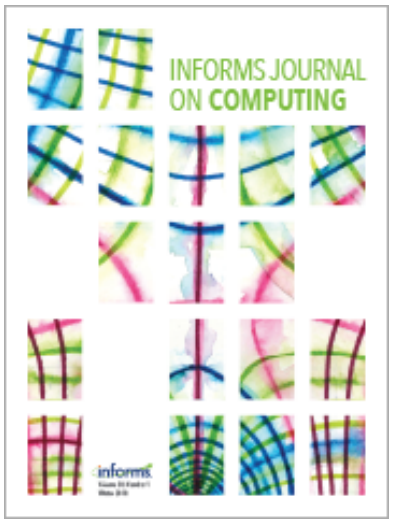

INFORMS Journal on Computing

Publication details, including instructions for authors and subscription information:
http://pubsonline.informs.org

Local Search Heuristics for the Single Machine Total Weighted Tardiness Scheduling Problem

H. A. J. Crauwels, C. N. Potts, L. N. Van Wassenhove,

To cite this article:
H. A. J. Crauwels, C. N. Potts, L. N. Van Wassenhove, (1998) Local Search Heuristics for the Single Machine Total Weighted Tardiness Scheduling Problem. INFORMS Journal on Computing 10(3):341-350. https://doi.org/10.1287/ijoc.10.3.341

INFORMS Journal on Computing
Vol. 10, No. 3, Summer 1998

0899-1499/98/1003-0341 $05.00

Local Search Heuristics for the Single Machine Total Weighted Tardiness Scheduling Problem

H. A. J. CRAUWELS / *De Nayer Instituut, J. De Nayerlaan 5, 2860 Sint-Katelijne-Waver, Belgium, Email: hcr@denayer.be*

C. N. POTTS / *Faculty of Mathematical Studies, University of Southampton, Southampton SO17 1BJ, U.K., Email: cnp@maths.soton.ac.uk*

L. N. VAN WASSENHOVE / *INSEAD, Boulevard de Constance, 77305 Fontainebleau, France, Email: Wassenhove@smtp1.insead.fr*

(Received: April 1994; revised August 1997; accepted: October 1997)

This paper presents several local search heuristics for the problem of scheduling a single machine to minimize total weighted tardiness. We introduce a new binary encoding scheme to represent solutions, together with a heuristic to decode the binary representations into actual sequences. This binary encoding scheme is compared to the usual "natural" permutation representation for descent, simulated annealing, threshold accepting, tabu search and genetic algorithms on a large set of test problems. Computational results indicate that all of the heuristics which employ our binary encoding are very robust in that they consistently produce good quality solutions, especially when multistart implementations are used instead of a single long run. The binary encoding is also used in a new genetic algorithm which performs very well and requires comparatively little computation time. A comparison of neighborhood search methods which use the permutation and binary representations shows that the permutation-based methods have a higher likelihood of generating an optimal solution, but are less robust in that some poor solutions are obtained. Of the neighborhood search methods, tabu search clearly dominates the others. Multistart descent performs remarkably well relative to simulated annealing and threshold accepting.

The single machine total weighted tardiness problem may be stated as follows. Each of n jobs (numbered $1, \ldots, n$) is to be processed without interruption on a single machine that can handle only one job at a time. Job i ($i = 1, \ldots, n$) becomes available for processing at time zero, requires an integer processing time p_i, has a positive weight w_i and a due date d_i. For a given processing order of the jobs, the (earliest) completion time C_i and the tardiness $T_i = \max\{C_i - d_i, 0\}$ of job i ($i = 1, \ldots, n$) can be computed. The objective is to find a processing order of the jobs that minimizes $\Sigma_{i=1}^{n} w_i T_i$.

The total weighted tardiness problem is (unary) NP-hard (Lawler[8] and Lenstra et al.[9]). Many dynamic programming and branch and bound algorithms have been suggested in literature: Abdul-Razaq et al.[1] present a computational comparison of several of these algorithms. Each of the available enumerative algorithms places substantial demands on computation time and/or core storage, especially when the number of jobs is more than about 50. Consequently, computer resources are not always adequate to obtain exact solutions. Therefore, the quest for good and robust heuristics is of great practical relevance.

As indicated by Potts and Van Wassenhove,[14] simple dispatching rules (EDD, SWPT, COVERT or AU) do not consistently provide good quality solutions. In recent years, much attention has been devoted to a number of *local search* heuristics for 'solving' combinatorial optimization problems. Neighborhood search forms one category of local search heuristics. Starting from an arbitrary solution, these methods construct a sequence of solutions by repeatedly moving from the current solution to a solution in an appropriately defined neighborhood. The most common method of this type is descent in which the transition from neighbor to neighbor continues until a better solution can no longer be found. The resulting solution is a local optimum. The main weakness of descent is that the local optimum may deviate significantly from the true (global) optimum. Simulated annealing, threshold accepting, and tabu search are local search strategies designed to overcome this weakness. They allow neighborhood moves which result in a deterioration of the objective function value and can therefore potentially escape from a local optimum. Genetic algorithms form another category of local search methods. By working with a population of solutions and by allowing these solutions to interact, 'children' are produced that hopefully retain the desirable characteristics of their parents. An introductory overview of these methods and some applications are given by Goldberg[7] and Pirlot.[12]

Potts and Van Wassenhove[14] propose descent and simulated annealing methods for the total weighted tardiness problem. In this paper, we extend their work by giving a thorough comparison of single- and multi-start versions of descent, simulated annealing, threshold accepting, tabu search, and a genetic algorithm. In addition to comparing different types of methods, we present results for two different representations of solutions; the natural permutation representation and a new binary representation.

The remaining sections of this paper are organized as follows. In Section 1, we present the two different representations for a solution, and define the structure of the corre-

Subject classifications: Production scheduling, deterministic single machine.
Other key words: Single machine, sequencing, heuristic, local search.

sponding neighborhoods. Sections 2 and 3 describe neighborhood search methods and a genetic algorithm, respectively. Computational results for the different methods on the test data sets of Potts and Van Wassenhove[14] are reported in Section 4. Finally, Section 5 summarizes the main conclusions.

1. Representations and Neighborhoods

For the development of a local search heuristic, one has to choose how to represent a solution and how this representation is to be manipulated during the search process. In this section, we present the 'natural' permutation representation, and suggest a new binary representation that incorporates some problem-specific knowledge.

1.1 Permutation Representation

For the single machine total weighted tardiness problem, the natural representation of a solution is a permutation of the integers $1, \ldots, n$, which defines the processing order of jobs. This representation is used by Potts and Van Wassenhove[14] in their descent and simulated annealing algorithms.

For this representation, the following neighborhood is defined. Two positions u and v ($1 \leq u < v \leq n$) in the current sequence of jobs are selected, and a new sequence is generated by interchanging the jobs in positions u and v. The (u, v) pairs can be selected at random, or with a systematic and repeated scanning of all job positions, e.g., $(u, v) = (1, 2), (1, 3), \ldots, (n-1, n)$. This neighborhood, which has size $n(n-1)/2$, is called *general pairwise interchange* (GPI). The *adjacent pairwise interchange* (API) neighborhood is a subset of GPI containing the $n-1$ neighbors for which $v = u + 1$.

The permutation that is used as a starting solution in a neighborhood search algorithm either can be constructed with a dispatching rule (e.g., the Apparent Urgency rule of Morton et al.,[11] which is described in [10]), or can be generated at random.

1.2 Binary Representation

We assume that the jobs are renumbered in shortest weighted processing time (SWPT) order so that $p_1/w_1 \leq \ldots \leq p_n/w_n$, where $d_i \leq d_{i+1}$ whenever $p_i/w_i = p_{i+1}/w_{i+1}$ ($i = 1, \ldots n-1$). A representation consists of a string of n binary digits, where the elements correspond to jobs $1, \ldots, n$, respectively. If the element corresponding to any job i is set to 1, then i is considered to be early; otherwise, this element is 0 and job i is regarded as being late.

For this representation, we define a *bit inversion* neighborhood. A neighbor is generated by inverting the element in some position i of the string ($1 \leq i \leq n$), i.e., an element 1 is changed to 0 and vice versa. This neighborhood has size n.

The set of early jobs, as defined by such a bit string, cannot necessarily be scheduled so that each job is completed by its due date. In a heuristic procedure that is presented below, we attempt to construct a sequence with the actual early and late jobs matching the bit string as closely as possible.

We now describe methods to generate bit strings that can be used in the initialization of local search methods. A bit string can be constructed by a random assignment: the probability of setting a element to 1 is equal to 0.5. However, it is preferable to use the following more sophisticated approach. The number of elements set to 1 (corresponding to early jobs) is related to the average tardiness factor TF which is defined by $\mathrm{TF} = \max\{1 - d_{\mathrm{avg}}/P, 0\}$, where $d_{\mathrm{avg}} = \sum_{i=1}^{n} d_i/n$ is the average due date and $P = \sum_{i=1}^{n} p_i$ is the total processing time. Clearly, a relatively small value of TF arises when due dates are larger, and therefore there are more early jobs in an optimal solution. The value of p_i/w_i also influences the setting of the element corresponding to job i ($i = 1, \ldots, n$): the likelihood that job i is early in an optimal solution increases if p_i/w_i is small relative to the average value $(\sum_{j=1}^{n} p_j/w_j)/n$. Thus, the probability P_i, where $0.01 \leq P_i \leq 0.99$, of setting the element corresponding to job i to 1 should be larger when both TF and p_i/w_i are relatively small. Based on these observations, we define

$$P_i = 1 - \max\left\{0.01, \min\left\{0.99, n\mathrm{TF}(w_i/p_i) \Big/ \sum_{j=1}^{n} p_j/w_j\right\}\right\}. \tag{1}$$

A method is needed to decode a bit string into a solution defined by a sequence of jobs. Having found such a sequence, the corresponding objective function value can be computed. Based on the values of elements in the string under consideration, we define sets of early and late jobs. Firstly, the early set is checked for feasibility and, where necessary, jobs are transferred from the early to the late set. Having obtained a feasible early set, we use a heuristic method to construct a schedule. Our heuristic first schedules the early jobs as late as possible in EDD order, and subsequently attempts to insert jobs of the late set between the early jobs. During this insertion phase, preference is given to jobs which were transferred from the early to the late set. Finally, any remaining jobs are appended to the resulting partial sequence in SWPT order. We now give precise details of our heuristic.

Decoding Heuristic

Step 1. Construct a list E of early jobs in EDD order and let L_0 be a list of late jobs. Let $E = (\pi(1), \ldots, \pi(n_e))$, where n_e is the number of jobs in E. Set $i = 0$, $j = 1$ and $t = 0$. Set $L_1 = \varnothing$ (a list of jobs transferred from the early set).

Step 2. If $t + p_{\pi(j)} \leq d_{\pi(j)}$, set $i = i + 1$, $\sigma(i) = \pi(j)$ and $t = t + p_{\pi(j)}$. If $t + p_{\pi(j)} > d_{\pi(j)}$, remove job $\pi(j)$ from E and append it to L_1. Set $j = j + 1$. If $j < n_e$, repeat Step 2. If $j = n_e$, set $n_e = i$.

Step 3. Reorder jobs of the lists L_0 and L_1 of late jobs in SWPT order. If $L_1 = \varnothing$, set $L = L_0$ and $L_0 = \varnothing$; otherwise, set $L = L_1$. Compute latest start times for the jobs of E using $s_{\sigma(n_e)} = d_{\sigma(n_e)} - p_{\sigma(n_e)}$, and $s_{\sigma(i)} = \min\{s_{\sigma(i+1)}, d_{\sigma(i)}\} - p_{\sigma(i)}$ for $i = n_e - 1, n_e - 2, \ldots, 1$. Compute $R = 2n \max_{i=1,\ldots,n}\{p_i/w_i\}/\sum_{i=1}^{n} p_i/w_i$. Set $t = 0$ (t denotes the completion time of the current partial sequence), and set e to be the first job in E.

Step 4. If $s_e = t$, go to Step 8. Otherwise, set i to be the first job in L and, if possible, find the first job j for which $t + p_j > d_j$ (the job is late) and $p_j/w_j \leq Rp_i/w_i$ (the ratios p_i/w_i and p_j/w_j are relatively close). If j cannot be found, go to Step 8. If $t + p_j \leq s_e$, go to Step 5 (we search to find whether

there is a better job than j in L to be scheduled before job e); otherwise, if $t + p_j > s_e$, go to Step 6 (we search to find whether there is a job of L that can be scheduled before job e without making it late).

Step 5. Search for the first job k that follows job j in L for which $t + p_k > d_k$ (the job is late), $t + p_k \leq s_e$ (the job can be scheduled before job e) and one of the two following conditions is satisfied:

(a) $t + p_j + p_k \leq s_e$ (both j and k can be scheduled before job e) and $V_k > V_j$, where $V_j = w_j(t + p_j - d_j)$ and $V_k = w_k(t + p_k - d_k)$;

(b) $t + p_j + p_k > s_e$ (jobs j and k cannot both be scheduled before e) and $V_{kej} \leq V_{jek}$, where $V_{jek} = w_j(t + p_j - d_j) + w_k(t + p_j + p_e + p_k - d_k)$ and $V_{kej} = w_k(t + p_k - d_k) + w_j(t + p_k + p_e + p_j - d_j)$.

If k is found, set $h = k$ (job k is scheduled next) and go to Step 7; otherwise, set $h = j$ (job j is scheduled next) and go to Step 7.

Step 6. Search for the first job l that follows job j in L for which $t + p_l > d_l$ (the job is late), $t + p_l \leq s_e$ (the job can be scheduled before job e) and $V_{lej} < V_{ejl}$, where $V_{lej} = w_l(t + p_l - d_l) + w_j(t + p_l + p_e + p_j - d_j)$ and $V_{ejl} = w_j(t + p_e + p_j - d_j) + w_l(t + p_e + p_j + p_l - d_l)$. If l is found, set $h = l$ (job l is scheduled next) and go to Step 7; otherwise (job e is scheduled next), go to Step 8.

Step 7. Add job h to the partial sequence, delete h from L and set $t = t + p_h$. If $L = \emptyset$, then set $L = L_0$ and $L_0 = \emptyset$. Go to Step 4.

Step 8. Add the early job e to the partial sequence, delete e from E, and set $t = t + p_e$. If E is not empty, set e to be the first job in E and go to Step 4. Otherwise, add to the partial sequence in SWPT order any remaining jobs in L and L_0.

Step 9. The resulting sequence is taken as the initial solution for a descent algorithm which uses the API neighborhood. During this search, a neighbor is accepted if it decreases the objective value, or if it leaves the value unaltered and the two interchanged jobs become sequenced in EDD order.

Example. To illustrate the mechanics of the decoding heuristic, consider the following 5-job problem. The processing times are 10, 7, 5, 9, 6; the job weights 10, 6, 3, 5, 3; and the due dates are 22, 30, 17, 8, 0. For the bit string *01100*, jobs 2 and 3 are early, and jobs 1, 4 and 5 are late.

In Step 1, $E = (3, 2)$, $L_1 = \emptyset$ and $L_0 = (1, 4, 5)$. Since E is a feasible set of early jobs, sets E, L_1 and L_0 are unchanged in Step 2. In Step 3, after observing that no reordering of L_0 is necessary, we compute latest possible start times $s_2 = 23$ and $s_3 = 12$, compute $R = 2.62$, and set $t = 0$, $e = 3$ and $L = L_0$. In Step 4, $i = 1$, but job 1 cannot be scheduled before job 3 to start at time zero because it would not be late. Job 4, however, is late if scheduled to start at time zero, so $j = 4$. In Step 5, for the candidate job $k = 5$, we compute the values $V_{jek} = 65$ and $V_{kej} = 78$ based on the partial sequences (4, 3, 5) and (5, 3, 4) to test condition (b). Thus, k cannot be found, so Step 7 forms the partial sequence comprising job 4 and sets $t = 9$ and $L = (1, 5)$. After returning to Step 4 and setting $i = 1$ and $j = 5$, the heuristic proceeds to Step 6 to search for a job of L to be scheduled next so that it completes not later than time $s_3 = 12$. Since no such job can be found, Step 8 schedules job 3 next to give the partial sequence (4, 3), sets $t = 14$, $E = (2)$ and $e = 2$.

On returning to Step 4, $i = 1$, and we choose $j = 1$. Since $t + p_j = 24 > 23 = s_2$, job 1 cannot be scheduled before job 2 without making it late. In Step 6, the candidate job $l = 5$ fails because the weighted tardiness of the partial sequence (which starts at time 14) (5, 2, 1) is $V_{lej} = 210$ which exceeds $V_{ejl} = 201$ that corresponds to (2, 1, 5). Thus, l cannot be found, and we proceed to Step 8 where job 2 is scheduled to create the partial sequence (4, 3, 2) and E becomes empty. Step 8 also appends the jobs of L to give the complete sequence (4, 3, 2, 1, 5).

Lastly, Step 9 reduces the total weighted tardiness from 206 to 142 by interchanging jobs 2 and 1 to give the sequence (4, 3, 1, 2, 5).

In local search, there is normally an implicit assumption that when the relevant method makes suitable choices for producing new solutions, an optimal solution is eventually generated. For our binary representation, however, we are not able to guarantee that an optimal solution can always be obtained from some binary representation, and thus this assumption does not hold. Nevertheless, this is not a serious disadvantage since initial experiments with the binary representation show that high quality solutions are generated for most instances.

2. Neighborhood Search Methods

A neighborhood search method requires a representation of solutions to be chosen, and an initial solution to be constructed by some heuristic rule or created randomly. A neighbor is generated by some suitable mechanism, and an acceptance rule is used to decide on whether it should replace the current solution. This process is repeated until some termination criterion is satisfied. The acceptance rule in a neighborhood search method usually requires the comparison of objective function values for the current solution and its neighbor. If the former is larger (for a minimization problem), then we refer to the neighbor as an *improving move*; if the latter is larger, it is a *deteriorating move*; if both are the same, then it is a *neutral move*.

In a *descent* method, a series of moves from one solution to another solution in its neighborhood is performed, where each move results in an improvement of the objective function value. When no further improvement can be found, a classical descent procedure terminates and the solution is a *local optimum*. However, in the total weighted tardiness problem, neighbors frequently have the same objective function value as the current solution. Therefore, it can be opportune to accept neutral moves. Our stopping criterion in descent with neutral moves is defined by fixing the number of levels l, where a level refers to the systematic search of $|N|$ neighbors, where N is the neighborhood that is used and $|N|$ is its size.

The local optimum generated by a classical descent method is not necessarily a global optimum. A widely used remedy for this drawback is to perform multiple runs of the procedure starting from different initial solutions, and to

select the best sequence as final solution. Such an approach is called *multistart descent*.

Sections 2.1, 2.2, and 2.3 describe other neighborhood search methods that allow the search to escape from a local optimum. These methods each have parameters, the values of which are fixed using the results of preliminary tests. Our preliminary tests are described later in Section 4.2.

2.1 Simulated Annealing

In a *simulated annealing* procedure, improving and neutral moves are accepted, while deteriorating moves are accepted according to a given probabilistic acceptance function. The most common form of the acceptance function is $p(\delta) = \exp(-\delta/t)$, where $p(\delta)$ is the probability of accepting a move which results in an increase of δ (where $\delta > 0$) in the objective function value. The parameter t is known as the *temperature*, and its value changes at suitably chosen intervals. A review of simulated annealing is provided by Eglese.[4]

We use the implementation of Potts and Van Wassenhove,[14] where the temperature t_k at each level k, for $1 \leqslant k \leqslant l$, is derived from an acceptance probability K_k. In this scheme, K_k is the probability of accepting a one percent increase in the objective function value. Thus, $K_k = \exp(-0.01Z/t_k)$, where Z is the total weighted tardiness of the best solution found thus far, from which we obtain $t_k = -0.01Z/\ln K_k$. The acceptance probability decreases geometrically:

$$K_k = aK_{k-1} \quad \text{with } a = (K_l/K_1)^{1/(l-1)}$$

where K_1 and K_l are the initial and final acceptance probabilities, respectively. We use the parameter values $K_1 = 0.99$ and $K_l = 0.001$.

This acceptance probability K_k is kept constant for one level during which the GPI or the bit inversion neighborhood is systematically searched. The number of levels l is a parameter of the algorithm. For the permutation representation, an additional intensification device is built into our procedure. A descent algorithm, based on the API neighborhood, is applied to find a local optimum whenever an increase of the objective function is followed immediately by a decrease.

2.2 Threshold Accepting

Threshold accepting is a method similar to simulated annealing that uses a deterministic acceptance rule for solutions that cause a deterioration in the objective function value. Here, a move is accepted provided that it does not increase the objective function by more than V, where V is a *threshold value*. The threshold value plays a similar role to the temperature in simulated annealing. Threshold accepting is first introduced by Dueck and Scheuer,[3] who claim that it yields better results than simulated annealing.

To set the threshold values V_k, for $1 \leqslant k \leqslant l$, we use a geometric decreasing scheme:

$$V_k = \alpha V_{k-1} \quad \text{with } \alpha = (V_l/V_1)^{1/(l-1)}$$

where V_1 and V_l are the initial and final threshold values, respectively. These threshold values are computed from the objective function value of the initial solution Z_0 by setting $V_1 = v_1 Z_0$ and $V_l = v_l Z_0$, where v_1 and v_l are parameters that determine the maximum increase in objective function value, relative to Z_0, that can be accepted at the initial and final levels. We use the parameter values $v_1 = 0.02$ and $v_l = 0.0001$.

Other features of our threshold accepting algorithm, including neighborhood structures, method of search and the intensifying device, are analogous to those for simulated annealing.

2.3 Tabu Search

Tabu search is a third neighborhood search strategy designed to allow escape from local optima. In this method, one (or sometimes several) attributes of the most recently executed moves are contained in a *tabu list*, and moves which would reverse this attribute are *tabu*. This method executes a move from the current solution to the best solution in its neighborhood (or some subset), ignoring any tabu moves unless an *aspiration* criterion is used.

The main features of tabu search are described by Glover.[5] In addition to choosing a representation of solutions and a neighborhood, a tabu search method requires the selection of one or more attributes to store on the tabu list, a specification of which moves are forbidden, and the definition of aspiration criteria. There are, in addition, optional features such as devices to diversify the search. The aim of diversification is to drive the search into unexplored regions of the solution space.

2.3.1 Choice of Suitable Neighborhood

For the permutation representation, the GPI neighborhood requires a large computational investment to find the best neighbor. Our initial experiments show that it is preferable to perform more iterations with a restricted neighborhood. One method of restricting the neighborhood is to consider only API moves. However, under this approach, the moves are too restrictive to allow enough different regions of the solution space to be searched. As an alternative, we create a small subset containing n general interchange neighbors which are selected by generating (u, v) pairs at random. Another benefit of this smaller neighborhood is that there is a greater likelihood of escaping from a local optimum, thereby avoiding a long sequence of moves on a flat region. On the other hand, it is possible that some good improving moves are not attempted because the corresponding neighbors do not appear in the subset. Therefore, we incorporate an intensifying step which is described in Section 2.3.2.

When the binary encoding is used, the best non-tabu move is chosen from the bit inversion neighborhood, which is described in Section 1.2.

For both representations, if a non-tabu neighbor is found which improves the current objective function value, then the corresponding move is executed without further search of the neighborhood of the current solution.

2.3.2 The Tabu List and Aspiration

The structure of the tabu list depends on the representation. For the permutation encoding, after each move in which jobs in positions u and v are interchanged, the job in position v of the new sequence is stored in the tabu list together with the corresponding objective function value of the new sequence. The length of this list is equal to 7 in our implementation. Any neighbor that is created by interchanging a job which appears on the tabu list is prohibited unless an aspiration criterion is satisfied. This criterion is satisfied if the objective function value of the sequence created by the interchange is strictly smaller than the objective function value in the tabu list which corresponds to the job that is interchanged during the creation of this new solution.

For the binary representation, the tabu status is characterized by the element just inverted: this element cannot be reinverted to its original value during a number of iterations. The tabu list length depends on the problem size and is equal to $n/3$. Here, a simpler aspiration criterion is used: if the objective function value of a tabu neighbor is better than that for all solutions generated thus far, then its tabu status is overridden.

2.3.3 Intensification and Diversification

According to Glover,[6] a good heuristic search strategy is to alternate between phases involving intensification and diversification. The tabu list is a short-term memory device which helps to diversify the search. However, to strengthen the inherent intensifying and diversifying components already present in tabu search, we incorporate two additional devices. Firstly, as in the simulated annealing heuristic of Potts and Van Wassenhove,[14] a descent algorithm is applied to find a local optimum whenever an increase of the objective function is followed immediately by a decrease. The descent algorithm uses the API neighborhood. Because the tabu list is ignored during the descent procedure, the result is an intensified search to a local optimum. However, to avoid disruption to the tabu search algorithm, the search continues from the sequence that is input into the descent step, rather than from the local optimum sequence that is output.

There is no extra intensifying step under the binary representation, because the last step of our decoding heuristic performs descent with the API neighborhood.

To motivate the use of our second device, suppose that during a number of successive iterations the best move does not change the objective function value because the two jobs which are interchanged are both early. A stronger diversifying move is then necessary to drive the search into a new region of the solution space. For the permutation representation, this is achieved by imposing some additional restrictions on the selection of (u, v) pairs. We choose the best move from three candidate (u, v) pairs, where u is randomly chosen from the first $n/4$ positions in the sequence and v is randomly selected from the last $n/4$ positions. We use this diversifying device after performing $n/3$ consecutive iterations with neutral moves.

We also adopt a diversifying device for the binary representation. First, a new bit string is derived from the decoded sequence from the previous iteration. Elements corresponding to early jobs in this decoded sequence are set to 1; the others are set to 0. Then, only neighbors that reset a 1 to a 0 are considered. Consequently, there is a greater likelihood that the next move is deteriorating, because one additional job will be late. We apply this diversification step after 10 consecutive neutral moves.

3. A Genetic Algorithm

Genetic algorithms, motivated by natural selection and evolution, form another category of local search methods. Key components of a genetic algorithm include a "chromosomal" representation of solutions, a mechanism to generate an initial population, a measure of solution fitness (based on the objective function), and genetic operators that combine and alter current (parent) solutions to form new (child) solutions. An introductory account of the theory of genetic algorithms, as well as the main developments and applications, is given by Goldberg.[7]

3.1 Solution Encoding

The success of a genetic algorithm relies on an appropriate representation of solutions. The permutation representation of Section 1.1 and the binary representation of Section 1.2 are obvious candidates. However, disadvantages of a 'natural' permutation representation include the difficulty in applying the genetic operators and the possibility that child solutions do not inherit important parental features.

There have been attempts to define crossover operators for a permutation representation. For example, Goldberg[7] defines the *partially mapped crossover* (PMX) for the traveling salesman problem. This operator tends to maintain cities in their positions in the permutation, which is also a desirable characteristic for problems involving the sequencing of jobs. Della Croce et al.[2] use the *linear order crossover* (LOX) for the job shop problem. The LOX preserves, as far as possible, the relative positions between the elements and the absolute positions in the sequence. Our initial experiments with a permutation representation and these PMX and LOX operators yield poor quality solutions. Therefore, we only present a genetic algorithm which uses the binary representation.

3.2 Initial Population

In the first chromosome of our initial population, all elements are set to 1. This chromosome is required to find a quick solution to problems where all the jobs can be sequenced on time (in order of non-decreasing due dates, or EDD order). To create an initial population of size M, we randomly generate a further $M - 1$ chromosomes, where the probability P_i of setting the element corresponding to job i $(i = 1, \ldots, n)$ to 1 is defined by (1).

We now attempt to improve the quality of our initial population. First, every chromosome is decoded using the heuristic in Section 1.2, and its total weighted tardiness is evaluated. The average total weighted tardiness T_{avg} for these M chromosomes is computed. Then, $M/2$ new chro-

mosomes are created. In the first of these chromosomes, all elements are set to 0 (all jobs are regarded as being late), while the other $M/2 - 1$ additional chromosomes are randomly generated using a probability $P_i = 1 - \text{TF}$ of setting each element to 1. The following procedure is used to test whether an original chromosome with the largest total weighted tardiness $T_{\max}$ should be replaced by a new chromosome. The condition for replacement is that the total weighted tardiness for the new chromosome should be less than T_{avg}, and also less than $T_{\max}$. This test for replacement is applied for each of the $M/2$ new chromosomes.

The purpose of this more sophisticated initialization method is to reduce the number of populations that are generated by the genetic algorithm, thereby speeding up the procedure. By starting from a completely randomly generated population (each $P_i = 0.5$), it typically takes a number of generations to obtain a population that has similar characteristics to the one that is created with our initialization procedure.

The values of parameters are fixed using the results of preliminary tests, which are described later in Section 4.2. We use a population size $M = n$.

3.3 Fitness and Genetic Operators

When a new population is generated, each chromosome undergoes *evaluation*. First, the total weighted tardiness is computed using the decoding heuristic described in Section 1.2. Let $g_1, \ldots, g_M$ denote these values. Next, $g_1, \ldots, g_M$ are mapped to fitness values $f_1, \ldots, f_M$: we use $f_k = g_{\max} - g_k$ for $k = 1, \ldots, M$, where $g_{\max} = \max_{k=1, \ldots, M}\{g_k\}$. Sometimes the minimum and maximum fitness values are identical (all chromosomes have the same total weighted tardiness). In that case, the algorithm is terminated. Termination also occurs if a chromosome is found which yields a sequence with total weighted tardiness equal to zero.

The main genetic operators are reproduction, crossover, and mutation. The *reproduction* operator is an artificial version of natural selection. Chromosomes with a larger fitness value have a higher probability of survival by being placed in a *mating pool*, since the probability of selection is proportional to the fitness value. Instead of using raw fitness values $f_1, \ldots, f_M$ in the creation of the mating pool, they are replaced by scaled fitness values $F_1, \ldots, F_M$. This scaling prevents premature convergence and, in the latter stages of the algorithm, avoids random walks among mediocre solutions. We use linear scaling, defined by

$$F_k = af_k + b \qquad k = 1, \ldots, M, \tag{2}$$

where a and b are non-negative constants. We compute a and b from two scaling relationships. The first dictates that average fitness remains unaltered by scaling, so that

$$\sum_{k=1}^{M} F_k = \sum_{k=1}^{M} f_k. \tag{3}$$

The second relationship specifies that the expected number of copies of the best population member to appear in the mating pool is some given quantity C, which yields

$$\max_{k=1, \ldots, M} \{F_k\} = C \sum_{k=1}^{M} F_k/M. \tag{4}$$

Because of (4), there is a possibility that this scaling could produce negative fitness values. If the constants a and b that are derived from (2), (3) and (4) yield a negative minimum fitness value, then (4) is replaced by an alternative relationship which specifies that the minimum scaled fitness is zero. Thus,

$$\min_{k=1, \ldots, M} \{F_k\} = 0. \tag{5}$$

Therefore, a and b are defined by (2), (3), and (4) unless, upon substitution in (2), one or more negative scaled fitness values are obtained, in which case (2), (3), and (5) are used to determine a and b. For our computational tests, we set $C = 2$.

Based on the scaled fitness values, chromosomes are selected for the mating pool using *deterministic sampling*. Since the selection probability of chromosome k ($k = 1, \ldots, M$) is $F_k/\Sigma_{l=1}^{M} F_l$ and the mating pool is of size M, the expected number of copies in the mating pool is $e_k = MF_k/\Sigma_{l=1}^{M} F_l$. First, $\lfloor e_k \rfloor$ copies of each chromosome k are placed in the mating pool, and then the population is sorted in nonincreasing order of the fractional parts $e_k - \lfloor e_k \rfloor$. The remainder of the chromosomes needed to fill the mating pool are drawn from the start of this sorted list.

For *crossover*, the chromosomes in the mating pool are mated at random in pairs. Instead of applying the usual one- or two-point crossover to the two chromosomes, we use a scheme which interchanges several small sections of the strings, where the sections are randomly selected. For example, suppose that this crossover operator interchanges sections of length two, and is applied to the two chromosomes 1011 0101 and 0010 1001. If the two sections starting at the third and sixth element are interchanged, we obtain the resulting new chromosomes 1010 0001 and 0011 1101. The number of sections to be interchanged and the lengths of the sections are parameters. The two children replace their parent chromosomes in the population. For our computational tests, we implement crossover by interchanging $n/5$ two-element sections.

The *mutation* operator prevents loss of diversity in the population by randomly altering elements in the chromosomes that are generated under crossover. The number of elements that are changed depends upon a mutation probability P_{mut}. The expected total number of elements to be mutated is MnP_{mut} since each chromosome contains n elements. We invert $\lfloor MnP_{\text{mut}} \rfloor$ bits by repeatedly selecting a chromosome and a bit at random. The mutation probability in our computational tests is $P_{\text{mut}} = 0.001$.

We apply *simple elitism* to force the best chromosome found thus far to appear in the next generation. If this best chromosome does not appear, then it is introduced and a chromosome with the largest total weighted tardiness is removed.

We terminate the algorithm when the number of generations reaches the value l, where we set $l = n$ on the basis of

preliminary tests. However, we also use *aging* as a secondary stopping criterion to avoid superfluous iterations at the end of the algorithm. When the best solution in a population is not better than that of the previous population for a number of generations, which in our implementation is $n/2$, then the procedure is terminated.

4. Computational Experience

4.1 Test Problems

Test problems were generated as follows. For each job i, an integer processing time p_i was generated from the uniform distribution [1, 100] and an integer weight was generated from the uniform distribution [1, 10]. Problem "hardness" is likely to depend on the relative range of due dates (RDD) and on the average tardiness factor (TF). Having computed $P = \sum_{i=1}^{n} p_i$ and selected values of RDD and TF from the set {0.2, 0.4, 0.6, 0.8, 1.0}, an integer due date d_i from the uniform distribution $[P(1 - \mathrm{TF} - \mathrm{RDD}/2), P(1 - \mathrm{TF} + \mathrm{RDD}/2)]$ was generated for each job i. Five problems were generated for each of the 25 pairs of values of RDD and TF, yielding 125 problems for each value of n. Instances with $n = 40$, $n = 50$ and $n = 100$ were generated. We applied the branch-and-bound algorithm of Potts and Van Wassenhove[13] with the aim of finding an optimal solution value to each problem with $n = 40$ and $n = 50$ (for $n = 100$, computation times are prohibitive). With a computation time limit of 2 minutes, this method results in average computation times of 7 and 31 seconds, for $n = 40$ and $n = 50$, respectively. For $n = 40$, one problem remains unsolved, whereas for $n = 50$, this number increases to 22. For our computational tests, the local search heuristics were coded in C and run on a HP 9000-G50 computer.

Heuristics are compared by listing the average relative percentage deviation (ARPD) of the heuristic solution value from the optimal value, the maximum relative percentage deviation (MRPD) from the optimal value, the number of times (out of 125) that an optimal solution is found (NO), and the average computation time (ACT) in seconds. (For instances where the branch and bound algorithm does not find an optimal solution, we use the best known solution value instead of the optimal value.)

4.2 Parameter Selection

Much investigation on the design features and parameter settings of the different local search heuristics was undertaken in preliminary tests. In this section, we only summarize the most significant findings. The next section presents results that compare the different methods using "good" parameter settings.

For simulated annealing and threshold accepting, experiments with different values for K_1, K_l, v_1 and v_l show that the performance is fairly insensitive to these parameters provided that K_1 is large ($K_1 \geqslant 0.5$), v_1 is not too small ($0.01 \leqslant v_1 \leqslant 0.02$) and K_l and v_l are very small ($K_l \leqslant 0.001$ and $v_l \leqslant 0.001$).

In the tabu search method, a good characterization of the tabu status and the diversifying element are essential to achieve a good performance. The tabu list should be longer for the binary representation than for the permutation encoding, where a length of 7 is sufficient. The diversifying step should be executed more frequently for the binary encoding. Experiments with a tabu list in which its length changes dynamically indicate that there is no real advantage in using such a device.

Various implementations of a genetic algorithm using the permutation representation were tested. In most cases, the PMX and LOX crossover operators are ineffective in generating solutions that improve upon those in the initial population. Consequently, the solutions generated by such a genetic algorithm are substantially inferior to those obtained with the other local search methods. Thus, the permutation-encoded genetic algorithm is not considered further.

For our genetic algorithm with the binary representation, we have compared different reproduction schemes: roulette wheel selection, deterministic sampling and rank-based selection. The results show that, when linear scaling is applied, deterministic sampling outperforms the other two methods. Our results also indicate that the mutation probability should not be too large (possibly because our crossover operator incorporates a rather large diversifying element).

4.3 Comparative Results

If R denotes the representation of solutions and S is the number of starts in a multistart version of an algorithm, then we can adopt the following abbreviations.

AU:	the Apparent Urgency rule of Morton et al.[11] (see [10]) which performs quite well relative to other simple constructive heuristics.
$D(R, S)$:	a strict descent method where no neutral moves are accepted.
$DN(R, S)$:	a descent method which accepts all neutral moves.
$SA(R, S)$:	simulated annealing, as described in Section 2.1.
$TA(R, S)$:	threshold accepting, as described in Section 2.2.
$TS(R, S)$:	tabu search, as described in Section 2.3.
$GA(R, S)$:	the genetic algorithm, as described in Section 3.

In these abbreviations, R is replaced by P or B depending on whether the permutation or the binary representation is used.

Computational results for the two types of representation are listed in Tables I and II, respectively. The column labeled with l in these tables, gives the number of levels for DN, SA and TA, and the number of iterations for TS, and the maximum number of generations for GA. This number is chosen so that the different methods require roughly the same computation time, and is dependent on the number of jobs n, the neighborhood size, and the number of different starts S. For SA, TA and TS, we give two sets of results for each of $S = 1$ and $S = 5$; the second set is obtained by doubling l, which has the effect of running the algorithm for twice as long. However, for DN, doubling the number of levels has very little effect on the performance; therefore, we have doubled the number of different starts and give results for $S = 10$.

For the permutation representation, we observe from Table I that the simple AU rule and the single-start descent

Table I. Results for $n = 40$ and $n = 50$ with the permutation representation

		$n = 40$				$n = 50$			
Algorithm	l	ARPD	MRPD	NO	ACT	ARPD	MRPD	NO	ACT
AU		12.77	169.18	24	0.01	11.00	125.12	22	0.01
D(P, 1)		1.06	41.65	74	0.02	0.59	10.99	57	0.04
D(P, 2n)		0.04	4.21	120	1.73	0.00	0.26	119	4.42
DN(P, 1)	4n	0.57	19.27	80	1.20	0.66	13.30	56	2.79
DN(P, 5)	4n/5	0.02	0.81	105	1.20	0.03	0.84	91	2.80
DN(P, 10)	4n/5	0.01	0.41	115	2.40	0.02	0.84	102	5.60
SA(P, 1)	3n	0.23	8.72	114	1.30	0.17	11.20	101	3.03
SA(P, 5)	3n/5	0.09	6.08	117	1.34	0.10	11.20	108	3.11
SA(P, 1)	6n	0.11	8.72	120	2.58	0.11	11.20	104	6.03
SA(P, 5)	6n/5	0.01	0.81	121	2.61	0.09	11.20	115	6.11
TA(P, 1)	4n	0.23	8.72	109	1.32	0.24	11.20	100	3.14
TA(P, 5)	4n/5	0.00	0.17	121	1.36	0.09	11.20	115	3.23
TA(P, 1)	8n	0.15	6.70	113	2.62	0.23	11.20	101	6.27
TA(P, 5)	8n/5	0.06	6.70	121	2.70	0.01	0.28	111	6.42
TS(P, 1)	2n^2	0.06	6.70	115	1.35	0.01	0.42	111	2.99
TS(P, 5)	2n^2/5	0.00	0.33	118	1.32	0.01	0.28	113	2.95
TS(P, 1)	4n^2	0.00	0.25	121	2.65	0.01	0.36	115	5.99
TS(P, 5)	4n^2/5	0.00	0.17	123	2.64	0.00	0.16	118	5.92

method (D(P, 1)) are not effective. There is little improvement when neutral moves are accepted (DN(P, 1)); these neutral moves are not sufficient to allow the search to escape from a local optimum. However, the multistart versions of descent are quite good. The descent method with many restarts (D(P, 2n)) finds an optimal solution for almost all of the test problems.

For the other three neighborhood search methods with the permutation representation, the best results are obtained with tabu search. It is the only method where the single-start version (TS(P, 1)) can compete with the multistart descent methods. The multistart version (TS(P, 5)) performs a little bit better than TS(P, 1). For simulated annealing and threshold accepting, it appears preferable to use multistart with fewer levels, rather than execute one long run. Note that SA and TA exhibit some erratic behaviour as exemplified by their large MRPD values, even for multistart versions of the algorithms. They are clearly less robust than multistart TS.

The results in Table II for the binary representation are quite similar to those in Table I. We again observe a very good performance for the multistart descent methods (D(B, 2n) and DN(B, 10)). Multistart versions are clearly necessary for all neighborhood search methods to achieve low MRPD values.

A comparison of results for the two types of representation shows that, on average, the methods that use the permutation representation produce slightly better NO values (except for DN(P, S)) than those for the binary representation. However, the multistart methods have lower MRPD values under the binary representation.

The genetic algorithm GA(B, 1) performs quite well and uses less computation time than the other methods (except for simple descent). One reason for this quick convergence to good quality solutions is that the initial population is not completely random but uses some information about the problem. Performing more iterations (by omitting the aging test) or enlarging the population size does not result in a significant improvement. The multistart version GA(B, 5) improves upon the single-start version GA(B, 1): more problems are solved optimally and a lower MRPD is obtained.

Generally, it can be concluded from Table II that the multistart versions of all local search methods perform very well. The differences between methods are quite small (certainly smaller than in Table I). It appears, therefore, that binary encoding leads to fairly robust behaviour. The reason for this could be due to the representation, since the decoding heuristic incorporates problem-specific knowledge. Although reasonably good solutions can be obtained from each of the neighborhood search methods under both encodings, the performance of the genetic algorithm is very poor under the permutation representation but extremely good under our new binary encoding scheme.

The different local search heuristics all use some randomization: multistart methods for their starting solutions, simulating annealing in its acceptance rule, tabu search (TS(P, S)) in its neighbor generation, and the crossover and mutation operator in the genetic algorithm. In Tables I and II, we report on the results of a single run, where the same initial seed of the random number generator is used for each method. In order to investigate the effect of this initial seed on the performance of the methods, results for five runs with different seeds were obtained. For the single-start versions, there is indeed a significant effect. As an illustration, one run of SA(B, 1) with $n = 40$ results in a MRPD of 77.09, while

Table II. Results for $n = 40$ and $n = 50$ with the binary representation

		$n = 40$				$n = 50$			
Algorithm	l	ARPD	MRPD	NO	ACT	ARPD	MRPD	NO	ACT
D(B, 1)		2.56	66.38	55	0.04	1.96	15.64	40	0.06
D(B, 2n)		0.00	0.05	121	2.36	0.01	0.18	112	5.51
DN(B, 1)	2n	0.33	20.04	108	1.27	0.12	6.08	96	2.77
DN(B, 5)	2n/5	0.00	0.45	121	1.28	0.01	0.21	109	2.73
DN(B, 10)	2n/5	0.00	0.05	123	2.52	0.01	0.19	114	5.50
SA(B, 1)	2n	0.18	20.04	107	1.34	0.05	3.42	94	2.98
SA(B, 5)	2n/5	0.01	0.16	110	1.35	0.02	0.30	98	3.00
SA(B, 1)	4n	0.17	20.04	111	2.68	0.06	6.51	99	5.93
SA(B, 5)	4n/5	0.00	0.07	116	2.66	0.01	0.45	101	6.00
TA(B, 1)	2n	0.41	20.04	106	1.33	0.04	2.44	100	2.92
TA(B, 5)	2n/5	0.00	0.32	116	1.31	0.01	0.19	107	2.82
TA(B, 1)	4n	0.36	20.04	106	2.64	0.03	2.44	99	5.84
TA(B, 5)	4n/5	0.00	0.05	119	2.60	0.01	0.18	108	5.69
TS(B, 1)	4n	0.20	18.63	109	1.23	0.04	2.17	104	2.63
TS(B, 5)	4n/5	0.00	0.21	119	1.33	0.01	0.36	107	2.79
TS(B, 1)	8n	0.20	18.63	111	2.44	0.04	2.17	108	5.20
TS(B, 5)	8n/5	0.00	0.21	122	2.55	0.01	0.36	115	5.37
GA(B, 1)	n	0.01	0.45	111	0.32	0.02	0.75	107	0.78
GA(B, 5)	n	0.00	0.08	119	1.40	0.01	0.19	113	3.12

another run yields a MRPD of 0.64. The MRPD values of the multistart versions of DN, SA and TA are also sensitive to the initial seed. However, the effect is small for multistart TS and for the genetic algorithm (GA(B, 1) and GA(B, 5)): different runs have performance values which are quite close to those given in Tables I and II.

It is not easy to indicate which method is best. When only limited computation time is available, it is preferable to use the binary encoded genetic algorithm because of its robustness (small MRPD). When long runs are possible, multistart tabu search with the permutation representation is a good alternative.

The different local search heuristics were also tested on the larger problems with $n = 100$. Results are shown in Table III, where ARPD, MRPD and NO now relate to the best heuristic solution value found since the branch and bound algorithm cannot deal with 100-job problems. As in Tables I and II, the number of levels or iterations l is chosen so that each method requires approximately the same computation time.

Table III confirms our previous findings. Tabu search with the permutation representation has the best NO value, but there are a few problems for which poor quality solutions are generated (a large MRPD). Large MRPD values can be observed for all methods under the permutation representation, but these methods have better NO values than for those under the binary representation. Conversely, very good MRPD values can be observed for TS(B, 1) and TS(B, 5) but their NO values are rather disappointing. Thus, the methods that use the binary representation are robust (consistent), while permutation-based methods have a higher likelihood of finding an optimal solution. Both TS and GA perform well. For both representations, TA appears to be slightly better than SA, although the performance of both is slightly inferior to DN. A possible explanation is that DN employs 10 different starts, whereas the other two methods only have 5.

5. Concluding Remarks

This paper compares the performance of a number of local search heuristics for the problem of scheduling a single machine to minimize total weighted tardiness. New features in the implementation of these heuristics include the binary representation and the additional diversifying element in the tabu search methods.

Extensive computational tests are used to compare the 'natural' permutation representation with binary encoding. The performance of the various methods under both representations is quite good. In order to obtain higher quality solutions, multistart versions are preferable to single-start longer run versions. On average, the permutation-based methods have a higher likelihood of generating an optimal solution (large NO), whereas the methods that use a binary representation have smaller maximum relative percentage deviations from the optimal solution value (low MRPD).

The performance of simulated annealing and threshold accepting is comparable with that of multistart descent,

Table III. Results for $n = 100$

	Permutation representation					Binary representation				
Algorithm	l	ARPD	MRPD	NO	ACT	l	ARPD	MRPD	NO	ACT
AU		16.24	208.77	22	0.1					
D(R, 1)		0.45	15.04	38	0.3		1.93	18.76	27	0.5
D(R, 100)		0.06	4.78	86	39.2		0.06	1.85	57	40.1
DN(R, 10)	$2n/5$	0.07	4.78	72	40.1	$n/4$	0.04	1.85	69	40.7
SA(R, 5)	$n/2$	0.12	4.78	59	40.5	$9n/20$	0.16	1.05	59	40.8
TA(R, 5)	$3n/5$	0.10	4.78	70	40.1	$9n/20$	0.06	1.86	64	39.6
TS(R, 1)	$2n^2$	0.06	4.78	96	38.3	$6n$	0.03	0.45	66	41.4
TS(R, 5)	$2n^2/5$	0.04	4.39	103	37.6	$6n/5$	0.04	0.34	66	45.1
GA(R, 1)						n	0.07	2.17	69	10.3
GA(R, 5)						n	0.03	0.76	77	37.3

although the two former methods sometimes show a somewhat erratic behavior. Tabu search (TS(P, S)) dominates other neighborhood search methods. Our binary encoded genetic algorithm (GA(B, S)) generates solutions of very good quality. It is a viable alternative to other heuristic methods, especially in view of its small maximum relative deviations and modest computation times. Multistart tabu search and the genetic algorithm have the added advantage of reliability since their performance is not very sensitive to the random number seed.

In conclusion, the natural representation of solutions in local search methods is not necessarily the most effective. The design of the method should include an evaluation of alternative representations so that the best choice can be made. For the single machine total weighted tardiness problem, our new binary encoding scheme produces very robust results. A major contributory factor to its success is the heuristic method which decodes the binary representation of a solution into a sequence of jobs. Tabu search methods which use either the permutation or the binary representation also perform well.

Acknowledgments

The authors are grateful to Richard Congram and to four anonymous referees whose constructive comments have been used to improve this paper. This research was partly supported by the International Association for the Promotion of Cooperation with Scientists from the Independent States of the Former Soviet Union, INTAS-93-257 and INTAS-93-257-Ext.

References

1. T.S. Abdul-Razaq, C.N. Potts, and L.N. Van Wassenhove, 1990. A Survey of Algorithms for the Single Machine Total Weighted Tardiness Scheduling Problem, *Discrete Applied Mathematics 26*, 235–253.
2. F. Della Croce, R. Tadei, and G. Volta, 1995. A Genetic Algorithm for the Job Shop Problem, *Computers & Operations Research 22*, 15–24.
3. G. Dueck and T. Scheuer, 1990. Threshold Accepting: A General Purpose Optimization Algorithm Appearing Superior to Simulated Annealing, *Journal of Computational Physics 90*, 161–175.
4. R.W. Eglese, 1990. Simulated Annealing: A Tool for Operational Research, *European Journal of Operational Research 46*, 271–281.
5. F. Glover, 1989. Tabu Search—Part I, *ORSA Journal on Computing 1*, 190–206.
6. F. Glover, 1990. Tabu Search: A Tutorial, *Interfaces 20*, 74–94.
7. D.E. Goldberg, 1989. *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, Reading, MA.
8. E.L. Lawler, 1977. A "Pseudopolynomial" Algorithm for Sequencing Jobs to Minimize Total Tardiness, *Annals of Discrete Mathematics 1*, 331–342.
9. J.K. Lenstra, A.H.G. Rinnooy Kan, and P. Brucker, 1977. Complexity of Machine Scheduling Problems, *Annals of Discrete Mathematics 1*, 343–362.
10. T.E. Morton and D.W. Pentico, 1993. *Heuristic Scheduling Algorithms*, Wiley, NY.
11. T.E. Morton, R.M. Rachamadugu, and A. Vepsalainen, 1984. Accurate Myopic Heuristics for Tardiness Scheduling, GSIA Working Paper No. 36-83-84, Carnegie-Mellon University, PA.
12. M. Pirlot, 1992. General Local Search Heuristics in Combinatorial Optimization: A Tutorial, *Belgian Journal of Operations Research, Statistics and Computer Science 32*, 8–67.
13. C.N. Potts and L.N. Van Wassenhove, 1985. A Branch and Bound Algorithm for the Total Weighted Tardiness Problem, *Operations Research 33*, 363–377.
14. C.N. Potts and L.N. Van Wassenhove, 1991. Single Machine Tardiness Sequencing Heuristics, *IIE Transactions 23*, 346–354.