Genetic algorithms for single machine scheduling with quadratic earliness and tardiness costs

Jorge M. S. Valente

LIAAD – INESC Porto L.A., Faculdade de Economia, Universidade do Porto, Rua Dr. Roberto Frias, 4200-464 Porto, Portugal

Phone: + 351 225 571 100

Fax: + 351 225 505 050

E-mail: jvalente@fep.up.pt

Maria R. A. Moreira

EDGE, Faculdade de Economia, Universidade do Porto, Rua Dr. Roberto Frias, 4200-464 Porto, Portugal

Alok Singh

Department of Computer and Information Sciences, University of Hyderabad, P.O. Central University, Hyderabad - 500 046, A.P., India

Rui A. F. S. Alves

Faculdade de Economia, Universidade do Porto, Rua Dr. Roberto Frias, 4200-464 Porto, Portugal

Abstract

In this paper, we consider the single machine scheduling problem with quadratic earliness and tardiness costs, and no machine idle time. We propose a genetic approach based on a random key alphabet, and present several algorithms based on this approach. These versions differ on the generation of both the initial population and the individuals added in the migration step, as well as on the use of local search. The proposed procedures are compared with the best existing heuristics, as well as with optimal solutions for the smaller instance sizes.

The computational results show that the proposed algorithms clearly outperform the existing procedures, and are quite close to the optimum. The improvement over the existing heuristics increases with both the difficulty and the size of the instances. The performance of the proposed genetic approach is improved by the initialization of the initial population, the generation of greedy randomized solutions and the addition of the local search procedure. Indeed, the more sophisticated versions can obtain similar or better solutions, and are much faster. The genetic version that incorporates all the considered features is the new heuristic of choice for small and medium size instances.

Keywords *scheduling, single machine, quadratic earliness and tardiness, genetic algorithms*

Introduction

In this paper, we consider the single machine scheduling problem with quadratic earliness and tardiness costs, and no machine idle time. Formally, the problem can be stated as follows. A set of n independent jobs $\{1, 2, \dots, n\}$ has to be scheduled on a single machine that can handle at most one job at a time. The machine is assumed to be continuously available from time zero onwards, and preemptions are not allowed. Job j, $j = 1, 2, \dots, n$, requires a processing time p_j and should ideally be completed on its due date d_j. Also, let h_j and w_j denote the earliness and tardiness penalties of job j, respectively. For a given schedule, the earliness and tardiness of job j are defined as $E_j = \max\{0, d_j - C_j\}$ and $T_j = \max\{0, C_j - d_j\}$, respectively, where C_j is the completion time of job j. The objective is then to find a schedule that minimizes the sum of the weighted quadratic earliness and tardiness costs $\sum_{j=1}^{n} (h_j E_j^2 + w_j T_j^2)$, subject to the constraint that no machine idle time is allowed.

Even though scheduling models with a single processor may appear to arise infrequently in practice, this scheduling environment actually occurs in several activities (for a specific example in the chemical industry, see [1]). Also, the performance of many production systems is quite often dictated by the quality of the schedules for a single bottleneck machine. Moreover, the study of single machine problems provides results and insights that prove valuable for scheduling more complex systems.

Early/tardy scheduling models have received considerable and increasing attention from the scheduling community, due to their practical importance and relevance. In fact, scheduling problems with earliness and tardiness costs are compatible with the concepts of supply chain management and just-in-time production. Indeed, these production strategies, which have been increasingly adopted by many organisations, view both early and tardy deliveries as undesirable.

In this paper, we consider quadratic earliness and tardiness penalties, instead of the more usual linear objective function, in order to penalize more heavily deliveries that are quite early or tardy. On the one hand, this is appropriate

for practical settings where non-conformance with the due dates is increasingly undesirable, and it is in line with the loss function proposed by Taguchi [2].

More specifically, the Taguchi loss function is equal to $L(x) = k\,(x - a)^2$, where $L(x)$ is the loss to society (producer and/or customers) when one unit of a certain output is produced at level x, k is a constant and a is the ideal target level of the output. Thus, this function specifies that the loss increases quadratically as the output deviates from its desired level. In the considered scheduling problem, the jobs' due dates d_j and completion times C_j correspond to the desired target level a and the actual output level x, respectively, while the weights h_j and w_j are a generalization of the constant k. Therefore, the considered objective function is in accordance with Taguchi's loss function.

Moreover, and on the other hand, the quadratic penalties are in some settings more appropriate than linear or maximum penalties. Indeed, and as discussed in [3], quadratic penalties avoid schedules in which a single or only a few jobs contribute the majority of the cost, without regard to how the overall cost is distributed.

The assumption that no machine idle time is allowed is also actually appropriate for many production settings. In fact, idle time should be avoided when the machine has limited capacity or high operating costs, and when starting a new production run involves high setup costs or times. Some specific examples of production settings where the no idle time assumption is appropriate have been given in [4, 5].

This problem has been previously considered, and both exact and heuristic approaches have been proposed. Regarding the exact approach, a lower bounding procedure and a branch-and-bound algorithm were developed in [6]. The heuristics include several dispatching rules and simple improvement procedures [7], beam search algorithms [8] and greedy randomized dispatching heuristics [9].

The corresponding problem with linear costs $\sum_{j=1}^{n} (h_j E_j + w_j T_j)$ has also been considered by several authors, and both exact [10 - 13] and heuristic [14 - 16] approaches have been proposed. The minimization of the quadratic lateness, where the lateness of job j is defined as $L_j = C_j - d_j$, has also been studied in [17 - 20]. Baker and Scudder [21] and Hoogeveen [22] provide excellent surveys of scheduling problems with earliness and tardiness penalties, while a review of scheduling models with inserted idle time is given in [23].

The considered problem is combinatorial in nature, since a schedule is given by a permutation of the jobs, and there are $n!$ possible permutations. In this paper, we present several genetic algorithms based on a random key alphabet. In the random key approach, each chromosome is a vector of n random numbers between 0 and 1. Such a chromosome can be decoded into a permutation of the jobs, and therefore a schedule, by performing a simple sort operation.

The various versions of the genetic approach differ on the generation of both the initial population and the individuals added in the migration step, as well as on the use of local search. The genetic algorithms are compared with existing procedures, as well as with optimal solutions for some problem sizes, on a wide range of test instances.

The remainder of this paper is organized as follows. In the next section, we describe the proposed genetic algorithm approach, and present the several versions that were considered. Then, the computational results are reported. Finally, we provide some concluding remarks.

The proposed genetic algorithm procedures

In this section, we first briefly describe the main features of genetic algorithms. Then, the encoding used to represent the problem solutions is presented. The evolutionary strategy, i.e. the transitional process between consecutive populations, is also described. Finally, we present the six different versions that were considered for the proposed genetic algorithm approach.

Genetic algorithms

Genetic algorithms are adaptive methods that can be used to solve optimization problems. The term genetic algorithm was first used by Holland [24]. Even though Holland's work placed little emphasis on optimization, the majority of the research on genetic algorithms has indeed since been focused on solving optimization problems. Due to their increasing popularity in recent years, the literature on genetic algorithms now includes a quite large number of papers. References describing in detail the genetic algorithm approach and its applications can be found in [25 - 27].

Genetic algorithms are based on the evolution process that occurs in natural biology. Indeed, over the generations, natural populations tend to evolve

according to the principles of natural selection or survival of the fittest. The genetic algorithms mimic this process, by evolving populations of solutions to optimization problems.

In order to apply a genetic algorithm to a specific problem, it is first necessary to choose a suitable encoding or representation. In this encoding, a solution to the problem being considered is represented by a set of parameters, known in genetic terminology as genes. These genes are joined together in a string of values that represents (or encodes) the solution to the problem; this string is denoted as a chromosome or individual. A fitness value is also associated with each chromosome. This value measures the quality of the solution represented by that chromosome.

At each iteration, the genetic algorithm evolves the current population of chromosomes into a new population. This evolution is conducted using selection, crossover and mutation mechanisms. Some of the current individuals may be simply selected and copied to the new population. Also, a crossover operator is used in the reproduction phase to combine parent individuals selected from the current population, in order to produce offspring which are placed in the new population. The parent chromosomes are chosen randomly, although this selection is usually performed using a scheme which favours fitter individuals. The genes of the two parents are then combined by the crossover operator, yielding one or more offspring. Finally, a mutation operator can be applied to some individuals, in order to change one or more of their genes.

The reproduction phase and the crossover operator tend to increase the quality of the populations, since fitter individuals are more likely to be selected as parents. However, they also tend to force a convergence of those populations (i.e. the individuals tend to become quite similar). This convergence effect can be offset by the mutation mechanism. Indeed, by changing the genetic material, the mutation operator tries to guarantee the diversity of the population, thereby ensuring a more extensive search of the solution space.

Chromosome representation and decoding

The genetic algorithm approach proposed in this paper uses the random key alphabet U(0,1) [28] to encode the chromosomes. In the random key encoding, each gene is a uniform random number between 0 and 1. Consequently,

each chromosome is then encoded as a vector of random keys (random numbers). Therefore, in the proposed algorithms each chromosome is composed of n genes g_j, $j = 1, 2, \ldots, n$, so the size of each chromosome is equal to the number of jobs, i.e. chromosome = $(g_1, g_2, \ldots, g_n)$.

In order to calculate the fitness of an individual, it is first necessary to decode its chromosome into the corresponding solution to the considered problem, i.e. into a sequence of the jobs. This decoding or mapping of a chromosome into a schedule is accomplished by performing a simple sort of the jobs. The priorities used in this sorting operation are provided by the genes. More specifically, the priority of job j in the sorting operation is equal to g_j (see figure 1 for an example).

An important feature of the random key alphabet is the fact that all offspring generated by crossover are feasible solutions. This is accomplished by moving the feasibility issue into the chromosome decoding procedure. If any vector of random numbers can be decoded into a feasible solution, then any chromosome obtained via crossover also corresponds to a feasible solution. Through its internal dynamics, the genetic algorithm then learns the relationship between random key vectors and solutions with good fitness and objective function values.

This feature is a significant advantage of the random key alphabet over the more natural encoding where each chromosome is a permutation of the job indexes. Indeed, with the natural encoding, the crossover operation is made more difficult and complicated by the need to assure that the resulting offspring correspond to a feasible solution.

The evolutionary strategy

A quite large number of genetic algorithm variants can be obtained by choosing different selection, reproduction, crossover and mutation operators. We now describe the evolutionary strategy employed in the proposed approach, i.e. the specific mechanisms that are used to generate a new population from the current set of individuals.

Throughout the procedure, the size of the population is kept constant. This size is set equal to a multiple *pop_mult* of the size of the problem (i.e. the number of jobs n), where *pop_mult* is a user-defined parameter. This strategy has proved

adequate in previous applications of genetic algorithms based on the same evolutionary approach [29 - 31].

Given a current population, the next population is obtained through elitist selection, crossover and migration mechanisms. The discussion of the generation of the initial population is deferred to the next section. The calculation of the fitness value of a chromosome will also be addressed in that section.

The elitist selection strategy [25] copies some of the best individuals in the current population to the new population. The number of chromosomes that are copied in this elitist selection phase is set equal to a proportion *elit_prop* of the population size, where *elit_prop* is a user-defined parameter. The advantage of the elitist selection strategy over the traditional generational approach where the entire population is completely replaced with new chromosomes is that the best individual in the population improves monotonically over time. A potential downside is that it can lead to a premature convergence of the population. However, this can be overcome by using high mutation or migration rates.

In the proposed evolutionary strategy, the migration mechanism is used instead of the traditional gene-by-gene mutation operator. In the migration phase, new individuals are generated and added to the new population. The number of newly generated individuals is equal to a proportion *mig_prop* of the population size, where *mig_prop* is a user-defined parameter. The specific process by which these new individuals are generated will be addressed in the next section. Like in the traditional mutation operator, and as previously mentioned, the migration mechanism tries to prevent premature convergence, as well as to assure the diversity of the population and an extensive search of the solution space.

Finally, the remaining individuals of the new population are generated via crossover. In the reproduction and crossover phase, two parents are initially selected. The first parent is chosen at random from the elite individuals in the current population (i.e. the individuals that are copied to the new population in the elitist selection phase). The second parent, on the other hand, is randomly selected from the entire current population. Then, the parameterized uniform crossover method [32], described below, is used to create an offspring that is added to the new population. This process is repeated until the new population has been fully generated.

In the parameterized uniform crossover method, a random uniform number between 0 and 1 is generated for each gene. Then, this random number is compared with a user-defined parameter *cross_prob*. If the random number is less than or equal to the *cross_prob* parameter, the gene in the offspring is set equal to the corresponding gene in the first parent. Otherwise, the value of the gene is instead copied from the second parent (see figure 2 for an example).

The proposed evolutionary strategy is repeated until a stopping criterion is met. In our approach, we have chosen the number of consecutive iterations without improvement as stopping criterion. Thus, the genetic algorithms terminate when *stop_iter* consecutive populations have been generated without improving the best solution found so far, where *stop_iter* is a user-defined parameter. The evolutionary strategy is depicted in figure 3, and the main steps of the proposed approach are presented in figure 4.

The genetic algorithm versions

The discussion of the generation of the initial population and the new individuals added in the migration step, as well as the calculation of the fitness value, have been deferred to this section. Indeed, different strategies were considered for these issues. Therefore, we developed six genetic algorithm versions, corresponding to various combinations of these strategies. These six versions will now be described, and their main characteristics are summarized in table 1.

The first version consists in a basic genetic algorithm, and is therefore denoted simply by GA. In this version, the entire initial population, as well as all the new individuals created in the migration step, are generated randomly. Also, the fitness value of a chromosome is set equal to the opposite of the objective function value of the sequence associated with that chromosome. This ensures that the individuals with a lower value of the objective function have a higher fitness.

The second version differs from the GA procedure only in the way in which the initial population is generated. Therefore, the migration step and the calculation of the fitness value are identical to those in the GA algorithm. However, in the second version, the initial population is not entirely generated at random. Indeed, the initial population contains four non-random chromosomes, while the remaining individuals are again randomly generated. Indeed, previous

studies, e.g. [33, 34], have shown that introducing chromosomes that correspond to solutions generated by simple heuristics can improve the performance of a genetic algorithm. In this paper, this will be referred to as initializing the first population. This second version will then be denoted by GA_IN, since the basic genetic algorithm (GA) is enhanced with the initialization (IN) of the first population.

These four non-random chromosomes are created so that their corresponding schedules are equal to the sequences generated by four of the dispatching rules considered in [7]. More specifically, these four heuristics are the procedures denoted in [7] by WPT_s_j_E, EDD, WPT_s_j_T and ETP_v2. The WPT_s_j_E and WPT_s_j_T heuristics performed well for instances where most jobs were early and tardy, respectively. The EDD heuristic performed better than the WPT_s_j_E and WPT_s_j_T rules when there was a greater balance between the number of early and tardy jobs. Finally, the ETP_v2 heuristic is the best-performing of the dispatching rules analysed in [7]. In order to keep the paper self-contained, these four procedures are described in detail in the appendix.

The third version also sets the fitness value of a chromosome equal to the opposite of the objective function value of the sequence associated with that chromosome. Furthermore, this version also includes in the initial population the same four non-random chromosomes used in the GA_IN procedure. However, the third version employs an additional heuristic procedure in the generation of the initial population, as well as on the migration phase.

This additional heuristic procedure is the best-performing of the greedy randomized dispatching rules proposed in [9]. These heuristics perform a greedy randomization of the ETP_v2 rule, so a different schedule can be obtained each time one of these rules is executed. The greedy randomized dispatching heuristic that is used in the third proposed algorithm is the procedure denoted in [9] by RCL_VB, since this strategy provided the best results, both in solution quality and in computation time, among all the approaches analysed in [9]. Again, the RCL_VB procedure is described in detail in the appendix. The third genetic algorithm version will then be identified by GA_GR, since the genetic algorithm makes use of a greedy randomized (GR) dispatching rule.

The GA_GR heuristic can be seen as performing a hybridization of the genetic algorithm and GRASP metaheuristics. Indeed, in this version, the greedy

randomized strategy that is used in the construction phase of a GRASP procedure is employed to generate some chromosomes for the genetic algorithm. This hybridization has been used in previous studies, e.g. [35, 36].

The initial population of the GA_GR procedure is then generated as follows. First, and as previously mentioned, the four non-random chromosomes that are used in the GA_IN procedure are added to the population. Then, a proportion *init_gr* of the remainder of the initial population is generated using the RCL_VB greedy randomized heuristic, where *init_gr* is a user-defined parameter. Thus, these chromosomes are created so that they correspond to schedules generated by the RCL_VB procedure. Finally, the remaining chromosomes of the initial population are generated at random.

As for the migration step, in the GA_GR heuristic this phase is executed as follows. First, a proportion *mig_gr* of the chromosomes to be generated in this step are created using the RCL_VB procedure, where *mig_gr* is again a user-defined parameter. The remainder of the migration phase chromosomes are then randomly generated.

The remaining three versions differ from their GA, GA_IN and GA_GR counterparts only in the calculation of the fitness value. That is, these last three versions generate the initial population and perform the migration step in the same way as the GA, GA_IN and GA_GR procedures. However, these last versions additionally use a local search procedure to improve the sequence that is decoded from a chromosome.

More specifically, the fitness of a chromosome is calculated as follows in the remaining three versions. First, the chromosome is decoded, in order to get its corresponding sequence (similarly to what is done in the GA, GA_IN and GA_GR algorithms). Then, however, a local search procedure is used to improve this sequence. The fitness value is set equal to the opposite of the objective function value of this improved sequence. Finally, the chromosome is changed (by rearranging its genes), so that it now corresponds to the improved sequence obtained after the application of the local search procedure.

These last three versions of the proposed approach combine a genetic evolutionary strategy with a local search procedure. Therefore, they can also be viewed as memetic algorithms [37]. Hence, these versions will be denoted by MA, MA_IN and MA_GR, since the only difference from their GA counterparts

resides in the use of local search which, as was just mentioned, allows them to be seen as memetic algorithms (MA).

We considered three local search procedures: adjacent pairwise interchanges (API), 3-swaps (3SW) and first-improve interchanges (INTER). The API procedure considers pairs of adjacent jobs, and interchanges such a pair when that swap improves the objective function value. The procedure terminates when no improving adjacent interchange can be performed. The 3SW method instead considers three consecutive jobs. All the possible permutations of the three jobs are analysed, and the current configuration is replaced with the best when this improves the objective function value. Again, the procedure terminates when no improving 3-swap is found. Finally, the INTER procedure considers interchanging two jobs, regardless of whether or not they are adjacent. An interchange is performed whenever it improves the objective function value, and the procedure stops when no further improving interchange can be found.

Computational results

In this section, we first present the set of problems used in the computational tests. Then, the preliminary computational experiments are described. These experiments were performed to determine adequate values for the parameters required by the several genetic algorithms. Finally, the computational results are presented. We first compare the genetic algorithms with existing procedures, and the heuristic results are then evaluated against optimum objective function values for some instance sizes. Throughout this section, and in order to avoid excessively large tables, we will sometimes present results only for some representative cases.

Experimental design

The computational tests were performed on a set of problems with 10, 15, 20, 25, 30, 40, 50, 75 and 100 jobs. These problems were randomly generated as follows.

For each job j, an integer processing time p_j, an integer earliness penalty h_j and an integer tardiness penalty w_j were generated from one of the two uniform distributions [45, 55] and [1, 100], to create low (L) and high (H) variability, respectively. In the tables, the processing time and penalty variability will be

denoted by *var*, and so instances with low and high variability will be identified by *var* = L and *var* = H, respectively.

For each job *j*, an integer due date d_j is generated from the uniform distribution $[P(1 - T - R/2), P(1 - T + R/2)]$, where P is the sum of the processing times of all jobs, T is the tardiness factor, set at 0.0, 0.2, 0.4, 0.6, 0.8 and 1.0, and R is the range of due dates, set at 0.2, 0.4, 0.6 and 0.8.

For each combination of problem size *n*, processing time and penalty variability *var*, T and R, 50 instances were randomly generated. Therefore, a total of 1200 instances were generated for each combination of problem size and variability.

All the algorithms were coded in Visual C++ 6.0, and executed on a Pentium IV - 2.8 GHz personal computer. Due to the large computational times that would be required, the GA heuristic was not applied to the instances with 100 jobs.

Preliminary tests

In this section, we describe the preliminary experiments that were performed to determine adequate values for the parameters required by the genetic algorithms. A separate problem set was used to conduct these preliminary experiments. This test set included instances with 25 and 50 jobs, and contained 5 instances for each combination of instance size, processing time and penalty variability, T and R. The instances in this smaller test set were generated randomly just as previously described for the full problem set.

We considered the following values for the several parameters required by the proposed genetic algorithms:

pop_mult = {1, 2, 3};

elit_prop = {0.05, 0.10, 0.15, 0.20};

mig_prop = {0.10, 0.15, 0.20, 0.25};

cross_prob = {0.6, 0.7, 0.8};

stop_iter = {10, 30, 50};

init_gr = {0.1, 0.2, ... , 0.9};

mig_gr = {0.1, 0.2, ... , 0.9}.

The intervals for the *elit_prop*, *mig_prop* and *cross_prob* values were based on previous applications of genetic algorithms based on the same

evolutionary approach [29 - 31]. Indeed, good results have consistently been obtained using values inside the considered ranges. The intervals for the *pop_mult* and *stop_iter* parameters were determined based not only on a previous application of this evolutionary strategy to a scheduling problem [29], but also on some initial tests. For the MA, MA_IN and MA_GR versions, we additionally considered the API, 3SW and INTER local search procedures, as previously mentioned.

The genetic algorithms were then applied to the test instances for all parameter (and local search procedure, for the MA, MA_IN and MA_GR versions) combinations. A thorough analysis of the objective function values and runtimes was then conducted, in order to select the values that provided the best trade-off between solution quality and computation time. The parameter values and local search procedure selected for the several genetic versions are given in table 2.

The same *elit_prop*, *mig_prop* and *cross_prob* values proved appropriate for all the versions. For the versions that use a local search procedure, the results were actually virtually identical for all the combinations of these parameters. For the other versions, there were some small differences in performance, and the chosen values provided good results for all instance types.

Table 2 shows that smaller values are required for the parameters *pop_mult* and *stop_iter* as the sophistication of the genetic versions increases. In fact, smaller populations and/or a lower number of consecutive iterations without improvement can be used, without compromising the solution quality, with the introduction in the genetic approach of features such as local search, population initialization and generation of greedy randomized solutions.

The *init_gr* parameter is smaller for the MA_GR version. In the GA_GR procedure, a higher percentage of greedy randomized solutions is then required in order to generate an initial population that contains high quality solutions. In the MA_GR version, however, the local search procedure already improves the quality of the solutions in the initial population, so only a lower percentage of greedy randomized initial solutions is required.

For both the GA_GR and MA_GR algorithms, the most appropriate value of the *mig_gr* parameter was equal to 0.5. Therefore, half of the new chromosomes introduced in the migration step are produced by the greedy

randomized heuristic, while the remaining half are randomly generated. This shows that the best results are obtained when there is a balance between the introduction of relatively good solutions (generated by the greedy randomized heuristic) and random solutions. Indeed, the greedy randomized solutions increase the solution quality of the population, but the random solutions are also important, since they assure the diversity of the population and avoid its premature convergence.

Finally, the API local search procedure was selected. This procedure provided results that were virtually identical to those given by the other methods, and was significantly faster. We recall that the parameter values were selected with the objective of obtaining the best trade-off between solution quality and computation time. Therefore, lower objective function values can still be obtained for some of the test instances, at the cost of increased computation times, by increasing the *pop_mult* or *stop_iter* values.

Comparison with existing heuristics

In this section, the proposed genetic algorithms are compared with existing heuristic procedures. On the one hand, the proposed algorithms are compared with the recovering beam search (RBS) heuristic developed in [8]. Additionally, the RCL_VB_3SW greedy randomized heuristic proposed in [9] is also included in the heuristic comparison. The RCL_VB_3SW procedure essentially generates greedy randomized solutions using the RCL_VB strategy previously mentioned, and applies the 3SW improvement procedure to each of those solutions. In the following, and in order to reduce the size of the identifier, the RCL_VB_3SW procedure will be denoted as R_V_3.

Among the existing heuristics, the RBS algorithm provides the best average performance for small and medium size instances, while the R_V_3 procedure is the heuristic of choice for medium to large problems. The RBS algorithm includes a final improvement step that uses the 3SW improvement procedure. Also, in the R_V_3 heuristic, and as previously mentioned, each of the greedy randomized constructed schedules is improved by the 3SW procedure. Therefore, the 3SW method was also applied, as a final improvement step, to the best solution generated by each of the genetic algorithms. For each instance, 10

independent runs, with different random number seeds, were performed for all versions of the genetic algorithm (as well as for the R_V_3 heuristic).

Table 3 provides the mean relative improvement in objective function value over the RBS procedure. In table 4, we give the percentage number of times the R_V_3 heuristic and the genetic algorithms performs better (<), equal (=) or worse (>) than the RBS procedure. The relative improvement over the RBS heuristic is calculated as (*rbs_ofv* - *heur_ofv*) / *rbs_ofv* × 100, where *rbs_ofv* and *heur_ofv* are the objective function values of the RBS procedure, on the one hand, and the R_V_3 procedure or the appropriate genetic version, on the other hand.

In table 3, the *avg* column provides the relative improvement calculated with the average of the objective function values obtained for all the 10 runs. In the *best* column, the value of relative improvement has been calculated using the best of those 10 objective function values. The results in the *avg* column provide an indication of the relative improvement we can achieve if the procedure is executed only once, while the *best* column shows the improvement that can be obtained if the algorithm is allowed to perform 10 runs.

The processing time and penalty variability has a major impact on the difficulty of the problem, and therefore on the differences between the results obtained by the several heuristic procedures. When the variability is low, the problem is much easier, and even simple procedures can obtain optimum or near optimum results, so there is little or no room for more sophisticated procedures to provide a large improvement. This has been established and discussed in the previous literature on this problem, and will also be shown quite clearly by the comparison with the optimum results performed in the next section.

For the low variability instances, the performance of the considered procedures is nearly identical. The results in table 4 show that the heuristics do generate different solutions for a few instances. However, the objective function values of those solutions are nevertheless extremely close, as can be seen from the relative improvement values presented in table 3, which are virtually equal to 0. When the variability is high, however, the problem becomes considerably more difficult, and the difference in performance between the several algorithms is much more noticeable.

When the average results over the 10 runs are considered, the three genetic versions with local search are superior to their GA, GA_IN and GA_GR

counterparts, as shown by the results in tables 3 and 4. Indeed, not only do the MA, MA_IN and MA_GR versions provide a larger relative improvement over the RBS heuristic, but they also give better results for a larger percentage of the test instances, and are seldom inferior to the RBS procedure. Therefore, the addition of a local search procedure improves the average performance, in terms of solution quality, of the genetic algorithms.

The average performance of the three genetic versions that incorporate a local search procedure is quite similar. Indeed, the results given by the MA, MA_IN and MA_GR heuristics are rather close. When local search is not used, there is only a slight difference in the average performance. In fact, the results show that the GA_IN version is, in terms of average performance, somewhat inferior to the GA and GA_GR algorithms.

When the best result over the 10 runs is considered, the performance of the several genetic algorithms, in terms of solution quality, is quite close. The genetic versions that incorporate a local search procedure are extremely robust. Indeed, the best and average relative improvement values are usually quite close for the MA, MA_IN and MA_GR algorithms.

The results given in tables 3 and 4 show that the several genetic versions are clearly superior to the RBS and R_V_3 heuristics. Indeed, the genetic algorithms provide a relative improvement of about 1-3% over the RBS procedure. Also, the genetic algorithms give better results for a larger percentage of the test instances. The versions with local search, in particular, give better results for a quite large percentage of the instances, and are seldom inferior to the RBS procedure. The genetic versions also clearly outperform the R_V_3 heuristic. Furthermore, the improvement provided by the genetic algorithms is increasing with the instance size.

In table 5, we present the effect of the T and R parameters on the relative improvement (calculated with the average objective function value) over the RBS procedure, for instances with 50 jobs. The relative improvement is quite minor for the extreme values of T (T = 0.0 and T = 1.0). When the tardiness factor assumes more intermediate values, the relative difference in objective function values becomes much larger. Indeed, for some parameter combinations the genetic algorithms provide a relative improvement of over 10%.

Again, this is in accordance with results obtained in the existing literature on this problem. Indeed, the problem is much easier when most jobs are early (T = 0.0) or tardy (T = 1.0). Once more, this will also be shown quite clearly in the next section. For the more intermediate values of T, the number of early and tardy jobs becomes more balanced, and the problem becomes harder. Hence, there is more room for improvement in the harder instances with intermediate values of the tardiness factor. Therefore, the genetic versions provide a large relative improvement for the more difficult instances. In fact, as previously mentioned, for the high variability instances with an intermediate value of the tardiness factor T, the genetic algorithms can provide an improvement of over 10%.

The heuristic runtimes (in seconds) are presented in table 6. For the genetic algorithms and the R_V_3 procedure, we provide the average runtime, i.e. the average of the runtimes for each of the 10 runs. The R_V_3 algorithm is quite clearly the fastest and most efficient of the considered heuristics. The RBS procedure is more computationally demanding, but is faster than the genetic algorithms. Nevertheless, the genetic procedures, with the exception of the GA version, are still somewhat efficient, since they are capable of solving instances with 100 jobs in about 2 seconds.

The GA procedure is considerably more computationally demanding than the other genetic versions. This may seem surprising, since the other versions incorporate several elements that require additional time when compared with the simpler implementation used in the GA algorithm. Indeed, initializing the first population, generating greedy randomized solutions and using local search requires additional computation time.

However, the inclusion of these features also increases the sophistication of the procedure, and enables it to find a high quality solution in fewer iterations and using smaller populations. Therefore, and as previously mentioned, lower values are required for the parameters *pop_mult* and *stop_iter* in the more evolved genetic versions. The computational results show that the time required by the additional features included in the GA_IN to MA_GR versions is more than offset by the fewer iterations and smaller populations required by these procedures.

In terms of solution quality, all the genetic versions were virtually identical when the best result was considered, while the versions with local search were somewhat superior in average performance. However, in terms of computation

effort, the more sophisticated versions, particularly those with local search, are clearly more efficient, and can then obtain similar or better results in less computation time.

The MA_GR version is then the recommended heuristic for small and medium instance sizes. This procedure provides the best results (along with the other versions with local search), and is the most efficient of the genetic algorithms, with the exception of the GA_IN version. For large problems, however, a genetic algorithm approach will require excessive time. The RBS procedure can be applied to slightly larger instances than the genetic algorithm, but for the quite large problems only the R_V_3 algorithm (or eventually a dispatching heuristic) will be able to provide results in reasonable computation times.

Comparison with optimum results

In this section, we compare the heuristic procedures with the optimum objective function values, for instances with up to 20 jobs. The optimum objective function values were obtained with the branch-and-bound algorithm developed in [6]. Table 7 gives the average of the relative deviations from the optimum (%*dev*), calculated as $(H - O) / O \times 100$, where H and O are the heuristic and the optimum objective function values, respectively. The percentage number of times each heuristic generates an optimum schedule (%*opt*) is also provided.

The results given in table 7 confirm that, as mentioned in the previous section, the problem is much more difficult when the processing time and penalty variability is high. In the low variability setting, the heuristic procedures provide optimum results for nearly all the instances. In fact, the MA_GR algorithm actually generates an optimal solution for all instances with low variability.

When the variability is high, however, the problem becomes harder, and there is more room to improve upon the RBS and R_V_3 results. Indeed, the genetic algorithms clearly outperform these procedures for the high variability instances. This is particularly clear for the better performing versions with local search. In fact, these versions provide an optimal solution for over 96% of the test instances. The GA, GA_IN and GA_GR algorithms also perform quite well. Indeed, not only is their relative deviation from the optimum extremely low, but they also provide optimal solutions for about 80-90% of the instances.

In table 8, we present the effect of the T and R parameters on the relative deviation from the optimum, for instances with 20 jobs. Again, the results given in this table confirm that, as previously mentioned, the problem is harder when there is a greater balance between the number of early and tardy jobs. In fact, when $T \leq 0.2$ or $T \geq 0.8$, all the heuristics are optimal or nearly optimal. The problem, however, becomes harder when $T = 0.4$ or $T = 0.6$, particularly when the due date range is low. For these more difficult instances, the improvement the genetic algorithms provide over the RBS and R_V_3 heuristics is much higher. Indeed, for these instances, the genetic procedures are much closer to the optimum. This is particularly true for the versions with local search, which are optimal for nearly all of the more difficult instances.

Conclusion

In this paper, a genetic approach was proposed for the single machine scheduling problem with quadratic earliness and tardiness costs, and no machine idle time. Several genetic algorithms based on this approach were presented. These versions differ on the generation of both the initial population and the individuals added in the migration step, as well as on the use of local search.

We first performed initial experiments, in order to determine appropriate values for the parameters required by the genetic algorithms. The proposed procedures were then compared with the best existing heuristics, as well as with optimal solutions for the smaller instance sizes.

The genetic algorithms, particularly the versions with local search, clearly outperform the existing procedures. Also, the genetic heuristics are quite close to the optimum. Indeed, the versions with local search provided an optimal solution for over 96% (and in some cases actually all) of the test instances. The improvement in performance provided by the genetic algorithms is much larger for the more difficult instances, i.e. instances with a high processing time and penalty variability and a greater balance between the number of early and tardy jobs. Also, the improvement given by the genetic versions increases with the instance size.

The performance of the proposed genetic approach was improved by the initialization of the initial population, the generation of greedy randomized solutions and the addition of a local search procedure. In terms of solution quality,

all the genetic versions were virtually identical when the best result was considered, while the versions with local search were somewhat superior in average performance. However, in terms of computational effort, the more sophisticated versions, particularly those which use a local search procedure, are clearly more efficient, and can then obtain similar or better results in less computation time. Therefore, the time required by the additional elements included in the more sophisticated versions is more than offset by the fact that they require smaller populations and/or a lower number of iterations without improvement.

The MA_GR algorithm is the new heuristic of choice for small and medium instance sizes. Indeed, and on the one hand, this procedure provided the best results in terms of solution quality, along with the other genetic versions with local search. Also, and on the other hand, this is the most efficient of the genetic algorithms, with the exception of the GA_IN version.

As a possibility for future research, other metaheuristic procedures, such as tabu search or iterated local search, could be tested on the considered problem. Also, the problem could be extended by introducing elements such as distinct release dates and/or setup times, which are relevant in several practical settings. Yet another possible direction for future research is to apply the quadratic early/tardy objective function to scheduling problems with more than one processor, such as parallel machines or flowshops.

References

1. Wagner BJ, Davis DJ, Kher H (2002) The production of several items in a single facility with linearly changing demand rates. Decis Sci 33:317-346.

2. Taguchi G (1986) Introduction to quality engineering. Asian Productivity Organization, Tokyo, Japan.

3. Sun X, Noble JS, Klein CM (1999) Single-machine scheduling with sequence dependent setup to minimize total weighted squared tardiness. IIE Trans 31:113-124.

4. Korman K (1994). A pressing matter. Video February:46-50.

5. Landis K (1993) Group technology and cellular manufacturing in the Westvaco Los Angeles VH department. Project report in IOM 581, School of Business, University of Southern California.

6. Valente JMS (2007) An exact approach for single machine scheduling with quadratic earliness and tardiness penalties. Working Paper 238, Faculdade de Economia, Universidade do Porto, Portugal.

7. Valente JMS, Alves RAFS (2008) Heuristics for the single machine scheduling problem with quadratic earliness and tardiness penalties. Comp Oper Res 35:3696-3713.
8. Valente JMS (2010) Beam search heuristics for quadratic earliness and tardiness scheduling. J Oper Res Soc 61:620-631.
9. Valente JMS, Moreira MRA (2009) Greedy randomized dispatching heuristics for the single machine scheduling problem with quadratic earliness and tardiness penalties. Int J Adv Manuf Technol . 44:995-1009
10. Abdul-Razaq T, Potts CN (1988) Dynamic programming state-space relaxation for single machine scheduling. J Oper Res Soc 39:141-152.
11. Li G (1997) Single machine earliness and tardiness scheduling. Eur J Oper Res 96:546-558.
12. Liaw CF (1999) A branch-and-bound algorithm for the single machine earliness and tardiness scheduling problem. Comp Oper Res 26:679-693.
13. Valente JMS, Alves RAFS (2005) Improved lower bounds for the early/tardy scheduling problem with no idle time. J Oper Res Soc 56:604-612.
14. Ow PS, Morton TE (1989) The single machine early/tardy problem. Manag Sci 35:177-191.
15. Valente JMS, Alves RAFS (2005) Improved heuristics for the early/tardy scheduling problem with no idle time. Comp Oper Res 32:557-569.
16. Valente JMS, Alves RAFS (2005) Filtered and recovering beam search algorithms for the early/tardy scheduling problem with no idle time. Comp Ind Eng 48:363-375.
17. Gupta SK, Sen T (1983) Minimizing a quadratic function of job lateness on a single machine. Eng Costs Prod Econ 7:187-194.
18. Sen T, Dileepan P, Lind MR (1995) Minimizing a weighted quadratic function of job lateness in the single machine system. Int J Prod Econ 42:237-243.
19. Su LH, Chang PC (1998) A heuristic to minimize a quadratic function of job lateness on a single machine. Int J Prod Econ 55:169-175.
20. Schaller J (2002) Minimizing the sum of squares lateness on a single machine. Eur J Oper Res 143:64-79.
21. Baker KR, Scudder GD (1990) Sequencing with earliness and tardiness penalties: A review. Oper Res 38:22-36.
22. Hoogeveen H (2005) Multicriteria scheduling. Eur J Oper Res 167:592-623.
23. Kanet JJ, Sridharan V (2000) Scheduling with inserted idle time: Problem taxonomy and literature review. Oper Res 48:99-110.
24. Holland JH (1975) Adaptation in natural and artificial systems. University of Michigan Press, Ann Arbor, Michigan (re-issued in 1992 by MIT Press).
25. Goldberg DE (1989) Genetic algorithms in search, optimization, and machine learning. Addison-Wesley, Reading, Massachusetts..
26. Reeves CR (1997) Genetic algorithms for the operations researcher. INFORMS J Comp 9:231-250.
27. Reeves C (2003) Genetic algorithms. In: Glover F, Kochenberger GA (eds) Handbook of metaheuristics. Kluwer Academic Publishers, Dordrecht, pp 55-82.

28. Bean JC (1994) Genetics and random keys for sequencing and optimization. ORSA J Comp 6:154-160.
29. Valente JMS, Gonçalves JF (2009) A genetic algorithm approach for the single machine scheduling problem with linear earliness and quadratic tardiness penalties. Comp Oper Res 36:2707-2715.
30. Gonçalves JF (2007) A hybrid genetic algorithm-heuristic for a two-dimensional orthogonal packing problem. Eur J Oper Res 183:1212-1229.
31. Gonçalves JF, Mendes JJM, Resende MGC (2005) A hybrid genetic algorithm for the job shop scheduling problem. Eur J Oper Res 167:77-95.
32. Spears WM, De Jong KA (1991) On the virtues of parameterized uniform crossover. In: Belew R, Booker L (eds) Proceedings of the fourth international conference on genetic algorithms. Morgan Kaufman, San Mateo, CA, pp 230-236.
33. Reeves CR (1995) A genetic algorithm for flowshop sequencing. Comp Oper Res 22: 5-13.
34. Ahuja RK, Orlin JB (1997) Developing fitter GAs. INFORMS J Comp 9:251-253.
35. Ahuja RK, Orlin JB, Tiwari A (2000) A greedy genetic algorithm for the quadratic assignment problem. Comp Oper Res 27: 917-934.
36. Armony, Klincewicz JC, Luss H, Rosenwein MB (2000) Design of stacked self-healing rings using a genetic algorithm. J Heuristics 6:85-105.
37. Moscato P, Cotta C (2003) A gentle introduction to memetic algotihms. In: Glover F, Kochenberger GA (eds) Handbook of metaheuristics. Kluwer Academic Publishers, Dordrecht, pp 105-144.

Appendix

In the appendix, we will provide details concerning the EDD, WPT_s_j_E, WPT_s_j_T and ETP_v2 dispatching heuristics presented in [7] and the RCL_VB greedy randomized dispatching rule proposed in [9].

The earliest due date (EDD) rule simply schedules the jobs in non-decreasing order of their due dates. The WPT_s_j_E, WPT_s_j_T and ETP_v2 dispatching rules calculate, whenever the machine becomes available, a priority index value for each unscheduled job. The job with the largest priority value is then selected for immediate processing. Let $I_j(t)$ denote the priority index of job j at time t. The WPT_s_j_E, WPT_s_j_T and ETP_v2 dispatching rules use the following priority indexes:

WPT_s_j_E: $I_j(t) = (h_j / p_j) * [p - 2 * \max(s_j; 0)]$

WPT_s_j_T: $I_j(t) = (w_j / p_j) * [p + 2 * \max(t + p_j - d_j; 0)]$

ETP_v2: $$I_j(t) = \begin{cases} (w_j / p_j) * [p + 2 * (t + p_j - d_j)] & \text{if } s_j \leq 0 \\ \max\{(w_j / p_j) * p; (h_j / p_j) * (p - 2 * s_j)\} & \text{otherwise} \end{cases}$$

where $s_j = d_j - t - p_j$ is the slack of job j and p is the average processing time of the unscheduled jobs.

The WPT_s_j_E priority index includes both weighted processing time (WPT) and slack (s_j) components. Also, this rule performed well for instances where most jobs were early (E). The priority expression of the WPT_s_j_T heuristic also includes weighted processing time and (inverse) slack components. However, this procedure performed well for instances where most jobs were tardy (T). The ETP part of the ETP_v2 procedure's identifier stands for earliness and tardiness priority, since the priority index of this heuristic includes components relative to both early and tardy costs.

Several greedy randomized dispatching rules were proposed in [9]. The general pseudo-code for the greedy randomized construction of a schedule for a single machine scheduling problem is as follows:

1. Set $S = \varnothing$, $U = \{1, 2, \ldots, n\}$ and $t = 0$.
2. While $U \neq \varnothing$
   - 2.1. Calculate the priority value $I_j(t)$ for all jobs $j \in U$.
   - 2.2. Create a candidate list CL of the unscheduled jobs that will be considered to be scheduled in the next position.
   - 2.3. Calculate the score sc_j for all jobs $j \in CL$.
   - 2.4. Calculate the biased score bsc_j for all jobs $j \in CL$.
   - 2.5. Calculate the probability $prob_j$ of selecting each job $j \in CL$: $prob_j = bsc_j / \Sigma_{j \in CL} bsc_j$.
   - 2.6. Randomly select the next job to be scheduled from the jobs in CL according to the probabilities $prob_j$.
   - 2.7. Let k denote the chosen job. Add k to set S and remove it from set U. Set $t = t + p_k$.

3. Return the schedule in set *S*.

Various choices can be made regarding steps 2.1, 2.2, 2.3 and 2.4, thereby leading to different greedy randomized heuristics. The RCL_VB strategy provided the best result among the alternatives analysed in [9]. The specific way in which steps 2.1 to 2.4 are implemented in this approach will now be described.

In step 2.1, the priority value is calculated using the ETP_v2 priority index. In step 2.2, the candidate list is created using a restricted candidate list (RCL) strategy. More specifically, the candidate list *CL* will contain the jobs with priority values $I_j(t) \in [I_{max} - \alpha(I_{max} - I_{min}), I_{max}]$, where I_{max} and I_{min} are the maximum and minimum values of the priorities of all the unscheduled jobs, respectively, and α is a user-defined parameter. After preliminary experiments, the value of α was set at 0.5 for $n \leq 25$, 0.05 when $25 < n < 100$ and 0.002 when $n \geq 100$.

The value biased (VB) approach was used in step 2.3, so the score of a job was simply set equal to its priority value, i.e. $sc_j = I_j(t)$. Finally, an exponential bias function was used in step 2.4. Therefore, we have $bsc_j = exp\_base^{sc_j}$, where *exp_base* is the user-defined base of the exponential expression. After preliminary experiments, the *exp_base* parameter was set at $1 + 0.1n^{-0.33}$.

Table 1 The proposed genetic algorithms

heuristics	initial population	migration step	fitness value
GA	random	random	equal to the opposite of the objective function value of the corresponding sequence
GA_IN	4 non-random chromosomes [1]	random	
GA_GR	4 non-random chromosomes [1] + RCL_VB greedy randomized chromosomes[2]	RCL_VB greedy randomized chromosomes[2] + random	
MA	random	random	equal to the opposite of the objective function value of the corresponding sequence after improvement by a local search procedure
MA_IN	4 non-random chromosomes [1]	random	
MA_GR	4 non-random chromosomes [1] + RCL_VB greedy randomized chromosomes[2]	RCL_VB greedy randomized chromosomes[2] + random	

[1] These are created so that their corresponding schedules are equal to the sequences generated by the WPT_s_j_E, EDD, WPT_s_j_T and ETP_v2 dispatching rules considered in [7].
[2] Some chromosomes are created so that they correspond to a schedule generated by the greedy randomized dispatching rule RCL_VB proposed in [9].

Table 2 Parameter values

	GA	GA_IN	GA_GR	MA	MA_IN	MA_GR
pop_mult	3	2	2	1	1	1
elit_prop	0.05	0.05	0.05	0.05	0.05	0.05
mig_prop	0.25	0.25	0.25	0.25	0.25	0.25
cross_prob	0.8	0.8	0.8	0.8	0.8	0.8
stop_iter	30	30	30	10	10	10
init_gr	---	---	0.4	---	---	0.1
mig_gr	---	---	0.5	---	---	0.5
local search	---	---	---	API	API	API

Table 3 Comparison with the RBS heuristic - relative improvement

		n = 25		*n* = 50		*n* = 75		*n* = 100	
var	heur	*best*	*avg*	*best*	*avg*	*best*	*avg*	*best*	*avg*
L	R_V_3	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	GA	0.00	0.00	0.00	0.00	0.00	0.00	---	---
	GA_IN	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	GA_GR	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	MA	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	MA_IN	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	MA_GR	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
H	R_V_3	0.15	-0.86	0.33	-0.28	0.43	-0.20	0.11	-0.27
	GA	1.36	0.84	2.23	2.06	2.43	2.32	---	---
	GA_IN	1.26	0.58	2.19	1.87	2.40	2.17	2.76	2.66
	GA_GR	1.29	0.92	2.20	1.97	2.41	2.27	2.76	2.60
	MA	1.37	1.29	2.23	2.20	2.43	2.41	2.78	2.77
	MA_IN	1.37	1.30	2.23	2.20	2.43	2.41	2.78	2.76
	MA_GR	1.37	1.28	2.23	2.18	2.43	2.40	2.78	2.77

Table 4 Comparison with the RBS heuristic - percentage of better, equal and worse results

		n = 25			*n* = 50			*n* = 75			*n* = 100		
var	heur	<	=	>	<	=	>	<	=	>	<	=	>
L	R_V_3	0.6	98.8	0.6	2.4	95.9	1.7	6.1	92.1	1.8	2.7	88.3	9.1
	GA	0.4	97.6	2.0	1.8	93.2	5.0	4.0	90.5	5.5	---	---	---
	GA_IN	0.4	97.1	2.5	1.2	93.3	5.5	1.9	92.2	5.9	2.5	88.6	8.9
	GA_GR	0.4	97.4	2.2	1.4	93.7	5.0	2.5	92.5	5.1	3.7	88.8	7.5
	MA	0.7	99.3	0.0	2.9	97.1	0.1	7.1	92.9	0.0	10.5	89.5	0.0
	MA_IN	0.7	99.3	0.0	2.9	97.1	0.0	7.1	92.9	0.0	10.5	89.5	0.0
	MA_GR	0.7	99.3	0.1	2.9	97.1	0.0	7.1	92.9	0.0	10.4	89.6	0.0
H	R_V_3	10.8	75.7	13.5	23.0	57.6	19.4	34.4	43.2	22.4	31.8	40.2	28.0
	GA	18.5	63.1	18.3	35.5	43.6	20.9	46.0	30.8	23.3	---	---	---
	GA_IN	15.1	66.9	18.1	30.2	48.9	20.9	37.6	42.0	20.4	41.5	40.7	17.8
	GA_GR	16.8	75.5	7.8	35.8	52.1	12.2	45.8	40.4	13.7	49.2	37.9	12.9
	MA	22.3	75.6	2.1	42.9	54.8	2.3	56.9	40.9	2.2	64.5	33.9	1.6
	MA_IN	22.3	75.8	1.9	43.0	54.9	2.2	56.9	41.2	1.9	64.3	34.2	1.5
	MA_GR	22.3	76.6	1.2	42.9	55.5	1.6	57.2	41.3	1.5	64.4	34.6	1.1

Table 5 Relative improvement over the RBS heuristic for instances with 50 jobs

		var = L				*var* = H			
heur	*T*	*R* = 0.2	*R* = 0.4	*R* = 0.6	*R* = 0.8	*R* = 0.2	*R* = 0.4	*R* = 0.6	*R* = 0.8
R_V_3	0.0	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	0.2	0.00	0.00	0.00	0.00	-0.01	-0.05	-0.02	-0.01
	0.4	0.00	0.00	0.00	0.00	-0.79	-0.32	-1.17	-1.17
	0.6	0.00	0.00	0.00	0.00	-1.34	-1.63	-0.71	-0.05
	0.8	0.00	0.00	0.00	0.00	0.42	0.13	-0.02	-0.02
	1.0	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.00
GA_IN	0.0	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	0.2	0.00	0.00	0.00	0.00	-0.04	-0.06	-0.04	-0.01
	0.4	0.00	0.00	0.00	0.00	4.30	1.37	0.06	-0.55
	0.6	0.00	0.00	0.00	0.00	16.45	10.12	7.02	2.33
	0.8	0.00	0.00	0.00	0.00	3.53	0.45	-0.01	0.00
	1.0	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
GA_GR	0.0	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	0.2	0.00	0.00	0.00	0.00	0.04	-0.01	0.00	0.01
	0.4	0.00	0.00	0.00	0.00	4.41	1.63	0.60	-0.09
	0.6	0.00	0.00	0.00	0.00	16.43	10.13	7.20	2.84
	0.8	0.00	0.00	0.00	0.00	3.59	0.49	0.02	0.02
	1.0	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.00
MA_IN	0.0	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	0.2	0.00	0.00	0.00	0.00	0.10	0.01	0.00	0.02
	0.4	0.00	0.00	0.00	0.00	5.24	2.25	1.13	0.27
	0.6	0.00	0.00	0.00	0.00	16.96	10.91	8.28	3.42
	0.8	0.00	0.00	0.00	0.00	3.64	0.51	0.04	0.03
	1.0	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.00
MA_GR	0.0	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	0.2	0.00	0.00	0.00	0.00	0.10	0.00	0.00	0.02
	0.4	0.00	0.00	0.00	0.00	5.07	2.18	1.02	0.28
	0.6	0.00	0.00	0.00	0.00	16.92	10.91	8.21	3.36
	0.8	0.00	0.00	0.00	0.00	3.64	0.51	0.04	0.03
	1.0	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.00

Table 6 Runtimes (in seconds)

var	heur	*n* = 15	*n* = 25	*n* = 50	*n* = 75	*n* = 100
L	RBS	0.002	0.007	0.037	0.104	0.226
	R_V_3	0.001	0.003	0.008	0.017	0.020
	GA	0.032	0.129	0.923	3.007	---
	GA_IN	0.008	0.019	0.068	0.149	0.262
	GA_GR	0.011	0.040	0.163	0.467	0.972
	MA	0.010	0.030	0.214	0.691	1.656
	MA_IN	0.009	0.029	0.209	0.682	1.637
	MA_GR	0.009	0.024	0.154	0.484	1.139
H	RBS	0.002	0.007	0.038	0.109	0.240
	R_V_3	0.002	0.004	0.013	0.030	0.034
	GA	0.033	0.131	0.937	3.046	---
	GA_IN	0.011	0.032	0.159	0.441	0.982
	GA_GR	0.014	0.065	0.488	2.110	3.647
	MA	0.010	0.032	0.253	0.907	2.362
	MA_IN	0.010	0.031	0.240	0.857	2.258
	MA_GR	0.009	0.026	0.183	0.631	1.523

Table 7 Comparison with optimum objective function values

		n = 10		*n* = 15		*n* = 20	
var	heur	*%dev*	*%opt*	*%dev*	*%opt*	*%dev*	*%opt*
L	RBS	0.00	99.92	0.00	100.00	0.00	99.67
	R_V_3	0.00	99.65	0.00	99.75	0.00	99.64
	GA	0.00	99.40	0.00	98.93	0.00	98.30
	GA_IN	0.00	99.27	0.00	98.60	0.00	97.58
	GA_GR	0.00	99.63	0.00	99.03	0.00	98.24
	MA	0.00	100.00	0.00	100.00	0.00	99.99
	MA_IN	0.00	100.00	0.00	100.00	0.00	99.98
	MA_GR	0.00	100.00	0.00	100.00	0.00	100.00
H	RBS	0.22	95.67	0.91	87.92	1.40	82.50
	R_V_3	0.29	93.12	0.71	85.74	1.03	80.36
	GA	0.00	90.48	0.00	82.77	0.00	77.57
	GA_IN	0.16	90.10	0.15	81.11	0.13	76.64
	GA_GR	0.04	94.66	0.06	88.48	0.05	85.44
	MA	0.00	99.97	0.00	98.61	0.00	96.44
	MA_IN	0.00	99.99	0.00	98.90	0.00	96.63
	MA_GR	0.00	99.99	0.00	98.91	0.00	96.82

Table 8 Relative deviation from the optimum for instances with 20 jobs

		var = L				*var* = H			
heur	*T*	*R* = 0.2	*R* = 0.4	*R* = 0.6	*R* = 0.8	*R* = 0.2	*R* = 0.4	*R* = 0.6	*R* = 0.8
RBS	0.0	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	0.2	0.00	0.00	0.00	0.00	0.10	0.00	0.00	0.00
	0.4	0.00	0.00	0.00	0.00	3.88	6.22	0.68	0.80
	0.6	0.00	0.00	0.00	0.00	10.94	3.63	2.64	0.74
	0.8	0.00	0.00	0.00	0.00	2.61	1.05	0.03	0.17
	1.0	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.00
R_V_3	0.0	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	0.2	0.00	0.00	0.00	0.00	0.09	0.03	0.01	0.01
	0.4	0.00	0.00	0.00	0.00	3.24	5.27	0.58	1.77
	0.6	0.00	0.00	0.00	0.00	7.89	2.68	2.12	0.50
	0.8	0.00	0.00	0.00	0.00	0.13	0.28	0.00	0.00
	1.0	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
GA_IN	0.0	0.00	0.00	0.00	0.00	0.00	0.01	0.00	0.00
	0.2	0.00	0.00	0.00	0.00	0.19	0.03	0.08	0.02
	0.4	0.00	0.00	0.00	0.00	0.31	0.32	0.63	0.96
	0.6	0.00	0.00	0.00	0.00	0.22	0.08	0.08	0.04
	0.8	0.00	0.00	0.00	0.00	0.03	0.03	0.00	0.10
	1.0	0.00	0.00	0.00	0.00	0.00	0.01	0.01	0.01
GA_GR	0.0	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	0.2	0.00	0.00	0.00	0.00	0.09	0.00	0.01	0.01
	0.4	0.00	0.00	0.00	0.00	0.12	0.39	0.14	0.01
	0.6	0.00	0.00	0.00	0.00	0.22	0.07	0.01	0.00
	0.8	0.00	0.00	0.00	0.00	0.05	0.00	0.00	0.00
	1.0	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
MA_IN	0.0	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	0.2	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	0.4	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	0.6	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	0.8	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	1.0	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
MA_GR	0.0	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	0.2	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	0.4	0.00	0.00	0.00	0.00	0.01	0.00	0.00	0.00
	0.6	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	0.8	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	1.0	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

Figure 1 Chromosome decoding example

Figure 2 Parameterized uniform crossover example

Figure 3 Evolutionary strategy

Figure 4 Genetic approach