Computers & Operations Research 70 (2016) 115–126

Contents lists available at ScienceDirect

Computers & Operations Research

journal homepage: www.elsevier.com/locate/caor

Metaheuristics for the single machine weighted quadratic tardiness scheduling problem

Tomás C. Gonçalves [a], Jorge M.S. Valente [b,*], Jeffrey E. Schaller [c]

[a] Faculdade de Economia da Universidade do Porto, C/O Jorge Valente, Rua Dr. Roberto Frias, s/n, 4200-464 Porto, Portugal
[b] LIAAD – INESCTEC LA, Faculdade de Economia da Universidade do Porto, Rua Dr. Roberto Frias, s/n, 4200-464 Porto, Portugal
[c] Department of Business Administration, Eastern Connecticut State University, 83 Windham St., Willimantic, CT 06226-2295, USA

ARTICLE INFO

Available online 12 January 2016

Keywords:
Scheduling
Single machine
Weighted quadratic tardiness
Metaheuristics
Iterated local search
Variable greedy
Genetic algorithm

ABSTRACT

This paper considers the single machine scheduling problem with weighted quadratic tardiness costs. Three metaheuristics are presented, namely iterated local search, variable greedy and steady-state genetic algorithm procedures. These address a gap in the existing literature, which includes branch-and-bound algorithms (which can provide optimal solutions for small problems only) and dispatching rules (which are efficient and capable of providing adequate solutions for even quite large instances). A simple local search procedure which incorporates problem specific information is also proposed.

The computational results show that the proposed metaheuristics clearly outperform the best of the existing procedures. Also, they provide an optimal solution for all (or nearly all, in the case of the variable greedy heuristic) the smaller size problems. The metaheuristics are quite close in what regards solution quality. Nevertheless, the iterated local search method provides the best solution, though at the expense of additional computational time. The exact opposite is true for the variable greedy procedure, while the genetic algorithm is a good all-around performer.

1. Introduction

This paper considers a single machine scheduling problem with weighted quadratic tardiness costs. Formally, the problem can be stated as follows. A set of n independent jobs $\{1,2,\cdots,n\}$ is to be scheduled on a single machine that can handle only one job at a time. The machine is continuously available from time zero onwards, and preemptions are not allowed. Job $j, j=1,2,\cdots,n$, requires a processing time p_j, has a weight w_j and should ideally be completed by its due date d_j. For a given schedule, the tardiness of job j is defined as $T_j=\max\{C_j-d_j;0\}$, 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 squared tardiness values $\sum_{j=1}^{n} w_j T_j^2$.

Single machine scheduling environments may appear to arise infrequently in practice. However, they actually occur in several practical settings. A specific example, arising in the chemical industry, is given in [1]. Scheduling models with a single machine are also useful for problems with multiple processors. Indeed, a single bottleneck machine is often the source of inefficiency in many production systems. Therefore, the performance of these systems will then depend mainly on the quality of the schedules generated for this single bottleneck processor. Moreover, the study of single machine problems provides results and insights that prove valuable for scheduling more complex settings, such as parallel machines, flow shops or even job shops.

The objective function considers squared tardiness costs. Tardiness is a widely used performance measure in scheduling, since tardy deliveries can result in contractual penalties, lost sales and loss of customer goodwill. A squared tardiness is used in this paper, instead of the more usual (in the literature) linear tardiness or maximum tardiness alternatives. Each of these three measures has its merits, and neither is intrinsically better. Indeed, every one of these criteria can be appropriate, depending on the setting and the goals and preferences of the decision maker.

A maximum tardiness criterion is adequate when the main objective is to prevent a quite large delay. As detailed in [2], however, maximum tardiness focuses on the job with the largest delay, and disregards the tardiness that might be incurred in all the other jobs. Thus, if the decision maker wishes to take into account all delays and all customers, measures such as linear or squared tardiness are preferable. The choice between linear or quadratic again depends on setting and preferences.

Under a linear tardiness, the distribution of the overall total tardiness is irrelevant. That is, a sequence in which only one or a few jobs are quite tardy is equivalent to another sequence where

* Corresponding author. Tel.: +351 225 571 100; fax: +351 225 505 050.
E-mail addresses: tomcabrita@hotmail.com (T.C. Gonçalves), jvalente@fep.up.pt (J.M.S. Valente), schallerj@easternct.edu (J.E. Schaller).

http://dx.doi.org/10.1016/j.cor.2016.01.004
0305-0548/

multiple jobs are only a little tardy, as long as the sum of the tardiness values is the same. A quadratic tardiness objective function, however, severely penalizes larges values of the tardiness, and will usually avoid schedules in which a single or only a few jobs contribute the majority of the cost, as described in more detail in [2].

In the same line, and as highlighted in [3,4], in linear tardiness the incremental penalty of a job does not change as the tardiness increases, so two jobs each one time unit late are equivalent to one job two units late. In contrast, under a squared tardiness measure, the incremental penalty of a job does increase as the tardiness increases, so one job two time units late incurs a larger cost than two jobs each one unit late. Furthermore, the loss function of Taguchi [5] proposes that a customer's dissatisfaction tends to increase quadratically with the tardiness, instead of linearly. Thus, a squared tardiness objective is relevant in practice; indeed, the scheduling methodology developed by Hoitomt et al. [3] used the quadratic tardiness objective and was actually implemented as part of a knowledge-based scheduling system at Pratt and Whitney.

The considered problem has previously been studied in [6,7]. Schaller and Valente [6] developed several dominance rules, as well as branch-and-bound procedures which incorporated these rules. Valente and Schaller [7], on the other hand, proposed and analyzed several efficient dispatching rules. To the best of our knowledge, only a limited number of other papers have considered a weighted quadratic tardiness objective function. Hoitomt et al. [3] developed a solution procedure based on lagrangean relaxation for parallel machines problems with simple precedence constraints, and demonstrated this procedure using three examples. Sun et al. [2] analyzed several heuristics for the problem with a single machine, release dates and sequence dependent setup times. Finally, a job shop scheduling problem with alternative processing plans was studied by Thomalla [4], who compared, on three small examples, a lagrangean relaxation based lower bound and heuristics with other methods.

The complexity of the single machine weighted quadratic tardiness problem is, again to the best of our knowledge, still open. However, and given existing complexity results, it seems most likely that the problem is hard. Indeed, the corresponding linear problem, i.e. the total weighted tardiness problem, is strongly NP-hard [8,9].

Two streams of research on single machine scheduling that are related to the considered problem are models with a quadratic performance measure and the total weighted tardiness problem. Among tardiness-related quadratic performance measures, the quadratic lateness problem has been studied by Gupta and Sen [10], Sen et al. [11], Su and Chang [12], Schaller [13] and Soroush [14,15]. Also, the linear earliness and squared tardiness problem was considered by Schaller [16], Valente [17–19], Valente and Schaller [20], and Behnamian and Zandieh [21]. The problem with both quadratic earliness and quadratic tardiness costs was studied by Valente and Alves [22], Valente and Moreira [23], Valente [24], Valente et al. [25], Singh et al. [26], Kianfar and Moslehi [27], and Vilà and Pereira [28]. A large number of papers have been published on the total weighted tardiness problem. Exact methods have been surveyed and compared in [29], and several heuristic methods were analyzed in [30]. Sen et al. [31] provide a more recent literature review of both exact and heuristic procedures for this linear problem.

This paper presents three metaheuristics, namely iterated local search, variable greedy and steady-state genetic algorithm procedures. These heuristics address a gap in the existing literature. Indeed, and as previously mentioned, the existing procedures consist of branch-and-bound algorithms, which can provide an optimal solution for small instances, and efficient dispatching rules, which are often the only heuristic approach capable of providing solutions, in reasonable time, for large problems. Metaheuristics are often quite valuable for medium sized instances, since they are usually able to provide high quality solutions (superior to those of dispatching rules) within acceptable computational times. A local search procedure, which is used in the metaheuristics, is also presented. This proposed local search is essentially an adjacent pairwise interchange procedure, which incorporates problem specific information.

The remainder of this paper is organized as follows. The local search procedure is described in Section 2. Section 3 presents the three proposed metaheuristics. The computational results are reported in Section 4. Finally, some concluding remarks are provided in Section 5.

2. Local search procedure

In this section, the proposed local search procedure is described. As previously mentioned, the local search is essentially an adjacent pairwise interchange improvement procedure. Therefore, after the application of the local search, no further improvement in the sequence is possible by swapping any pair of adjacent jobs. However, the procedure incorporates problem specific information. The pseudo-code for the proposed local search is given in Procedure 1. In this context, let i be a position in a sequence and $[i]$ be the job in position i.

Procedure 1. Local search procedure

1. Set $i=1$.
2. While $i<n$:
   2.1. If jobs $[i]$ and $[i+1]$ are early:
      2.1.1. If $d_{[i]}>d_{[i+1]}$:
         2.1.1.1. Swap jobs $[i]$ and $[i+1]$.
         2.1.1.2. If $i>1$, set $i=i-1$.
      2.1.2. Otherwise, set $i=i+1$.
   2.2. Else if jobs $[i]$ and $[i+1]$ are tardy:
      2.2.1. If $w_{[i]}(2T_{[i]}+1)p_{[i+1]}<w_{[i+1]}(2T_{[i+1]}+1)p_{[i]}$:
         2.2.1.1. If the objective function value is improved by swapping jobs $[i]$ and $[i+1]$:
            2.2.1.1.1. Swap jobs $[i]$ and $[i+1]$.
            2.2.1.1.2. If $i>1$, set $i=i-1$.
         2.2.1.2. Otherwise, set $i=i+1$.
      2.2.2. Otherwise, set $i=i+1$.
   2.3. Else if job $[i]$ is early and job $[i+1]$ is tardy:
      2.3.1. If $d_{[i]}\geq C_{[i+1]}$:
         2.3.1.1. Swap jobs $[i]$ and $[i+1]$.
         2.3.1.2. If $i>1$, set $i=i-1$.
      2.3.2. Else if $w_{[i]}(C_{[i+1]}-d_{[i]})^2<w_{[i+1]}(2T_{[i+1]}+1)p_{[i]}$:
         2.3.2.1. If the objective function value is improved by swapping jobs $[i]$ and $[i+1]$:
            2.3.2.1.1. Swap jobs $[i]$ and $[i+1]$.
            2.3.2.1.2. If $i>1$, set $i=i-1$.
         2.3.2.2. Otherwise, set $i=i+1$.
      2.3.3. Else, set $i=i+1$.
   2.4. Else, set $i=i+1$.

The procedure starts at the first position in the sequence, and stops when the final position is reached. At each iteration, the jobs at the current and next positions are analyzed. If the two jobs are swapped, the procedure backtracks one position when possible, since a new comparison can now be made. Otherwise, the procedure moves forward by one position.

There are four possibilities regarding the earliness/tardiness of the jobs at the current and next positions. More specifically, and in their current positions, those jobs can be both early, both tardy, early and tardy, and tardy and early.

Step 2.1 considers the situation in which both jobs are currently early. In this case, a swap is performed only when $d_{[i]} > d_{[i+1]}$, that is, if the first job has a larger due date. We remark that the objective function value is not affected by this move, since both jobs are early both before and after the swap. However, moving the job with the largest due date forward in the sequence may help in future iterations. Indeed, it might be possible to shift that job further into the sequence, thereby reducing the tardiness of other jobs.

In step 2.2, both jobs are currently tardy. In this case, a preliminary check on the usefulness of a swap is first performed in 2.2.1. More specifically, and as described in Schaller and Valente [6], $w_{[i]}(2T_{[i]}+1)p_{[i+1]}$ and $w_{[i+1]}(2T_{[i+1]}+1)p_{[i]}$ are a lower bound for the increase in the cost of job $[i]$ and an upper bound for the decrease in the cost of job $[i+1]$, respectively, if the swap is performed. Therefore, if the condition in 2.2 is not satisfied, no swap is performed, since it certainly would not improve the sequence, given that the lower bound on the cost increase is larger than the upper bound on the cost decrease. If the condition is satisfied, the procedure checks whether the swap indeed reduces the total cost, and performs the swap when the objective function is in fact improved.

Step 2.3 considers the situation in which the first job is early and the second is tardy. If job $[i]$ remains early after the swap, the exchange clearly reduces the cost, and the two jobs are swapped; this is handled in step 2.3.1 and its substeps. Otherwise, a preliminary check on the usefulness of a swap is then performed in 2.3.2. Again, $w_{[i+1]}(2T_{[i+1]}+1)p_{[i]}$ is an upper bound for the decrease in the cost of job $[i+1]$. The exact cost of job $[i+1]$ after the swap is given by $w_{[i]}(C_{[i+1]}-d_{[i]})^2$. Therefore, and once more, if the condition in 2.3.2 is not satisfied, no swap is performed, since it certainly would not improve the sequence. If the condition is satisfied, the procedure checks whether the swap reduces the total cost, and performs the swap when the objective function is indeed improved.

Finally, it is possible that the first job is tardy and the second is early. In this case, it is clear that a swap should not be performed, since it would increase the objective function. This situation is captured by step 2.4.

3. Metaheuristics

In this section, the proposed metaheuristics are presented. A pseudo-code is given for each procedure, as well as a description of the various components and implementation choices that were made.

3.1. Iterated local search

Local search or descent procedures terminate as soon as a local optimum is reached, since they are unable to escape from local optima. A simple approach that allows for the exploration of multiple local optima, with any local search algorithm, is to perform multiple runs, each starting from a different initial solution. The iterated local search (ILS) metaheuristic is based precisely on this approach.

In ILS, the next starting solution is obtained from the current solution (usually a local optimum), by applying some sort of prespecified type of random move to this current solution. This is known as perturbing, or *kicking*, the current solution. In this way, not all the good characteristics of a previously found solution are lost when generating the next starting solution, which is a serious concern when the initial solutions are fully randomly generated.

Some of the basic ideas in ILS were first described by Baxter [32], though the use of a randomized kick traces back to Baum [33,34]. Multiple studies [35–40] have shown that ILS can provide extremely competitive results. For a more detailed description of ILS, please see [35,38].

The pseudo-code for the proposed ILS implementation is given in Procedure 2. In the following, (S_{best}, ofv_{best}) denote the best solution found so far and its corresponding objective function value, respectively. Similarly, (S, ofv_S) and (S_k, ofv_k) provide the same information for the current solution and the kicked solution (that is, the solution obtained by performing a kick on the current solution), respectively.

Procedure 2. Iterated local search

1. Set $(S_{best}, ofv_{best})=(\varnothing, \infty)$.
2. (S, ofv_S)=Generate_Initial_Solution().
3. If Do_Local_Search(S) == TRUE, set (S, ofv_S)=Perform_Local_Search(S).
4. If $ofv_S < ofv_{best}$, set $(S_{best}, ofv_{best})=(S, ofv_S)$.
5. While stop criterion is not met:
   5.1. (S_k, ofv_k)=Perform_Kick(S).
   5.2. If Do_Local_Search(S_k) == TRUE, set (S_k, ofv_k)=Perform_Local_Search(S_k).
   5.3. If $ofv_k < ofv_{best}$, set $(S_{best}, ofv_{best})=(S_k, ofv_k)$.
   5.4. If Perform_Backtrack() == TRUE, set $(S, ofv_S)=(S_{best}, ofv_{best})$.
   5.5. Else, set $(S, ofv_S)=(S_k, ofv_k)$.

Step 1 simply initializes the procedure, by setting the best solution found so far and the respective objective function value to an empty sequence and infinity, respectively. The initial solution is then created in step 2. This solution is generated by the QBack_v6 dispatching rule, which provided the best results among the dispatching procedures proposed in [7].

Step 3, and similarly step 5.2, determines whether or not local search is to be applied to a solution, and performs the local search when appropriate. In the proposed implementation, local search is always applied when a solution is better than the best found so far. Otherwise, the local search is applied with a probability equal to a user defined parameter $0 \leq \text{ls\_prob} \leq 1$. When the local search is to be performed, the improvement procedure described in the previous section is then applied in order to try to improve the solution. Step 4, as well as step 5.3, updates the best solution found so far when appropriate.

In step 5, and its substeps, the algorithm iterates until a stopping criterion, identical for all proposed metaheuristics, is met. More specifically, the algorithm terminates either when a user defined maximum computation time max_rt is reached, or if a solution with an objective function value of 0 (which is clearly optimal) is found.

At each iteration, a new solution S_k is obtained by executing a kick on the current solution S (step 5.1). In this implementation, a kick consists in performing α random swaps (the swapped jobs need not be adjacent), where α is a user defined parameter. Steps 5.2 and 5.3, as previously described, are related with the local search and updating of the best solution found so far.

Finally, a new current solution is set in steps 5.4 and 5.5. If a so-called *backtrack* is to be performed, the current solution is set equal to the best solution found so far. Otherwise, the kicked solution becomes the new current solution. In the proposed implementation, a backtrack is performed when β consecutive

iterations have been performed without improving the best solution found so far, where β is a user defined parameter.

3.2. Variable greedy

The variable greedy (VG) algorithm is a metaheuristic that was developed recently by Framinan and Leisten [41]. This procedure combines features of two other metaheuristics, namely iterated greedy and variable neighborhood search.

Iterated greedy algorithms use a two-phase procedure, destruction followed by construction, to generate solutions. In the destruction phase, some components are removed from the current solution, yielding a partial solution. The next solution is then constructed by inserting those removed elements into the partial solution. Both phases are performed using a greedy procedure.

The variable neighborhood search metaheuristic, on the other hand, uses a finite set of preselected neighborhood structures, instead of a single neighborhood. The procedure switches from one neighborhood to the next when no improvement is made, in order to try to overcome local optima.

These two features are combined in the VG metaheuristic. Indeed, and more specifically, the concept of varying the neighborhood is applied to the destruction and construction phases. Thus, the number of components that are removed and then reinserted is variable. For a more detailed description of VG, please see [41]. Also, additional details on the iterated greedy and variable neighborhood search metaheuristics can be found in [42,43] and [44], respectively.

The pseudo-code for the proposed VG implementation is given in Procedure 3. The previously undescribed notation is defined as follows. Let k be the current number of components, which in this specific problem correspond to jobs, to be removed and then reinserted in the destruction and construction phases. Also, let k_{max} be the maximum allowed value for k, i.e. the maximum neighborhood size. In the following, (S_{dc}, ofv_{dc}) denote the solution that is obtained by applying destruction and construction to the current solution, and its corresponding objective function value, respectively. The set of jobs that are removed and then reinserted in order to obtain the solution S_{dc} from the current solution S is denoted by S_r. Finally, $0 \leq \text{ns\_prop} \leq 1$ is a user defined parameter.

Procedure 3. Variable greedy

1. Set $(S_{best}, ofv_{best})=(\varnothing, \infty)$, $k=1$ and $k_{max} =$ ns_prop $\times(n-1)$.
2. (S, ofv_S)=Generate_Initial_Solution().
3. If Do_Local_Search(S) == TRUE, set (S, ofv_S)=Perform_Local_Search (S).
4. If $ofv_S < ofv_{best}$, set $(S_{best}, ofv_{best})=(S, ofv_S)$.
5. While stop criterion is not met:
   5.1. S_{dc}=Perform_Destruction (S, S_r, k).
   5.2. (S_{dc}, ofv_{dc})=Perform_Construction ($S_{dc}S_r$,).
   5.3. If Do_Local_Search (S_{dc}) == TRUE, set (S_{dc}, ofv_{dc})= Perform_Local_Search (S_{dc}).
   5.4. If $ofv_{dc} < ofv_{best}$, set $(S_{best}, ofv_{best})=(S_{dc}, ofv_{dc})$.
   5.5. If $ofv_{dc} < ofv_S$:
      5.5.1. Set $(S, ofv_S)=(S_{dc}, ofv_{dc})$.
      5.5.2. Set $k=1$.
   5.6. Else:
      5.6.1. Set $k=k+1$.
      5.6.2. If $k \leq k_{max}$, set $(S, ofv_S)=(S_{dc}, ofv_{dc})$.
      5.6.3. Else:
         5.6.3.1. Set $k=1$.
         5.6.3.2. (S, ofv_S)=Generate_Random_Solution().
         5.6.3.3. If Do_Local_Search(S)==TRUE, set (S, ofv_S)= Perform_Local_Search(S).
         5.6.3.4. If $ofv_S < ofv_{best}$, set $(S_{best}, ofv_{best})=(S, ofv_S)$.

Step 1 again initializes the procedure. In addition to setting the best solution found so far and its objective function value, k is given an initial value of 1. Also, the maximum number of jobs to be removed and reinserted k_{max} is set equal to a proportion ns_prop of the maximum possible neighborhood size $(n-1)$.

In Framinan and Leisten [41], k_{max} was simply set equal to the maximum possible value $(n-1)$. In this implementation, the user defined parameter ns_prop is considered, in order to evaluate the trade-offs between solution quality and computational time that are likely to occur if a smaller maximum neighborhood size is chosen. The maximum neighborhood size k_{max} is then set equal to a proportion ns_prop of the maximum possible size.

Steps 2–4, which deal with the initial solution, local search and updating of the best variables, are then identical to those in the ILS procedure. In step 5, and its substeps, the algorithm again iterates until the stopping criterion is met.

At each iteration, a new solution S_{dc} is obtained by executing the two-phase destruction followed by construction procedure. In step 5.1 (destruction), S_{dc} is set equal to the partial solution that results from removing k jobs from the current solution$S$; those jobs are temporarily stored in S_r. Then, in step 5.2 (construction), the removed jobs are reinserted, yielding a complete solution.

In the chosen implementation, the destruction step removes the k jobs with the largest objective function values, with ties broken by selecting the job with the smallest due date. The construction step then reinserts these jobs one at a time, in decreasing order of those objective functions values, again with ties broken by giving preference to the smallest due date. The reinsertion procedure used for each job is based on the NEH heuristic [45]. More specifically, each job is inserted at all possible positions in the current partial solution, and the position that leads to the lowest objective function value (for the entire partial sequence) is then selected.

Steps 5.3 and 5.4 again deal with the possible application of the local search procedure and updating of the best variables. The next current solution and the value of k are then updated in steps 5.5 and 5.6. If the new solution S_{dc} is better than the current solution S, solution S_{dc} becomes the new current solution, and the size of the destruction and construction neighborhood k is reset to the minimum value of 1 (step 5.5 and its substeps). Otherwise (step 5.6 and its substeps), k is first increased by 1 in step 5.6.1, thereby enlarging the neighborhood.

If the new neighborhood size k is not larger than the limit k_{max}, solution S_{dc} again becomes the new current solution (step 5.6.2). However, if the maximum neighborhood size k_{max} is exceeded (step 5.6.3 and its substeps), k is reset to the minimum value of 1 and a new current solution is generated at random. The local search procedure is then possibly applied to this solution, and the best variables are updated when appropriate.

3.3. Steady-state genetic algorithm

The term genetic algorithm was first used by Holland [46], and though his work placed little emphasis on optimization, the majority of the research on genetic algorithms has indeed since been focused on solving optimization problems. Genetic algorithms are population-based metaheuristics, since at each iteration (or generation, in the genetic algorithm terminology) they consider a set (population) of solutions (chromosomes), instead of a single current solution.

In a genetic algorithm, an initial population is first generated. Successive populations are then evolved by mimicking the evolution process that occurs in natural biology. More specifically, new solutions are usually obtained via reproduction/crossover, which combines information from two current "parent" solutions in order to obtain an "offspring" chromosome, and/or through mutation, which creates a chromosome by changing an existing solution.

A steady-state genetic algorithm (SSGA), using a permutation encoding, is considered in this paper. In the steady-state population replacement method, at each iteration a single solution is generated, via crossover and/or mutation, and might eventually replace the current least fit member of the population. This is in contrast with other evolutionary strategies, such as the generational approach, which replaces the entire population at each iteration, and the elitist strategy, which replaces all except a certain number of the current best solutions. With the steady-state generational scheme, a new solution becomes immediately available for crossover and/or mutation, while in the other strategies it will only become available once the entire new population is generated. In the permutation encoding, each chromosome is represented by a permutation of integers. This is appropriate for the considered problem, since a solution can be quite naturally represented by a permutation of the jobs' indexes, representing the order in which these jobs are processed. For a more detailed description of genetic algorithm and their components and strategies, please see [47–49].

The pseudo-code for the proposed SSGA implementation is given in Procedure 4. The previously undescribed notation is defined as follows. In the following, (S_{worst}, ofv_{worst}) denote the worst solution in the current population and its corresponding objective function value, respectively. Similarly, (S_1, ofv_1) and (S_2, ofv_2) are solutions used in the crossover and/or mutation operations and their respective objective functions values. Also, (S_{cm}, ofv_{cm}) provide the same information for the offspring solution generated via crossover or mutation. Finally, let *pop* denote the set of solutions/chromosomes in the current population.

Procedure 4. Steady-state genetic algorithm

1. Set $(S_{best}, ofv_{best})=(\varnothing, \infty)$, $(S_{worst}, ofv_{worst})=(\varnothing, -\infty)$ and $pop=\varnothing$.
2. pop=Generate_Initial_Population().
3. While stop criterion is not met:
   3.1. If Do_Crossover() == TRUE:
      3.1.1. S_1=Select_Parent(pop).
      3.1.2. S_2=Select_Parent($pop \setminus S_1$).
      3.1.3. S_{cm}=Perform_Crossover(S_1,S_2).
   3.2. Else:
      3.2.1. S_1=Select_Chrom_Mutation(pop).
      3.2.2. S_{cm}=Perform_Mutation(S_1).
   3.3. If Do_Local_Search(S_{cm})==TRUE, set (S_{cm}, ofv_{cm})=Perform_Local_Search(S_{cm}).
   3.4. If $ofv_{cm} < ofv_{best}$, set $(S_{best}, ofv_{best})=(S_{cm}, ofv_{cm})$.
   3.5. If $(S_{cm} \notin pop)$ AND $(ofv_{cm} < ofv_{worst})$:
      3.5.1. Replace S_{worst} with S_{cm}.
      3.5.2. Update (S_{worst}, ofv_{worst})

A general outline of the proposed SSGA is first given. Then, some components and choices are described in more detail. Step 1 again initializes the procedure, by appropriately setting the best and worst solutions found so far, and creating a so far empty population. The initial population is then generated in step 2; the specifics are described below. In step 3, and its substeps, the algorithm then iterates until the stopping criterion is met.

At each iteration, a single chromosome is generated via crossover (step 3.1 and its substeps) or mutation (step 3.2 and its substeps). The crossover operation is chosen with a probability equal to a user defined parameter $0 \leq$ cross_prob ≤ 1. Thus, function Do_Crossover() will return TRUE with probability cross_prob.

When the new solution is obtained via crossover, two different parent solutions are first selected from the population. The new solution is then generated by performing a crossover operation on the chosen parents. Otherwise, a single solution is selected from the population, and the new solution is then obtained via a mutation operation. Details concerning the selection of solutions and the crossover and mutation operators are provided below.

Steps 3.3 and 3.4 deal with the possible application of the local search procedure and updating of the best variables. The replacement strategy is embodied in step 3.5 and its substeps. More specifically, the new solution replaces the current worst member of the population if it is unique (that is, no identical solution is present in the current population) and better than the current worst solution.

The details concerning the initial population and the crossover and mutation operations will now be presented. The pseudo-code for the generation of the initial population is given in Procedure 5. The previously undescribed notation is defined as follows. In the following, (S_{ip}, ofv_{ip}) denote a solution created during the generation of the first population and its corresponding objective function value, respectively. Also, let pop_size be a user defined parameter that corresponds to the desired population size. Finally, init_pop_seeded $\in$ {TRUE, FALSE} is a user defined parameter that indicates whether or not a non-random solution is to be inserted into the initial population. The pseudo-code for function Shift_Early_Jobs_Forward, which used in the generation of the initial population, is given in procedure 6.

Procedure 5. Function Generate_Initial_Population()

1. If init_pop_seeded == TRUE:
   1.1. S_{ip}=Generate_Seed_Solution().
   1.2. If Do_Local_Search(S_{ip}) == TRUE, set (S_{ip}, ofv_{ip})=Perform_Local_Search(S_{ip}).
   1.3. Set $pop = pop \cup S_{ip}$ and update (S_{best}, ofv_{best}) and (S_{worst}, ofv_{worst}).
2. If $ofv_{best}=0$, RETURN.
3. Set i_count=0.
4. While (#$pop <$ pop_size) OR (i_count $< 3 \times$ pop_size):
   4.1. S_{ip}=Generate_Random_Solution().
   4.2. S_{ip}=Shift_Early_Jobs_Forward(S_{ip}).
   4.3. If Do_Local_Search(S_{ip})==TRUE, set (S_{ip}, ofv_{ip})=Perform_Local_Search(S_{ip}).
   4.4. If $S_{ip} \notin pop$, set $pop = pop \cup S_{ip}$ and update (S_{best}, ofv_{best}) or (S_{worst}, ofv_{worst}) when appropriate.
   4.5. Set i_count=i_count+1.
   4.6. If $ofv_{best}=0$, BREAK.
5. If $ofv_{best}=0$, RETURN.
6. While (#$pop <$ pop_size):
   6.1. S_{ip}=Generate_Random_Solution().
   6.2. If $S_{ip} \notin pop$, set $pop = pop \cup S_{ip}$ and update (S_{best}, ofv_{best}) or (S_{worst}, ofv_{worst}) as appropriate.
   6.3. If $ofv_{best}=0$, BREAK.

Procedure 6. Function Shift_Early_Jobs_Forward

1. Set $i = 1$.
2. While $i < n$:
   2.1. If $(d_{[i]} \geq C_{[i+1]})$ AND $(d_{[i]} > d_{[i+1]})$:
      2.1.1. Swap jobs $[i]$ and $[i+1]$.
      2.1.2. If $i > 1$, set $i = i-1$.
   2.2. Otherwise, set $i = i+1$.

Step 1 in Generate_Initial_Population() checks whether the initial population is to be seeded, that is, if a nonrandom solution generated by a heuristic procedure is to be inserted in the population. When seeding is to be performed, the solution given by the QBack_v6 dispatching rule is generated in step 1.1. Step 1.2 then deals with the application of the local search procedure. In step 1.3, the solution is added to the population, and the best and worst variables are updated.

If a solution with an objective function of 0 (which, as previously mentioned, is clearly optimal) is obtained, the generation of the initial population is stopped in step 2. Otherwise, step 4 (and its substeps) then iterates until the initial population has been fully generated, or a maximum of $3 \times$ pop_size iterations have been performed; the rationale for this maximum number of iterations is described below.

A random solution is first generated in step 4.1. Step 4.2 tries to improve this solution by applying the Shift_Early_Jobs_Forward procedure. This function, as can be seen by the pseudo-code given in Procedure 6, is essentially a greatly simplified version of the local search procedure. As implied by the name, this function simply moves early jobs forward in the sequence as much as possible without those jobs becoming late. Indeed, as indicated in step 2.1, a job is swapped with the next when it is still early if scheduled at the next position, and its due date is larger than that of the following job. The local search procedure is then possibly applied in step 4.3. The solution is added to the population if it is unique, and the best or worst variables are updated when appropriate. If an optimal solution with an objective function value of 0 is generated, step 4 is immediately terminated.

The Shift_Early_Jobs_Forward procedure was introduced in order to try to prevent the generation of a (quite) poor initial population. Indeed, if this procedure was not used, and in the presence of a low user defined ls_prob local search probability, the initial population would consist mostly or fully of randomly generated solutions, which can be of very low quality. A quite poor initial population can have a negative effect on the performance of a genetic algorithm, which can then require a large number of iterations in order to generate good solutions.

We recall that the VG procedure also uses randomly generated solutions. However, this metaheuristic has a much stronger intensification component, since the destruction and construction phases themselves act as a sort of local search. Therefore, the Shift_Early_Jobs_Forward is not really required, and the randomness is actually needed for diversification and to act as a counterbalance of the intensification provided by the destruction and construction steps. In a genetic algorithm, however, crossover and mutation operations would take far more time to evolve good solutions from a poor initial population.

As previously mentioned, step 4 terminates if a maximum of $3 \times$ pop_size iterations have been performed without successfully filling the initial population. This limit is required in order to avoid, in some instances, a quite large number of iterations, or even an endless loop. Indeed, in some instances with loose due dates, the application of the Shift_Early_Jobs_Forward procedure may lead systematically to a single or a few solutions in which only a small number of jobs is tardy. The best of these solutions is likely to be optimal; however, since its objective function value is positive (since at least one job is tardy), it is not possible to guarantee that optimality.

Therefore, in these cases it might take an extremely large number of iterations, or actually be impossible at all, to fill the initial population in step 4 and its substeps. Consequently, and after some experimental tests, the limit of $3 \times$ pop_size iterations was imposed. When step 4 indeed fails to fill the initial population, the remaining solutions are generated at random in step 6 and its substeps. Again, a solution is only added to the population if it is unique, and step 6 is terminated should an optimal solution with an objective function value of 0 be generated.

The details concerning the crossover operation will now be described. Each parent is selected via a probabilistic binary tournament [48]. Thus, in order to choose each parent, two different candidate solutions C_1 and C_2 are first selected at random. Then, one of these is chosen probabilistically. In the proposed implementation, the selection probability is proportional to the solution quality. Therefore, the probability of selecting solution C_1 is equal to $ofv_{C_2}/(ofv_{C_1}+ofv_{C_2})$, where ofv_{C_1} and ofv_{C_2} are the objective function values of C_1 and C_2, respectively.

A unique offspring solution is then obtained via a uniform order based (UOB) crossover [47]. The UOB crossover considers each position in the sequence sequentially, and the corresponding job in the first parent is copied to the offspring with a probability p1_copy_prob. The vacant positions are then filled with the missing jobs, in the order in which they appear in the second parent. In the chosen implementation, p1_copy_prob is set equal to $ofv_{C_2}/(ofv_{C_1}+ofv_{C_2})$. Thus, the probability is again proportional to solution quality, and the offspring will tend to have more positions copied from the best parent [50].

The application of the UOB crossover is illustrated in Fig. 1. Let rand_gen01() be a function that returns a random uniform number between 0 and 1. The random number generated for positions 1, 3, 4 and 6 is lower than p1_copy_prob, and therefore the corresponding jobs in the offspring are inherited from the first parent. Jobs 2 and 5 are still missing from the sequence, and these are then inserted in the empty positions, in the order in which they appear in the second parent.

Finally, the details concerning the mutation operation are now presented. A solution is first chosen at random, and a mutated chromosome is then generated from this solution via a gene by gene mutation procedure. In gene by gene mutation, each position in the chromosome is involved in a random move with a given probability. The pseudo-code for the mutation function is given in Procedure 7. The previously undescribed notation is defined as follows. Let $0 \leq$ gene_mut_prob ≤ 1 be a user defined parameter that corresponds to the probability of each position being affected by a random move. Also, mut_type $\in$ {INT, INS} is a user defined parameter that indicates whether the random move consists of an interchange (INT) or an insertion (INS) operation.

Procedure 7. Function Perform_Mutation()

1. Set $i = 1$.
2. While $i < n$:

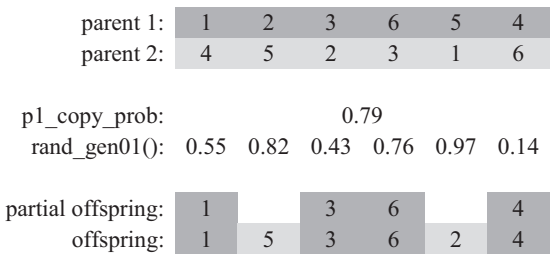

Fig. 1. Crossover example.

2.1. If rand_gen01() < gene_mut_prob:
2.1.3. Randomly select a position $p \neq i$.
2.1.2. If (mut_type == INT), swap jobs [i] and [p].
2.1.3. Otherwise, remove job [i] from its current position and insert it at position p.
2.2. Set $i = i+1$.

In step 2, each position in the sequence is considered sequentially. For each position i, there is a gene_mut_prob probability that the job in the position will be involved in a random move. When a random move is to be performed (step 2.1 and its substeps), a different position p is generated at random. If the random move is of the interchange type, the jobs in the two positions are swapped. Otherwise, the job in position i is removed from its current position and reinserted at position p.

4. Computational results

In this section, the computational experiments and results are presented. First, the set of test problems used to obtain the computational results is described. Next, the preliminary tests that were performed in order to determine adequate values for the parameters required by the metaheuristics are presented.

The proposed metaheuristics are then compared with existing procedures and solutions. The QBack_v6 dispatching rule, which provided the best results among the dispatching procedures proposed in [7], is naturally included in this set of comparison procedures. Also considered is this same dispatching rule, but followed by the application of the local search improvement procedure. In the following, let DR and DR+LS denote the QBack_v6 dispatching rule and this same heuristic followed by the application of the local search procedure, respectively.

A set of some of the heuristics proposed by Vilà and Pereira [28] was also considered for comparison purposes. These procedures were developed for the problem with both weighted quadratic earliness and weighted quadratic tardiness costs. However, they can also be applied to instances with only tardiness costs by simply setting the earliness weight h_j equal to 0 for all jobs, thus effectively disregarding earliness.

In the following, VP will denote a procedure that executes four of the heuristics proposed in [28], and returns the best solution found. The four heuristics included in VP, and according to the notation used in [28], are: 1) a NEH-based heuristic, followed by a local search; 2) APh followed by a local search; 3) APs followed by a local search; and 4) APdp(t). The APh and APs heuristics use solutions given by a partition-based lower bound obtained by solving an assignment problem, while the APdp(t) procedure employs dynamic programming together with precedence relations derived from a time-decomposition lower bound obtained by solving another assignment problem. For further details concerning these procedures, please see Vilà and Pereira [28].

The proposed metaheuristics and the existing procedures were compared not only amongst themselves, but also with optimal solutions for small problem sizes. Finally, additional information is given regarding the relative performance of the metaheuristic procedures.

4.1. Experimental design

The computational tests were performed on a set of randomly generated problems with 10, 15, 20, 25, 30, 40, 50, 75, 100, 250 and 500 jobs. The approach used to generate the problems was the same as the one that was used to create the linear weighted tardiness problem instances available in the OR-Library (http://people.brunel.ac.uk/~mastjjb/jeb/orlib/wtinfo.html).

More specifically, the problems were generated as follows. For each job j, an integer processing time p_j was generated from a uniform distribution between 1 and 100, while an integer weight w_j was obtained from a uniform distribution between 1 and 10. Also, an integer due date d_j was generated from the uniform distribution $\left[P\left(1-T-\frac{R}{2}\right), P\left(1-T+\frac{R}{2}\right)\right]$, where P is the sum of the processing times of all jobs, T is the tardiness factor and R is the range of due dates. Both the tardiness factor and the range of due dates parameters were set at 0.2, 0.4, 0.6, 0.8 and 1.0. This approach to the generation of the due dates is the same as the one used to create the OR-Library instances, as previously mentioned, and has also consistently been used in the literature since early papers [30,51,52] on problems with tardiness (or earliness and tardiness) criteria.

For each combination of problem size n, T and R, 50 instances were randomly generated. Therefore, a total of 1250 instances were generated for each problem size. The metaheuristics, as well as the DR and DR+LS procedures, were coded in C++ and executed on a personal computer with an Intel Core 2 Quad Q6600 2.40 GHz processor. For each instance, 10 independent runs, with different random number seeds, were performed for the ILS, VG and SSGA metaheuristics.

The VP procedure was run by professors Vilà and Pereira, in their current computing facilities. This method was executed on instances with additional earliness penalties, all set equal to 0, as previously mentioned. The heuristics included in the VP procedure were coded in C++, and this procedure was executed on a computer with an Intel Xeon E5–2670 2.60 GHz processor. The time limit on the resolution of the assignment problem included in APdp(t) was set to 3600 seconds. We remark that the use of different machines for the metaheuristics and DR and DR+LS procedures, on the one hand, and the VP algorithm, on the other hand, was necessary since they were compiled for and executed under different operating systems (Windows in the case of the former and Linux/Mac for the latter).

4.2. Parameter adjustment tests

Extensive preliminary tests were conducted to determine adequate values for the various parameters required by the proposed metaheuristics. These experiments were performed on a separate problem set, which included instances with 10, 25, 50, 100, 250 and 500 jobs, and contained 5 instances for each combination of n, T and R. Also, only a single run was performed for each metaheuristic on each instance.

Table 1 provides the values that were considered for each parameter; the chosen value is both in **bold and underlined**. We remark that some other alternatives were investigated regarding

Table 1
Metaheuristic parameter values.

Heur	Parameter	Values
All	max_rt	$0.5+0.0001 \times n^2$
ILS	α	**5**, 6, 7
	β	**5**, 10, 20, 25
	ls_prob	0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, **1.0**
VG	ns_prop	0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, **0.9**, 1.0
	ls_prob	0.0, 0.1, 0.2, 0.3, 0.4, 0.5, **0.6**, 0.7, 0.8, 0.9, 1.0
SSGA	pop_size	30, 40, **50**
	init_pop_seeded	**TRUE**, FALSE
	cross_prob	0.85, **0.90**, 0.95, 0.975
	mut_type	**INT**, INS
	gene_mut_prob	0.02, 0.03, **0.04**
	ls_prob	0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, **0.9**, 1.0

the maximum allowed runtime max_rt, though they were also all simple quadratic functions of the problem size. For the sake of space and brevity, these are omitted.

The chosen parameter values were selected by a thorough analysis of both the objective function values and the runtime until the best solution was found. It should be noted that, given the allotted maximum runtime, the procedures were quite robust in regard to solution quality. Indeed, and with the exception of setting ls_prob=0, which yielded worse solutions, the various parameter combinations provided similar results. The main difference, and therefore the major criterion in choosing the parameter values, was in the time required to reach that solution. Thus, the selected values are essentially the ones that allowed the algorithms to reach their best solution faster.

4.3. Comparison with existing results

In this section, the solutions provided by the various heuristics (proposed and existing) are compared amongst themselves, on all instance sizes, as well as with optimal results, for instances with up to 50 jobs. These optimum results were obtained using the previously mentioned branch-and-bound algorithms developed by Schaller and Valente [6]. As described in [6], the branch-and-bound algorithms are usually not able to solve larger instances within reasonable computational times.

Table 2 provides the mean relative improvement the optimum objective function value provides over the heuristic solution (ivh), as well as the number of times an optimal solution was generated by the heuristic (n_opt). For a given instance, the relative improvement of the optimum objective function value versus heuristic H_i is calculated as follows. Let ofv_{opt} and ofv_{H_i} denote the optimum and heuristic objective function values, respectively, where H_i={DR, DR+LS, VP, ILS, VG, SSGA}. When $ofv_{H_i}=0$, the relative improvement versus the heuristic solution is set at 0. Otherwise, the relative improvement provided by the optimum is calculated as $\left(ofv_{H_i}-ofv_{opt}\right)/ofv_{H_i}\times 100$. The ivh data in Table 2 illustrates the average performance of the ILS, VG and SSGA metaheuristics, since the average objective function value over the 10 runs was considered in the calculation of this indicator.

We remark that the relative improvement provided by the optimum when compared with the heuristic solution was chosen over the more usual relative deviation from the optimum, which is given by $\left(ofv_{H_i}-ofv_{opt}\right)/ofv_{opt}\times 100$. This is in line, and due to the same reasons, with the analysis performed in Valente and Schaller [7], concerning the evaluation of dispatching rules for the same problem. Indeed, and as described in [7], the optimum objective function values can be equal to 0 for instances with low T and high R. Thus, for such instances the relative deviation from the optimum would be undefined, thereby motivating the use of the relative improvement versus the heuristic result.

The results in Table 2 show that the simple dispatching rule itself performs quite well (as indicated in [7]), and its performance is further improved by the application of the local search procedure. Indeed, the mean relative improvement provided by the optimum is below 0.8%, and even for instances with 50 jobs the DR+LS procedure provides an optimal solution for over 50% of the instances. However, it should be noted that the performance of both the DR and DR+LS procedures deteriorates as the instance size increases.

The VP heuristic fails to match the performance of the DR+LS procedure. The number of optimal solutions it provides is quite similar to that of the DR+LS procedure. However, the non-optimal results it provides are further away from the optimal solutions, as shown by the mean relative improvement provided by the optimum. Indeed, while this measure is always below 0.7% for the DR+LS procedure, it is instead always higher than 1.2% for the VP heuristic, and it is larger than 2% for instances with 40 and 50 jobs.

The metaheuristics necessarily provide a schedule that is at least as good as the one given by DR+LS, since their first solution is generated precisely by the QBack_v6 dispatching rule and then improved by the local search procedure. Table 2 shows that the metaheuristics, particularly the ILS and SSGA procedures, actually do improve, when possible, over the DR+LS results. Indeed, the ILS and SSGA metaheuristics provide an optimal solution for all but a single or a couple of instances. The VG procedure also achieves the optimum for a quite large number of instances, and its mean relative improvement provided by the optimum solution is quite low, and clearly superior to that of the DR+LS heuristic.

The difference in performance is, however, much higher for some types of instance. This is clearly illustrated in Table 3, which provides the effect of the T and R parameters, for instances with 50 jobs. This table shows that the tardiness factor T and, although to a lesser degree, the range of due dates R, have a significant effect on the relative performance of the heuristics. For instances with a large tardiness factor, most jobs will be tardy, and even the simple dispatching rule provides results that are quite close to the optimum. However, the DR+LS heuristic still fails to achieve an optimal solution for several of these instances, while the ILS and SSGA metaheuristics always provide the optimum schedule, while the VG procedure is nearly always optimal.

The relative difference in performance is, however, much wider for instances with a medium and low value of T, i.e. instances where the number of tardy jobs is not as high. This is particularly true for instances with the lowest value of both T and R (T=0.2, R=0.2), with a tardiness factor of 0.4 and an intermediate value of R, and with T=0.6 and a high range of due dates. For these instances, the mean relative improvement provided by the optimum is usually over 1% (3%) for the DR+LS (VP) procedure, and in some cases is close or higher than 3% (10%). The metaheuristics, on the other hand, are again (nearly) always optimal.

The metaheuristics and the VP algorithm will now be compared with the DR+LS procedure. Table 4 provides, for the metaheuristics, the mean relative improvement over the solution generated by DR+LS heuristic (ivdr_ls), as well as the number of times they

Table 2
Comparison with optimum results.

	ivh						n_opt					
n	DR	DR+LS	VP	ILS	VG	SSGA	DR	DR+LS	VP	ILS	VG	SSGA
10	0.5142	0.2626	1.2312	0.0000	0.0000	0.0000	836	1117	1130	1250.0	1250.0	1250.0
15	0.5904	0.3906	1.5570	0.0000	0.0000	0.0000	670	1036	1021	1250.0	1250.0	1250.0
20	0.6522	0.4660	1.9383	0.0000	0.0000	0.0000	531	935	968	1250.0	1249.8	1250.0
25	0.7801	0.6217	1.9086	0.0000	0.0002	0.0000	485	882	880	1250.0	1248.2	1250.0
30	0.7026	0.5552	1.8814	0.0000	0.0034	0.0000	441	849	822	1250.0	1237.5	1250.0
40	0.6591	0.5559	2.0882	0.0080	0.0196	0.0080	372	744	739	1249.0	1210.0	1249.0
50	0.7388	0.6453	2.4347	0.0000	0.0210	0.0000	330	680	690	1250.0	1142.7	1249.8

provided a better (n_btr_dr_ls) or equal solution (n_eql_dr_ls). The number of times the DR+LS procedure generated an optimal solution (n_opt_dr_ls) is also repeated for convenience. Again, the mean relative improvement over the DR+LS solution is set at 0 when a metaheuristic and the DR+LS procedure provide identical objective function values. Otherwise, ivdr_ls is calculated as $(ofv_{DR+LS} - ofv_{H_i})/ofv_{DR+LS} \times 100$. For each metaheuristic, the objective function value ofv_{H_i} is again the average over the 10 runs.

In what regards the VP procedure, Table 4 gives the number of times it provided a solution that was better, equal and worse (n_wrs_dr_ls) than the one given by DR+LS. As previously mentioned, the VP heuristic fails to match the performance of the DR+LS procedure. Therefore, in terms of the relative improvement comparison between VP and DR+LS, the reverse measure is reported in Table 4. More specifically, this table gives the mean relative improvement the DR+LS heuristic provides over the solution generated by the VP procedure (drls_iv). Indeed, calculating ivdr_ls for the VP procedure would sometimes yield quite large negative numbers, corresponding to instances with a relatively low objective function value, but where the DR+LS procedure provided a solution with a cost much lower (in relative terms) than that of the VP heuristic. Thus, the drls_iv measure was selected for VP, to avoid those quite large negative numbers which would skew the mean.

The results in Table 4 show that the metaheuristics clearly outperform the DR+LS procedure. Though the relative improvement is only about 0.5–0.6%, such a value is consistent with the performance of the DR+LS heuristic, and in line with the results in Table 2. Therefore, the relative improvement provided by the metaheuristics is similar to that of the optimal solutions. Furthermore, and in line with the data presented in Table 3, the relative improvement is actually much larger for instances with a medium and low value of T, particularly those with (T=0.2,

Table 3
Comparison with optimum results for instances with 50 jobs.

		ivh						n_opt					
T	R	DR	DR+LS	VP	ILS	VG	SSGA	DR	DR+LS	VP	ILS	VG	SSGA
0.2	0.2	3.8941	3.6924	3.5117	0.0000	0.1884	0.0000	22	27	27	50.0	45.0	50.0
	0.4	0.1950	0.0000	0.0000	0.0000	0.0000	0.0000	48	50	50	50.0	50.0	50.0
	0.6	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	50	50	50	50.0	50.0	50.0
	0.8	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	50	50	50	50.0	50.0	50.0
	1.0	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	50	50	50	50.0	50.0	50.0
0.4	0.2	0.9732	0.8626	3.7780	0.0000	0.1706	0.0000	5	18	10	50.0	31.0	50.0
	0.4	3.0774	2.8580	5.1724	0.0000	0.0372	0.0000	3	14	9	50.0	40.9	50.0
	0.6	2.8745	2.5340	4.9816	0.0000	0.0208	0.0000	13	30	14	50.0	47.7	50.0
	0.8	1.7410	1.3009	10.2261	0.0000	0.0000	0.0000	33	43	29	50.0	50.0	50.0
	1.0	0.9766	0.6866	16.5313	0.0000	0.0000	0.0000	40	45	35	50.0	50.0	50.0
0.6	0.2	0.3319	0.2497	3.3026	0.0000	0.0352	0.0000	1	16	3	50.0	32.8	50.0
	0.4	0.5808	0.5161	4.1741	0.0000	0.0385	0.0000	1	10	1	50.0	38.3	50.0
	0.6	0.6097	0.5492	3.7023	0.0000	0.0110	0.0001	3	15	4	50.0	41.6	49.8
	0.8	1.8316	1.7473	1.8387	0.0000	0.0078	0.0000	2	13	14	50.0	43.1	50.0
	1.0	0.9522	0.8292	1.1302	0.0000	0.0144	0.0000	1	14	21	50.0	39.4	50.0
0.8	0.2	0.0827	0.0643	1.4210	0.0000	0.0016	0.0000	0	16	5	50.0	40.0	50.0
	0.4	0.1020	0.0828	0.8567	0.0000	0.0002	0.0000	0	10	19	50.0	47.5	50.0
	0.6	0.0477	0.0304	0.0949	0.0000	0.0002	0.0000	0	17	31	50.0	48.7	50.0
	0.8	0.0896	0.0714	0.0773	0.0000	0.0000	0.0000	0	17	26	50.0	49.4	50.0
	1.0	0.0649	0.0434	0.0519	0.0000	0.0001	0.0000	3	22	30	50.0	48.9	50.0
1.0	0.2	0.0116	0.0030	0.0038	0.0000	0.0000	0.0000	0	25	44	50.0	50.0	50.0
	0.4	0.0085	0.0024	0.0048	0.0000	0.0000	0.0000	3	35	36	50.0	49.0	50.0
	0.6	0.0104	0.0041	0.0048	0.0000	0.0000	0.0000	0	28	41	50.0	49.8	50.0
	0.8	0.0085	0.0041	0.0014	0.0000	0.0000	0.0000	1	32	47	50.0	49.6	50.0
	1.0	0.0059	0.0017	0.0013	0.0000	0.0000	0.0000	1	33	44	50.0	50.0	50.0

Table 4
Comparison with dispatching rule+local search results.

	dr_ls_iv	ivdr_ls_avg				n_btr_dr_ls			n_eql_dr_ls					n_wrs_dr_ls
n	VP	ILS	VG	SSGA	VP	ILS	VG	SSGA	VP	ILS	VG	SSGA	n_opt_dr_ls	VP
10	0.9523	0.2626	0.2626	0.2626	117	133.0	133.0	133.0	1026	1117.0	1117.0	1117.0	1117	107
15	1.1160	0.3906	0.3906	0.3906	167	214.0	214.0	214.0	885	1036.0	1036.0	1036.0	1036	198
20	1.4165	0.4660	0.4659	0.4660	252	315.0	314.8	315.0	758	935.0	935.2	935.0	935	240
25	1.2135	0.6217	0.6215	0.6217	266	368.0	367.8	368.0	678	882.0	882.2	882.0	882	306
30	1.2582	0.5552	0.5520	0.5552	271	401.0	397.6	401.0	631	849.0	852.4	849.0	849	348
40	1.4888	0.5478	0.5370	0.5478	295	506.0	492.4	506.0	534	744.0	757.6	744.0	744	421
50	1.7160	0.6453	0.6250	0.6453	306	570.0	531.3	570.0	488	680.0	718.7	680.0	680	456
75	1.8803	0.6196	0.5849	0.6196	337	719.0	634.2	719.0	375	531.0	615.8	531.0	–	538
100	1.2884	0.6521	0.5981	0.6521	342	788.0	668.4	788.0	354	462.0	581.6	462.0	–	554
250	1.0449	0.6023	0.5605	0.6022	340	937.9	800.8	937.6	289	312.1	449.2	312.4	–	621
500	0.9005	0.5126	0.4774	0.5122	288	959.0	856.2	956.2	284	291.0	393.8	293.8	–	678

$R=0.2$), or with a tardiness factor of 0.4 and an intermediate value of R. The number of instances for which a better solution is found is increasing in the problem size. For the largest instances, the ILS and SSGA heuristics improve upon the DR+LS solution in over 75% of the cases.

As previously mentioned, the metaheuristics provide an optimal solution for (nearly) all of the instances with up to 50 jobs, thus improving over the DR+LS procedure whenever possible. For the larger problem sizes, a significant number of the instances on which no improvement is achieved correspond to situations where the DR+LS heuristic already generated an optimal solution with an objective function value of 0. Indeed, and as previously mentioned, that is the case for about 200 instances for problems with 250 or 500 jobs. Thus, the data seem to provide evidence that the metaheuristics improve upon the DR+LS nearly as often as possible, and by nearly as much as possible.

The VP procedure, on the other hand, is outperformed by the DR+LS heuristic. Indeed, the DR+LS procedure usually provides a relative improvement over 1%, and gives better solutions for a larger number of instances. It should be remarked that the best solution generated by the VP heuristic was never achieved via the APdp(t) procedure, which provided the best performance among those in Vilà and Pereira [28], but instead through one of the other three simpler methods.

The APdp(t) heuristic achieved quite small optimality gaps for the problem with both quadratic earliness and tardiness costs [28]; however, it is usually unable to improve on the solution given by even the DR+LS procedure for the problem with only quadratic tardiness costs. This major difference in performance is probably due, for the most part, to the quite different nature and structure of the two problems. Indeed, when both earliness and tardiness are considered, all or nearly all jobs will have a positive cost, regardless of the position in which they are scheduled. The same is not true under a tardiness objective function: most jobs in earlier positions will tend to be early, and therefore have a cost of 0.

This is quite relevant, since APdp(t) and most of the other heuristics included in the VP procedure require solving an assignment problem. When jobs have positive costs regardless of their position in the sequence, the assignment problem can better differentiate between the various assignment possibilities. However, under a tardiness objective function, a large number of the coefficients involved in the assignment problem will be equal to 0 (corresponding to jobs assigned to positions were they will be early, or on time). This complicates matters for the assignment problem, and makes it harder for this procedure to provide useful guidance to the algorithms included in VP. The results in Table 3 actually illustrate this issue. Indeed, VP matches or even exceeds the performance of DR+LS for instances with both a high tardiness and a large range of due dates ($T=1.0$ and $R=0.8$, 1.0). In these instances, most jobs will have positive costs in the assignment problem, since most jobs, even those in the earlier positions, will indeed be tardy.

Therefore, the strategy used in the algorithms included in the VP procedure is suited for objective functions where the cost of each element will be almost always positive (and, as shown in [28], provided excellent results for one such problem). However, it does not seem to be a good fit for problems where several or most of the costs can be equal to 0, such as the weighted quadratic tardiness problem. In this latter case, it is then natural that it should be outperformed for methods that are specifically tailored to such an objective function.

Table 4 also shows that the ILS and SSGA procedures are quite close in solution quality, and somewhat superior to the VG metaheuristic. Indeed, the ivdr_ls values are higher for the ILS and SSGA heuristics, and these procedures also provide a better solution for a larger number of instances. Further details concerning the relative performance of the three metaheuristics are given in the next section.

4.4. Comparison of the metaheuristic procedures

This section presents additional information regarding the relative performance of the metaheuristic procedures, in regard to both solution quality and computational time. Table 5 gives the mean relative improvement versus the worst metaheuristic result (ivw_avg), as well as the number of times the metaheuristics provide a solution that is better than this worst result (n_btr_w).

Let ofv_{worst} denotes the worst objective function value given by the metaheuristics for a specific instance, that is, the highest objective function value among the 30 metaheuristic solutions (10 seeds for each procedure) generated for that instance. The mean relative improvement versus the worst provided by metaheuristic H_i is set at 0 when $ofv_{H_i} = ofv_{worst}$, thus avoiding divisions by 0. Otherwise, ivw_avg is calculated as $(ofv_{worst} - ofv_{H_i})/ofv_{worst} \times 100$. For each metaheuristic, the objective function value ofv_{H_i} is again the average over the 10 runs.

The results in Table 5 show that the metaheuristics are close and robust in what regards solution quality. Indeed, the mean relative improvement versus the worst result is never larger than 0.07%, so the worst objective function value is quite close to the average. The data in Table 5 also confirms that the ILS and SSGA procedures are somewhat similar regarding solution quality, with a very slight advantage to the ILS heuristic, and are superior to the VG metaheuristic. Indeed, and on the one hand, the ivw_avg values are smaller for the VG procedure. Also, and on the other hand, the ILS and SSGA heuristics provide a solution that is better than the worst for most of the instances, while the VG procedure fails to do so in most cases.

The average computational times (in seconds) for the metaheuristics are provided in Table 6. More specifically, this table gives the average runtime required to obtain the best solution found (rt_best), as well as the maximum allowed runtime (max_rt). Runtimes are not reported for the DR or DR+LS procedures, since these are extremely efficient. Indeed, and as an example, the DR+LS heuristic requires, on average, about less than 1 s for much larger instances with 2000 jobs.

The VP procedure was executed on a different computer, and its computational times are also not included in Table 6. However, the results showed that this procedure is substantially more time consuming than the metaheuristics. Specifically, and for instance, the average runtimes were larger than 33, 307 and 1313 s for instances with 100, 250 and 500 jobs, respectively. For these same instance sizes, the maximum time alloted to the metaheuristics, on the other hand, was 1.5, 6.75 and 25.5 s, respectively. Furthermore, the benchmarks available at http://www.cpubenchmark.net

Table 5
Comparison with worst metaheuristics result.

	ivw			n_btr_w		
n	ILS	VG	SSGA	ILS	VG	SSGA
10	0.0000	0.0000	0.0000	0.0	0.0	0.0
15	0.0000	0.0000	0.0000	0.0	0.0	0.0
20	0.0002	0.0001	0.0002	1.0	0.8	1.0
25	0.0013	0.0011	0.0013	8.0	6.5	8.0
30	0.0109	0.0076	0.0109	29.0	20.6	29.0
40	0.0204	0.0089	0.0204	73.0	44.1	73.0
50	0.0310	0.0102	0.0310	154.0	71.8	154.0
75	0.0697	0.0341	0.0698	356.3	140.9	357.0
100	0.0676	0.0123	0.0676	480.2	168.8	480.2
250	0.0471	0.0040	0.0469	812.9	192.9	811.4
500	0.0387	0.0024	0.0383	939.1	258.5	922.2

Table 6
Computational times (in seconds).

	rt_best			
n	ILS	VG	SSGA	max_rt
10	0.0000	0.0000	0.0000	0.5100
15	0.0000	0.0001	0.0000	0.5225
20	0.0000	0.0007	0.0001	0.5400
25	0.0000	0.0029	0.0002	0.5625
30	0.0001	0.0069	0.0004	0.5900
40	0.0006	0.0147	0.0012	0.6600
50	0.0021	0.0273	0.0023	0.7500
75	0.0126	0.0607	0.0084	1.0625
100	0.0416	0.0946	0.0225	1.5000
250	0.8114	0.3513	0.5217	6.7500
500	6.3276	1.3465	3.0996	25.5000

indicate that the computer on which the VP procedure was executed has a CPU that is over 4 times faster than the one used to run the metaheuristics. Thus, the runtimes given above for the VP procedure would have be multipled by about 4 to be comparable with the metaheuristic computation times. However, it should be remarked that the APdp(t) method accounts for most of the VP procedure's computation time. The remaining 3 methods included in the VP heuristic are quite fast and, as previously mentioned, the solution of the VP procedure was always given by one of these methods.

These results show that the time required for the VG procedure to reach its best solution scales better with the problem size. Indeed, and although it requires a larger time than the ILS and SSGA procedures for problems with 100 or fewer jobs, it is then much faster for the larger instances with 250 and 500 jobs. The ILS procedure, on the other hand, takes the longest to reach its best solution for these larger problem sizes. The rt_best values are much smaller than the maximum allowed runtime, so the chosen stop criterion does not seem to limit, at least in most cases, the ability of the algorithms to obtain a good solution.

The ILS procedure provides the best performance in what regards solution quality, albeit by a quite small margin when compared with the SSGA method. However, it also requires the longest to reach its best solution for the larger instance sizes. The VG algorithm is actually on the opposite end, since it is the fastest, but gives the worst solution quality results. The SSGA metaheuristic is a good all-around performer, providing a solution quality close to that of the ILS procedure, and an acceptable runtime.

When the available runtime is not likely to impair the ILS procedure, this is then the method of choice. However, if the time available to generate a solution is expected to be an issue, the VG algorithm usually reaches its best solution faster. In the face of uncertainty concerning the suitability of the available time, the SSGA metaheuristic is a good all-around performer.

5. Conclusion

In this paper, we considered the single machine scheduling problem with weighted quadratic tardiness costs. Three metaheuristics (ILS, VG and SSGA) were proposed, as well as a local search procedure which incorporates problem specific information. These heuristics fill a gap in the existing literature, which consists of branch-and-bound algorithms, which can provide an optimal solution for small instances, and efficient dispatching rules, which are often the only heuristic approach capable of providing solutions, in reasonable time, for large problems. Metaheuristics can often be quite useful for medium sized problems, for which they can usually give high quality solutions within reasonable computational times.

The proposed heuristics clearly outperform the DR, DR+LS and VP procedures. Indeed, the ILS and SSGA algorithms provide an optimal solution for all the smaller size problems, while the VG heuristic is nearly always optimal. The computational results provide evidence that the metaheuristics improve upon the DR+LS procedure both nearly as much as possible, and nearly as often as possible. Therefore, the proposed heuristics are the new procedures of choice for medium sized problems. For quite large instances, on which the metaheuristics are likely to require excessive computational time, the DR+LS heuristic can now replace the DR procedure, since the proposed local search method is both efficient and effective in improving the solution given by the QBack_v6 dispatching rule.

The ILS algorithm provides the best performance in terms of solution quality, but is also requires the longest time to reach its best solution for the larger of the tested instance sizes. The VG algorithm is exactly on the opposite end of the spectrum, while the SSGA is a good all-around performer. Thus, the choice of method is likely to depend on the information, or lack thereof, concerning the maximum available runtime and the computational effort required by each procedure.

Acknowledgment

The authors are most grateful to professors Mariona Vilà and Jordi Pereira for their help. Their willingness to run our instances on their algorithms made it possible to include a comparison with the VP procedure, thereby adding a valuable contribution to this paper. We greatly appreciate their help and effort.

References

[1] Wagner BJ, Davis DJ, Kher HV. The production of several items in a single facility with linearly changing demand rates. Decis Sci 2002;33(3):317–46.
[2] Sun XQ, Noble JS, Klein CM. Single-machine scheduling with sequence dependent setup to minimize total weighted squared tardiness. IIE Trans 1999;31(2):113–24.
[3] Hoitomt DJ, Luh PB, Max E, Pattipati KR. Scheduling jobs with simple precedence constraints on parallel machines. Control Syst Mag IEEE 1990; 10(2):34–40.
[4] Thomalla CS. Job shop scheduling with alternative process plans. Int J Prod Econ 2001;74(1-3):125–34.
[5] Taguchi G. Introduction to quality engineering: designing quality into products and processes. Tokio: Japan: Asian Productivity Organization; 1986.
[6] Schaller J, Valente JMS. Minimizing the weighted sum of squared tardiness on a single machine. Comput Oper Res 2012;39(5):919–28.
[7] Valente JMS, Schaller JE. Dispatching heuristics for the single machine weighted quadratic tardiness scheduling problem. Comput Oper Res 2012; 39(9):2223–31.
[8] Lawler ELA. "Pseudopolynomial" algorithm for sequencing jobs to minimize total tardiness. In: Hammer ELJBHK PL, Nemhauser GL, editors. Annals of discrete mathematics. Elsevier; 1977. p. 331–42.
[9] Lenstra JK, Rinnooy Kan AHG, Brucker P. Complexity of machine scheduling problems. In: Hammer ELJBHK PL, Nemhauser GL, editors. Annals of discrete mathematics. Elsevier; 1977. p. 343–62.
[10] Gupta SK, Sen T. Minimizing a quadratic function of job lateness on a single-machine. Eng Cost Prod Econ 1983;7(3):187–94.
[11] Sen T, Dileepan P, Lind MR. Minimizing a weighted quadratic function of job lateness in the single machine system. Int J Prod Econ 1995;42(3):237–43.
[12] Su LH, Chang PC. A heuristic to minimize a quadratic function of job lateness on a single machine. Int J Prod Econ 1998;55(2):169–75.
[13] Schaller J. Minimizing the sum of squares lateness on a single machine. Eur J Oper Res 2002;143(1):64–79.
[14] Soroush HM. A note on "minimizing a weighted quadratic function of job lateness in the single machine system". Int J Prod Econ 2009;121(1):296–7.
[15] Soroush HM. Single-machine scheduling with inserted idle time to minimise a weighted quadratic function of job lateness. Eur J Ind Eng 2010;4(2):131–66.
[16] Schaller J. Single machine scheduling with early and quadratic tardy penalties. Comput Ind Eng 2004;46(3):511–32.

[17] Valente JMS. Beam Search heuristics for the single machine scheduling problem with linear earliness and quadratic tardiness costs. Asia Pac J Oper Res 2009;26(3):319–39.
[18] Valente JMS. An exact approach for the single machine scheduling problem with linear early and quadratic tardy penalties. Asia Pac J Oper Res 2008; 25(2):169–86.
[19] Valente JMS. Heuristics for the single machine scheduling problem with early and quadratic tardy penalties. Eur J Ind Eng 2007;1(4):431–48.
[20] Valente JMS, Schaller JE. Improved heuristics for the single machine scheduling problem with linear early and quadratic tardy penalties. Eur J Ind Eng 2010;4(1):99–129.
[21] Behnamian J, Zandieh M. A discrete colonial competitive algorithm for hybrid flowshop scheduling to minimize earliness and quadratic tardiness penalties. Expert Syst Appl 2011;38(12):14490–8.
[22] Valente JMS, Alves RAFS. Heuristics for the single machine scheduling problem with quadratic earliness and tardiness penalties. Comput Oper Res 2008;35(11):3696–713.
[23] Valente JMS, Moreira MRA. Greedy randomised dispatching heuristics for the single machine scheduling problem with quadratic earliness and tardiness penalties. Int J Adv Manuf Technol 2009;44(9-10):995–1009.
[24] Valente JMS. Beam search heuristics for quadratic earliness and tardiness scheduling. J Oper Res Soc 2010;61(4):620–31.
[25] Valente JMS, Moreira MRA, Singh A, Alves RAFS. Genetic algorithms for single machine scheduling with quadratic earliness and tardiness costs. Int J Adv Manuf Technol 2011;54(1):251–65.
[26] Singh A, Valente JMS, Moreira MRA. Hybrid heuristics for the single machine scheduling problem with quadratic earliness and tardiness costs. Int J Mach Learn Cybern 2012;3(4):327–33.
[27] Kianfar K, Moslehi G. A branch-and-bound algorithm for single machine scheduling with quadratic earliness and tardiness penalties. Comput Oper Res 2012;39(12):2978–90.
[28] Vila M, Pereira J. Exact and heuristic procedures for single machine scheduling with quadratic earliness and tardiness penalties. Comput Oper Res 2013; 40(7):1819–28.
[29] Abdul-Razaq TS, Potts CN, van Wassenhove LN. A survey of algorithms for the single-machine total weighted tardiness scheduling problem. Discret Appl Math 1990;26(2–3):235–53.
[30] Potts CN, van Wassenhove LN. Single-machine tardiness sequencing heuristics. IIE Trans 1991;23(4):346–54.
[31] Sen T, Sulek JM, Dileepan P. Static scheduling research to minimize weighted and unweighted tardiness: a state-of-the-art survey. Int J Prod Econ 2003; 83(1):1–12.
[32] Baxter J. Local optima avoidance in depot location. J Oper Res Soc 1981; 32(9):815–9.
[33] Baum EB. Iterated descent: a better algorithm for local search in combinatorial optimization problems (Technical report). Pasadena, CA: Caltech; 1986.
[34] BE, B. Towards practical 'neural' computation for combinatorial optimization problems. In: Proceedings of AIP Conference 151 on Neural Networks for Computing. Snowbird, Utah, USA: American Institute of Physics Inc.; 1987. p. 53–8.
[35] Congram RK, Potts CN, van de Velde SL. An iterated dynasearch algorithm for the single-machine total weighted tardiness scheduling problem. INFORMS J Comput 2002;14(1):52–67.
[36] Johnson DS. Local Optimization and the Traveling Salesman Problem. Lect Notes Comput Sc 1990;443:446–61.
[37] Lourenco HR. Job-shop scheduling-computational study of local search and large-step optimization methods. Eur J Oper Res 1995;83(2):347–64.
[38] Lourenço HR, Martin OC, Stützle T. Iterated Local Search. In: Glover F, Kochenberger GA, editors. Handbook of metaheuristics. Dordrecht: Kluwer Academic Publishers; 2003. p. 321–53.
[39] Martin O, Otto SW, Felten EW. Large-step Markov chains for the traveling salesman problem. Complex Syst 1991;5(3):299–326.
[40] Martin OC, Otto SW. Combining simulated annealing with local search heuristics. Ann Oper Res 1996;63:57–75.
[41] Framinan JM, Leisten R. Total tardiness minimization in permutation flow shops: a simple approach based on a variable greedy algorithm. Int J Prod Res 2008;46(22):6479–98.
[42] Ruiz R, Stützle T. A simple and effective iterated greedy algorithm for the permutation flowshop scheduling problem. Eur J Oper Res 2007;177(3):2033–49.
[43] Ruiz R, Stützle T. An Iterated Greedy heuristic for the sequence dependent setup times flowshop problem with makespan and weighted tardiness objectives. Eur J Oper Res 2008;187(3):1143–59.
[44] Mladenovic N, Hansen P. Variable neighborhood search. Comput Oper Res 1997;24(11):1097–100.
[45] Nawaz M, Enscore Jr EE, Ham I. A heuristic algorithm for the m-machine, n-job flow-shop sequencing problem. Omega 1983;11(1):91–5.
[46] Holland JH. Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence. Ann Arbor, Michigan: University of Michigan Press; 1975.
[47] Davis L. Handbook of genetic algorithms. New York: Van Nostrand Reinhold; 1991.
[48] Goldberg DE, Deb K. A comparative analysis of selection schemes used in genetic algorithms. Found Genet Algorithms 1990:69–93.
[49] Reeves C. Genetic algorithms. In: Glover F, Kochenberger GA, editors. Handbook of metaheuristics. Dordrecht: Kluwer Academic Publishers; 2003. p. 55–82.
[50] Beasley JE, Chu PC. A genetic algorithm for the set covering problem. Eur J Oper Res 1996;94(2):392–404.
[51] Ow PS, Morton TE. Filtered beam search in scheduling. Int J Prod Res 1988; 26(1):35–62.
[52] Ow PS, Morton TE. The single-machine early tardy problem. Manag Sci 1989;35(2):177–91.