + V_{2b}) \vee (V_{3a} + V_{4a} < V_{3b} + V_{4b})$

Our first rule A1 – see Table 4 – establishes that a swap move will be performed (and the total tardiness recalculated) only if at least one of the new setup times, jointly with the processing times, introduced in the new sequence is smaller than the old ones. The rules A2 and A3 are similar to A1, but more restrictive. The rule A4 is based on the sum of the setup times and the processing times involved in the move before and after it.

The rules for the insertion neighborhood are also based on the exchange of the setup times and processing times, as shown in Fig. 3 and Table 5. The first rule I1 – see Table 5 – establishes that an insertion move will be evaluated only if one verifies that the deletion of job i as well as its insertion behind job j are advantageous. Rule I2 is less restrictive, selecting candidates if one of these conditions holds.

Table 5
Reduction rules for the insertion moves

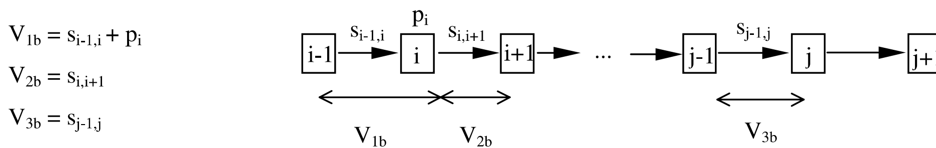
Reduction	Rules
I1	$(V_{1a} < V_{1b} + V_{2b}) \wedge (V_{2a} + V_{3a} < V_{3b})$
I2	$(V_{1a} < V_{1b} + V_{2b}) \vee (V_{2a} + V_{3a} < V_{3b})$

4.7.2. Computational evaluation

Table 6 shows the results attained by the six neighborhood reductions as well as a composite neighborhood that used the best schemes for swap and insertion. Figures are percentage deviation obtained by each rule with respect to MS using an all-pairs neighborhood, all of them running for the same amount of time (2 minutes).

The mean is calculated for the following values of (τ, R): (0.6, 0.2); (0.6, 0.6); (0.6, 1.0); (1.0, 0.2); (1.0, 0.6); (1.0, 1.0). We did not consider $\tau = 0.2$ because of the instability verified in this case. The MS often obtains values near zero when $\tau = 0.2$, and thus any deviation returns high percentage values. This instability could easily invalidate the results. The row labeled *Size* indicates the fraction of the size of the reduced neighborhood compared to the full-sized neighborhood.

Configuration before insertion:



Configuration after insertion:

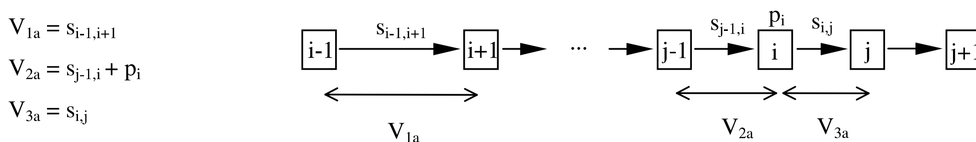


Fig. 3. Insertion move.

Table 6
Neighborhood reduction evaluation

% deviation of MA/OX/Structured population with respect to MS/all-pairs local search									
n	All-pairs	A1	A2	A3	A4	Insertion	I1	I2	A2/I2
20	0.8	0.9	0.6	0.4	*1.0*	0.4	−1.1	*0.7*	0.9
40	10.2	*10.3*	10.0	8.2	9.3	6.0	−5.1	*6.3*	11.3
60	9.4	10.4	*10.8*	9.3	10.5	4.6	−10.4	*6.5*	11.9
80	7.7	8.6	*10.0*	9.4	9.5	2.0	−17.7	*4.6*	11.1
100	5.9	6.0	8.0	7.2	*8.2*	−0.4	−23.0	*2.1*	10.7
Mean	6.8	7.2	*7.9*	6.9	7.7	2.5	−11.5	*4.0*	*9.2*
Size	*	0.51	0.05	0.05	0.21	*	<0.01	0.50	0.24

A very small influence of τ and R in the neighborhood reduction performance was observed. All configurations led to quite similar results. This can be explained by the fact that the reduction is based on the setup time values, which do not depend on τ and R.

The reduction schemes have been shown to improve the MA, especially for medium-sized instances. The rules A1, A2, A3 and A4 performed better than the full size neighborhood, although their sizes vary only from 5% up to 51% of the full scheme. With the insertion neighborhood, we noticed a different behavior. The reduction scheme I1 had a very poor performance – probably because of its extremely reduced size – while the I2 performed better. The hybrid including the best two rules (A2/I2) attained the best performance. According to it, each individual is submitted to an A2-based local search and the resulting sequence goes through an I2-based local search. The final solution is then reported. The size of this hybrid neighborhood is about a fourth of the all-pairs and insertion neighborhoods together, meaning that almost four times more generations are executed in the same amount of time. The computational time necessary for crossover does not change with the reduction, requiring only a small fraction of the total CPU time.

4.8. Selection to recombination

4.8.1. Non-structured population

The **selectToMerge** procedure followed the *roulette wheel* scheme [11]. Consider the total fitness of the population as being the sum of the fitness of all individuals. Dividing the individual fitness by this total fitness, one will obtain the fraction of the population's fitness that this individual has contributed to. For example, consider three individuals, with fitness 10, 15 and 30. The total fitness is 55 and the fraction of each individual is 0.18, 0.27 and 0.55, respectively. Therefore, the interval for each individual in the roulette wheel is $[0, 0.18]$, $[0.18, 0.45]$ and $[0.45, 1.0]$.

A random number between 0 and 1 is generated and the individual in whose interval the number lies is selected. Initial tests showed that the best individual among the population was participating in almost all crossovers, especially because we were keeping the best individual ever found always present in the gene pool. This is a common problem with roulette schemes. Therefore, we added a selection restriction to force diversity.

Each individual has an associated variable *recombine*, which can assume values *true* or *false*. If it is *true*, then the individual *may* be selected for recombination, otherwise it is forbidden. For the best individual, before each recombination, this variable is set *true* with a probability of 0.5, which means it may take part in 50% of the crossovers at maximum. This limited pressure on the best individual is worth doing, improving slightly but constantly the results, when compared to the ordinary roulette wheel scheme.

For all other individuals, the variable *recombine* is set to *true* at the start of the recombination loop and each time a non-incumbent individual is selected, its variable is set *false*. Finally, if more than 80% of the population is forbidden to recombine and the generation loop has not ended yet, all *recombine* variables are set to *true*.

4.8.2. Structured population

Recombination in the structured population can only be made between a leader and one of its supporters within the same cluster. The **selectToMerge** procedure always selects a leader – and consequently the cluster – without using the roulette wheel or any other biased selection scheme. Then, it chooses (uniformly at random) one of the three supporters present in the same cluster of the selected leader. No restriction to the number of recombinations an individual can take part in was implemented. The only intensification procedure is that the best individual must take part in approximately 10% of the recombinations, that is, 10% of the crossovers occur with individuals from the uppermost cluster.

4.9. Offspring insertion into population

4.9.1. Non-structured population

We used an intermediate population approach with a deterministic insertion policy. All individuals created during the crossover and mutation processes are saved as an intermediate population and only at the end of the recombination loop they are inserted into *Pop*. The **addInPopulation** function replaces the less adapted individuals according to the non-decreasing sequence of fitness.

A diversity control mechanism was also incorporated in the **addInPopulation**. It consists of copying individuals from the intermediate population to *Pop* only if they had a total tardiness value that is different from the ones associated with all other individuals in *Pop*. This prevents the population from having duplicated individuals, greatly reducing the diversity loss.

At the end of the insertion process, the intermediate population is erased. This policy can be classified as *elitist* since it always preserves the incumbent solution in the gene pool.

4.9.2. Structured population

According to the **addInPopulation** function for the structured population, if the fitness of an offspring is better than the leader selected by **selectToMerge**, the new individual substitutes it. Otherwise, it takes the place of the supporter that took part in the recombination. If the new individual is already present in the population, then it is not inserted as we adopted a policy of not allowing duplicated individuals to reduce loss of diversity. After the offspring has been inserted, the population is restructured. Since the fitness of any individual must be greater or equal to the fitness of all the individuals which form the subtree below it, the upper subgroups will have individuals with better fitness than the lower groups and the best solution will be the leader of the uppermost subgroup. The adjustment is done by comparing the leader of each subgroup with its supporters. If any supporter turns out to be better, then they swap places. Any simple tree-ordering algorithm can execute this job.

5. Computational experiments

As pointed out in Section 3, our GA and MA approaches for the SMS problem have also been compared with three other heuristics – MS, EDD_{ins} and ATCS. For each implementation of the GAs and MAs, we tested three different crossovers as described in Section 4.2. The population is structured for all the genetic and memetic approaches. The genetic RRX approach is not the same as Rubin and Ragatz [26]. We use just the RRX crossover proposed by them embedded in a different GA/MA framework developed by us. At the end of this section, we report tests comparing our MA with their original GA/RRX. We also used a fixed CPU time of 2 minutes per algorithm, except for the ATCS and the EDD_{ins}, which required always less than 2 seconds. All methods were coded in Sun Java JDK 1.2 and executed using a PC-compatible PENTIUM II-266 MHz with 128 Mb RAM.

As a benchmark, we took the results of the MS with all-pairs local search. The other approaches were compared to it and the Tables 7–15 show the mean relative performance of all strategies for a set of five runs each. This relative performance is calculated as

$$\mathrm{RP}_{\mathrm{alg}} = \left(1 - \frac{T_{\mathrm{alg}}}{T_{\mathrm{MS}}}\right) \times 100,$$

Table 7
$(\tau, R) = (0.2, 0.2)$

n	Genetic algorithm			Memetic algorithm			ATCS	EDD_{ins}
	RRX	OX	RRX/OX	RRX	OX	RRX/OX		
20	−28.3	4.8	5.0	3.9	*8.6*	6.7	−80.0	−110.0
40	−20.5	12.1	−3.6	5.4	*32.4*	23.9	−23.8	−33.4
60	−62.3	3.1	−7.2	21.9	*42.9*	36.4	−10.5	−14.5
80	−70.1	−0.5	−15.4	18.9	*38.2*	30.3	−3.0	−1.5
100	−112.1	−10.2	−32.7	19.9	*42.0*	40.0	3.3	5.7
Mean	−58.6	1.9	−10.8	14.0	*32.8*	27.5	−22.8	−31.7

Table 8
$(\tau, R) = (0.2, 0.6)$

n	Genetic algorithm			Memetic algorithm			ATCS	EDD_{ins}
	RRX	OX	RRX/OX	RRX	OX	RRX/OX		
20	−45.2	2.4	−22.0	13.0	*13.2*	13.2	−112.1	−307.9
40	−37.8	3.3	6.8	26.6	*54.2*	50.5	−70.2	−76.2
60	−60.4	−14.9	−20.7	35.7	*86.0*	78.6	−40.6	−67.7
80	−92.6	−22.2	−74.4	36.4	*82.5*	74.7	−15.9	−15.1
100	−123.7	−36.5	−98.8	23.1	*60.2*	54.6	−9.8	0.3
Mean	−71.9	−13.6	−41.8	26.9	*59.2*	54.3	−49.7	−93.3

Table 9
$(\tau, R) = (0.2, 1.0)$

n	Genetic algorithm			Memetic algorithm			ATCS	EDD_{ins}
	RRX	OX	RRX/OX	RRX	OX	RRX/OX		
20	−42.1	−23.0	−30.2	9.9	*12.6*	10.0	−95.1	−251.9
40	−88.9	−10.1	−44.2	16.0	*67.5*	62.6	−111.2	−181.1
60	−143.7	−47.7	−66.1	65.8	*100.0*	96.1	−137.6	−208.9
80	−168.1	−119.2	−140.8	45.6	*88.7*	86.6	−25.2	−100.1
100	−203.4	−170.5	−196.3	52.4	*86.7*	80.7	10.2	−65.0
Mean	−129.2	−74.1	−95.5	37.9	*71.1*	67.2	−71.8	−161.4

Table 10
$(\tau, R) = (0.6, 0.2)$

n	Genetic algorithm			Memetic algorithm			ATCS	EDD_{ins}
	RRX	OX	RRX/OX	RRX	OX	RRX/OX		
20	−8.8	−3.9	−3.6	1.7	*1.7*	1.7	−20.5	−40.4
40	−11.2	−5.4	−2.0	3.2	*16.2*	14.1	−13.1	−15.5
60	−16.7	−3.2	−9.6	5.7	*14.7*	13.1	−1.1	−3.8
80	−23.6	−1.9	−10.7	4.0	*12.8*	9.4	4.2	2.4
100	−27.2	−3.3	−16.1	5.3	*13.1*	11.0	5.3	3.2
Mean	−17.5	−3.5	−8.4	4.0	*11.7*	10.4	−5.0	−10.8

Table 11
$(\tau, R) = (0.6, 0.6)$

n	Genetic algorithm			Memetic algorithm			ATCS	EDD_{ins}
	RRX	OX	RRX/OX	RRX	OX	RRX/OX		
20	−10.1	−2.7	−3.3	1.2	*2.1*	1.8	−29.2	−20.3
40	−15.4	−3.0	−5.4	3.4	*18.5*	15.6	−9.8	−9.9
60	−16.6	−7.0	−7.2	4.3	*15.6*	15.0	−4.2	−4.5
80	−18.7	−6.3	−12.5	5.8	*16.2*	15.8	2.3	2.7
100	−22.6	−8.6	−17.5	5.3	*15.5*	12.2	5.3	4.9
Mean	−16.7	−5.5	−9.2	4.0	*13.6*	12.1	−7.1	−5.4

Table 12
$(\tau, R) = (0.6, 1.0)$

n	Genetic algorithm			Memetic algorithm			ATCS	EDD_{ins}
	RRX	OX	RRX/OX	RRX	OX	RRX/OX		
20	−8.9	1.0	−2.2	*0.5*	*0.5*	*0.5*	−23.9	−32.1
40	−19.7	−3.6	−4.7	3.5	*15.5*	*15.5*	−14.7	−11.9
60	−18.7	−3.7	−8.8	4.1	*16.7*	15.6	−2.7	−6.7
80	−30.3	−8.9	−10.3	3.8	*18.4*	14.0	−3.4	−4.1
100	−39.3	−12.9	−27.3	2.7	*17.5*	11.0	1.0	1.2
Mean	−23.3	−5.6	−10.6	2.9	*13.7*	11.3	−8.7	−10.7

Table 13
$(\tau, R) = (1.0, 0.2)$

n	Genetic algorithm			Memetic algorithm			ATCS	EDD_{ins}
	RRX	OX	RRX/OX	RRX	OX	RRX/OX		
20	−2.5	−1.3	−3.2	*0.2*	*0.2*	*0.2*	−8.3	−15.9
40	−10.7	−2.9	−4.8	1.7	*6.2*	5.9	−3.5	−8.1
60	−12.4	−2.0	−3.1	2.0	*7.3*	6.4	3.3	−1.4
80	−15.2	−4.2	−5.4	1.4	*6.8*	5.0	4.1	−0.8
100	−16.4	−3.9	−9.8	1.7	*6.2*	5.2	4.7	1.0
Mean	−11.4	−2.9	−5.2	1.4	*5.3*	4.5	−0.5	−5.0

Table 14
$(\tau, R) = (1.0, 0.6)$

n	Genetic algorithm			Memetic algorithm			ATCS	EDD_{ins}
	RRX	OX	RRX/OX	RRX	OX	RRX/OX		
20	−6.3	−1.3	−2.1	*0.7*	*0.7*	*0.7*	−9.2	−12.8
40	−7.5	−3.1	−0.9	2.1	4.7	*4.8*	−4.4	−6.7
60	−7.6	−3.5	−3.3	2.5	*7.9*	6.8	1.2	−2.0
80	−10.6	−3.2	−5.6	1.3	*6.5*	5.4	2.6	−2.8
100	−12.7	−5.7	−9.4	1.5	*6.2*	3.9	3.8	1.2
Mean	−8.9	−3.3	−4.3	1.6	*5.2*	4.3	−1.2	−4.6

Table 15
$(\tau, R) = (1.0, 1.0)$

n	Genetic algorithm			Memetic algorithm			ATCS	EDD_{ins}
	RRX	OX	RRX/OX	RRX	OX	RRX/OX		
20	−4.2	−2.5	−2.9	0.1	*0.3*	*0.3*	−4.7	−16.2
40	−5.2	−2.9	−2.2	2.6	*6.7*	6.1	−3.5	−2.5
60	−7.8	−2.1	−3.3	2.5	*9.1*	7.1	−0.1	−1.3
80	−9.6	−3.1	−6.1	1.8	*6.0*	4.3	1.5	0.3
100	−12.5	−3.7	−8.3	3.1	*5.9*	4.8	1.8	−0.1
Mean	−7.8	−2.9	−4.5	2.2	*5.6*	4.7	−0.2	−3.2

where RP_{alg} is the relative performance of the method in comparison with the MS, T_{alg} the total tardiness obtained by the method being analyzed and the T_{MS} is the total tardiness obtained by MS using the all-pairs neighborhood. The neighborhood chosen for the MAs was A2/I2. Each table refers to a combination of (τ, R), considering the values 0.2, 0.6 and 1.0.

Notation

RRX crossover strategy proposed by Rubin and Ragatz [26]
OX order crossover strategy
ATCS dispatching rule proposed by Lee et al. [16]

The 100% improvement indicates that the MA found a total tardiness equal to zero in all runs while the MS did not find any.

Analyzing the results obtained by the GAs and the MAs, we can conclude that OX is the best crossover operator in this study. The RRX crossover performed poorly, probably because of the quick loss of diversity. The MS outperformed GA with OX in all but few cases, although with a small gap, rarely greater than 10%. On the other hand, the MA always outperforms the MS. For $\tau = 0.2$, the improvements were extremely high. That is because in these cases, the optimal total tardiness equals zero or lies near zero. Therefore, any deviation will return very high improvements. The ATCS dispatching rule performed well, showing better results than EDD_{ins} in all but one data set. Its performance was shown to be better as the number of jobs increases. For more than 80 jobs the ATCS rule can rival MS and be better than the GA in almost all parameter configurations. Table 16 summarizes the mean deviation observed for each combination of τ and R.

In an overall evaluation of the various methods implemented, we can say that the MAs appear as the best ones (OX crossover is preferable), fol-

Table 16
Mean deviation

(τ, R)	Genetic algorithm			Memetic algorithm			ATCS	EDD_{ins}
	RRX	OX	RRX/OX	RRX	OX	RRX/OX		
(0.2, 0.2)	−58.6	1.9	−10.8	14.0	*32.8*	27.5	−22.8	−31.7
(0.2, 0.6)	−71.9	−13.6	−41.8	26.9	*59.2*	54.3	−49.7	−93.3
(0.2, 1.0)	−129.2	−74.1	−95.5	37.9	*71.1*	67.2	−71.8	−161.4
(0.6, 0.2)	−17.5	−3.5	−8.4	4.0	*11.7*	10.4	−5.0	−10.8
(0.6, 0.6)	−16.7	−5.5	−9.2	4.0	*13.6*	12.1	−7.1	−5.4
(0.6, 1.0)	−23.3	−5.6	−10.6	2.9	*13.7*	11.3	−8.7	−10.7
(1.0, 0.2)	−11.4	−2.9	−5.2	1.4	*5.3*	4.5	−0.5	−5.0
(1.0, 0.6)	−8.9	−3.3	−4.3	1.6	*5.2*	4.3	−1.2	−4.6
(1.0, 1.0)	−7.8	−2.9	−4.5	2.2	*5.6*	4.7	−0.2	−3.2
Mean	−31.8	−12.1	−21.1	10.5	*24.2*	21.8	−18.6	−32.7

lowed by the GA with OX operator. ATCS is the next method, followed by the other GAs. The worst performance was obtained by the EDD_{ins} rule.

5.1. Tests with the instances of Rubin and Ragatz

Table 17 shows the computational results obtained on the problems used in the article of Rubin and Ragatz [26], where the RRX crossover was proposed. The instances vary from 15 to 45 jobs and the results include our GA and MA with OX operator, the ATCS rule and the genetic algorithm presented in Ref. [26] – column labeled "Genetic RRX". In their work, they used upper bounds and optimal values as comparison benchmarks – optimal values for small problems found by a branch-and-bound method [24] and upper bounds provided by its current incumbent solution value obtained after two million nodes for the larger ones. Two stopping criteria based on CPU time were implemented for GA and MA: 2 minutes of CPU time for each run (GA_{2m} and MA_{2m}) and the approximate equivalent CPU time used for Genetic RRX (GA_{RR} and MA_{RR}).

Table 17
Comparisons with Rubin and Ragatz's problems

n	MA_{2m}	MA_{RR}	GA_{2m}	GA_{RR}	ATCS	Genetic RRX
15	0	0	−4.9	−11.1	−74.4	−11.1
	0	0	0	0	0	0
	0	0	−0.6	−1.1	−2.4	−0.6
	0	0	0	0	−4.1	0
	0	0	0	0	0	0
	0	0	0	0	0	0
	0	0	−0.1	−0.2	−4.5	−0.7
	0	0	0	−0.5	−5.8	−0.3
25	1.8	1.7	−0.8	−2.5	−6.4	−3.8
	0	0	0	0	0	0
	0.4	0.3	−0.2	−0.3	−0.6	−0.1
	0	0	0	0	0	0
	0	0	0	0	0	0
	0	0	0	0	0	0
	0	0	−1.3	−2.2	−4.5	−1.8
	17.7	16.8	17.0	1.1	9.1	5.3
35	70.0	61.7	11.0	9.7	−61.7	−2.1
	0	0	0	0	0	0
	4.0	3.7	2.2	0.5	1.0	0.4
	1.5	1.4	−0.7	−1.0	−4.0	−0.1
	18.3	16.4	−5.8	−11.0	−27.1	−10.9
	0	0	0	0	0	0
	2.2	2.2	−0.9	−0.3	−3.0	−0.7
	33.1	31.3	29.6	21.0	29.0	21.1
45	9.0	8.5	−47.0	−55.5	−69.2	−52.4
	0	0	0	0	0	0
	1.6	1.4	−1.2	−1.2	−1.3	−0.8
	5.0	3.6	0.8	−0.3	1.5	0.4
	4.1	0.7	−42.9	−51.7	−82.1	−41.1
	0	0	0	0	0	0
	5.9	5.0	3.9	0.3	−0.2	0.7
	6.0	4.7	3.7	−1.1	−0.3	0.9
Average	5.6	5.0	−1.2	−3.3	−9.7	−3.0

The equivalent time was calculated taking into consideration the different circumstances between Rubin and Ragatz's experiment and ours, respectively,

- processor: Intel 486/33 MHz and Intel Pentium II/266 MHz,
- language: WATFOR-77 and Java.

In our experiment, the equivalent CPU time used was taken as 10% of the CPU time reported by Rubin and Ragatz for the Genetic RRX.

The figures in Table 17 correspond to the average relative performance calculated as

$$RP_{alg} = \left(1 - \frac{T_{alg}}{T_{B\&B}}\right) \times 100,$$

where RP_{alg} is the average relative performance of the algorithm, T_{alg} the total tardiness obtained by the algorithm under consideration and the $T_{B\&B}$ is the total tardiness obtained by the B&B method (optimum or upper bound). The values were obtained over five runs, except the ones of the Genetic RRX extracted from Ref. [26], which correspond to averages calculated over 20 runs.

The results indicate that both MAs performed much better than all GAs and ATCS. This behavior confirms the foregoing using the random instances we generated for the previous set of experiments. The original Rubin and Ragatz's results are a little better than our genetic OX ones but it is worth remarking that their algorithm uses a local search embedded in the mutation procedure, while ours does not. An interesting observation is that almost the same results can be obtained when one executes the MA during 2 minutes or a very small fraction of this time. This is because of the small dimension of this instance set. For larger instances, the MA needs more time to reach good solutions.

6. Summary and conclusions

This paper shows the performance of an MA when compared to other heuristic approaches for the total tardiness SMS problem with sequence-dependent setup times. The MA returned the best results of all, even when the crossover introduced by Rubin and Ragatz (RRX) was used. We also noticed that the RRX performed significantly worse than the already well-known order crossover (OX). The later turned to be the best recombination operator for this sequencing problem in our study. The performance of the RRX crossover was rather disappointing, given that it had been especially crafted by Rubin and Ragatz [26] for this particular problem. We tested the algorithms with several random instances from 20 to 100 jobs and with 9 configurations for the due dates averages and generation interval. In a complementary computational experiment, aimed to cross-validate the previous findings, the instances used in Ref. [26] were solved with our algorithms and the results compared against the ones obtained by the original GA with the RRX crossover. The MA outperformed the other methods also in this case.

Two of the strongest features of the MA implementation proposed in this paper are the neighborhood reductions and the hierarchically structured population. Computational tests using a non-structured population and less elaborated neighborhoods, such as the *full all-pairs* neighborhood, have led to a considerable loss of performance, especially for large instances.

The ATCS dispatching rule had an equivalent behavior compared to GA and MA with RRX in almost all instances with 80 and 100 jobs. For smaller instances, the ATCS dramatically loses performance and cannot compete with evolutionary approaches. It is important to keep in mind that ATCS has been specially tailored for the SMS problem and is rather complicated to implement, while EDD_{ins} is a simple adaptation of an NEH-type algorithm [22].

During the design process of the MA, another conclusion could be drawn. The good performance of the MA results from intensive set of tests with several tentative approaches including selection of neighborhoods and parameter settings, as well as specific parent selection strategies and best-individual intensification. All these aspects, when simultaneously considered had a fundamental role in the performance, since the comparison has been made against elaborated techniques. If any of

these considerations had been set aside, MA results would have been similar or even worse than those of ATCS and MS, especially for instances with 80 or more jobs and $\tau = 1.0$.

We also believe that the MA performance can be considerably increased if new components are added into the current framework. Therefore, further work on new population structures, improved neighborhoods and other aspects might lead to even better results.

Acknowledgements

This work was supported by the Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) and Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP), Brazil.

References

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

[2] R. Berretta, P. Moscato, The number partitioning problem: An open challenge for evolutionary computation?, in: D. Corne, M. Dorigo, F. Glover (Eds.), in: New Ideas in Optimization, McGraw-Hill, New York, 1999, pp. 261–278.

[3] E.K. Burke, J.P. Newall, R.F. Weare, A memetic algorithm for university exam timetabling, Lecture Notes in Computer Science 1153 (1996) 241–250.

[4] E.K. Burke, J.P. Newall, A multi-stage evolutionary algorithm for the timetable problem, IEEE Transactions on Evolutionary Computation 3 (1) (1999) 63–74.

[5] E.K. Burke, A.J. Smith, Hybrid Evolutionary Techniques for the Maintenance Scheduling Problem, IEEE Power Engineering Society Transactions, to appear.

[6] R. Cheng, M. Gen, Parallel machine scheduling problems using memetic algorithms, Computers and Industrial Engineering 33 (3-4) (1997) 761–764.

[7] D. Corne, F. Glover, M. Dorigo, in: New Ideas in Optimization, McGraw-Hill, New York, 1999.

[8] J. Du, J.Y.T. Leung, Minimizing total tardiness on one machine is NP-hard, Mathematics of Operations Research 15 (1990) 483–495.

[9] M. Gen, R. Cheng, in: Genetic Algorithms and Engineering Design, Wiley, New York, 1997.

[10] F. Glover, Scatter search and star-paths – beyond the genetic metaphor, OR Spektrum 17 (2–3) (1995) 125–137.

[11] D.E. Goldberg, in: Genetic Algorithms in Search, Optimization and Machine Learning, Addison-Wesley, Reading, MA, 1989.

[12] R.L. Graham, E.L. Lawler, J.K. Lenstra, A.H.G. Rinooy Kan, Optimization and approximation in deterministic sequencing and scheduling: A survey, Annals of Discrete Mathematics 5 (1979) 287–326.

[13] S.C. Graves, A review of production scheduling, Operations Research 29 (1981) 646–675.

[14] J. Holland, Adaptation in Natural and Artificial Systems, The University of Michigan Press, Ann Arbor, MI, 1975.

[15] C. Koulamas, The total tardiness problem: review and extensions, Operations Research 42 (6) (1994) 1025–1041.

[16] Y.H. Lee, K. Bhaskaran, M. Pinedo, A heuristic to minimize the total weighted tardiness with sequence-dependent setups, IIE Transactions 29 (1997) 45–52.

[17] P. Merz, B. Freisleben, A comparison of memetic algorithms, tabu search, and ant colonies for the quadratic assignment problem, in: Proceedings of the International Congress of Evolutionary Computation (CEC'99), 1999, to appear.

[18] P. Merz, B. Freisleben, Fitness landscapes and memetic algorithm design, in: D. Corne, M. Dorigo, F. Glover (Eds.), New Ideas in Optimization, McGraw-Hill, New York, 1999, pp. 245–260, to appear.

[19] P. Merz, B. Freisleben, Memetic algorithms and the fitness landscape of the graph bi-partitioning problem, in: A.-E. Eiben, T. Bäck, M. Schoenauer, H.-P. Schwefel (Eds.), Proceedings of the Fifth International Conference on Parallel Problem Solving From Nature, Lecture Notes in Computer Science 1498, Springer, Berlin, 1998, pp. 765–774.

[20] P. Moscato, An introduction to population approaches for optimization and hierarchical objective functions: A discussion on the role of Tabu Search, Annals of Operations Research 41 (1–4) (1993) 85–121.

[21] P. Moscato, On evolution, search, optimization, genetic algorithms and martial arts: Towards memetic algorithms, Technical Report, Caltech Concurrent Computation Program, C3P Report 826, 1989.

[22] N. Nawaz, E.E. Enscore, I. Ham, A heuristic algorithm for the m-machine, n-job flowshop sequencing problem, OMEGA – International Journal of Management Science 11 (1) (1983) 91–95.

[23] B. Paechter, A. Cumming, M.G. Norman, H. Luchian, Extensions to a memetic timetabling system, in: E.K. Burke, P. Ross (Eds.), The Practice and Theory of Automated Timetabling, Lecture Notes in Computer Science 1153, Springer, Berlin, 1996, pp. 251–265.

[24] G.L. Ragatz, A branch-and-bound method for minimum tardiness sequencing on a single processor with sequence dependent setup times, in: Proceedings of the 24th Annual Meeting of the Decision Sciences Institute, 1993, pp. 1375–1377.

[25] N. Raman, R.V. Rachamadugu, F.B. Talbot, Real time scheduling of an automated manufacturing center,

European Journal of Operational Research 40 (1989) 222–242.

[26] P.A. Rubin, G.L. Ragatz, Scheduling in a sequence dependent setup environment with genetic search, Computers and Operations Research 22 (1) (1995) 85–99.

[27] K.C. Tan, R. Narasimhan, Minimizing tardiness on a single processor with sequence-dependent setup times: A simulated annealing approach, OMEGA – International Journal of Management Science 25 (6) (1997) 619–634.