LOCAL SEARCH HEURISTICS FOR SINGLE MACHINE SCHEDULING WITH BATCHING TO MINIMIZE THE NUMBER OF LATE JOBS

by
H.A.J. CRAUWELS*
C.N. POTTS**
and
L.N. VAN WASSENHOVE†

95/73/TM
(revised version of 94/45/TM)

* Professor at KIHDN, J. De Nayerlaan 5, 2860 Sint-Katelijne-Waver, Belgium.

** Professor, Faculty of Mathematical Studies at the University of Southampton, Southampton SO17 1BJ, UK.

†† Professor of Operations Management and Operations Research at INSEAD, Boulevard de Constance, Fontainebleau 77305 Cedex, France.

Printed at INSEAD, Fontainebleau, France

Local Search Heuristics for Single Machine Scheduling with Batching to Minimize the Number of Late Jobs

H. A. J. Crauwels
KIHDN
J. De Nayerlaan 5
2860 Sint-Katelijne-Waver, Belgium

C. N. Potts
Faculty of Mathematical Studies
University of Southampton
Southampton SO17 1BJ, United Kingdom

L. N. Van Wassenhove
INSEAD
Boulevard de Constance
77305 Fontainebleau, France

Abstract

Local search heuristics are developed for a problem of scheduling jobs on a single machine. Jobs are partitioned into families, and a set-up time is necessary when there is a switch in processing jobs from one family to jobs of another family. The objective is to minimize the number of late jobs. Four alternative local search methods are proposed: multi-start descent, simulated annealing, tabu search and a genetic algorithm. The performance of these heuristics is evaluated on a large set of test problems. The best results are obtained with the genetic algorithm; multi-start descent also performs quite well.

Keywords: Scheduling; Single machine; Batches: Set-up time; Local search heuristics; Descent; Simulated annealing; Tabu search; Genetic algorithm

1 Introduction

Many practical scheduling problems involve processing several families of related jobs on common facilities, where a set-up time is incurred whenever there is a switch from processing a job in one family to a job in another family. For example, consider a mechanical parts manufacturing environment in which jobs have to be sequenced for processing on a multi-tool machine. Whenever production changes to a new family of jobs, time is spent in retooling the machine. Based on the principles of group technology, it is conventional to schedule contiguously all jobs from the same family. However, this is not necessarily the best strategy. It may be better to partition each family of jobs into several batches, where all jobs of a batch are scheduled contiguously,

and then schedule the batches. A review of algorithms and complexity for a variety scheduling problems that require the batching of jobs is given by Potts and Van Wassenhove [10].

In this paper, we consider a single machine scheduling problem in which every job belongs to a specified family. Also, each job is available for processing at time zero, and has a given processing time and due date. A sequence independent set-up time, which is associated with the family of the job to be processed next, is necessary before the first job is processed and whenever there is a switch in processing jobs from one family to jobs of another family. The objective is to find a schedule which minimizes the number of late jobs.

Monma and Potts [6] consider a variety of single machine scheduling problems under the assumption that sequence dependent set-up times for families satisfy the 'triangle inequality': the set-up time associated with a changeover from family f to h is assumed to take no longer than that for changeover from family f to g followed by a changeover from family g to h (our sequence independent set-up times clearly satisfy this triangle inequality). They present a dynamic programming algorithm which is polynomially-bounded in the number of jobs, but exponential in the number of families. However, this dynamic programming algorithm is largely of theoretical interest unless the number of families is very small.

For an arbitrary number of families, the problem is NP-hard (Bruno and Downey [1]). Various attempts by the authors to solve problem instances of a reasonable size using a branch and bound approach indicate its challenging nature. Therefore, it is worthwhile to investigate some heuristic techniques. Over the past ten years, several new local search methods have been proposed and developed: simulated annealing, tabu search and genetic algorithms. An introductory overview of these methods and some applications are given by Pirlot [8].

In this paper, several neighbourhood structures are described and used in descent, simulated annealing and tabu search algorithms. A genetic algorithm is also proposed. In a practical setting, these methods would be used in an on-line scheduling system where the schedule is updated when a new order is received. Thus, our main focus of attention is to develop local search heuristics which can generate reasonably good quality solutions using limited computational resources.

In Section 2, we give a formal statement of our problem, and describe some dominance criteria. Sections 3, 4, 5 and 6 describe the different local search heuristics: descent, simulated annealing, tabu search and a genetic algorithm. Section 7 reports on computational experience, and some concluding remarks are given in Section 8.

2 Problem formulation and dominance criteria

To state our scheduling problem more precisely, we are given N jobs that are divided into F families. Each family f, for $1 \leq f \leq F$, contains n_f jobs. The jobs are numbered $1, \ldots, N$. Sometimes it is more convenient to refer to job (i, f), which is the i'th job in family f, for $1 \leq i \leq n_f$. All jobs are available for processing at time zero, and are to be scheduled on a single machine. We let p_{if} denote the processing time of job (i, f), and d_{if} is its due date. A sequence independent set-up time s_f is incurred whenever a job in family f is processed immediately after a job in a different family. Also, an initial set-up time s_f is required if a job from family f is the first to be processed. Any job (i, f) which is completed no later than time d_{if} is *early*; otherwise it is *late*. The objective is to find a schedule which minimizes the number of late jobs.

As an example, consider a problem with 4 jobs that are divided into 2 families containing jobs $\{1, 2\}$ and $\{3, 4\}$, respectively. Let $s_1 = 3$ and $s_2 = 4$. Also, the processing times are 5, 7, 10 and 3, and the due dates are

10, 25, 15 and 20, respectively. Without splitting any family into batches, either jobs 1 and 2, or jobs 3 and 4 are late. In the optimal solution, however, both families are split in two batches, resulting in the sequence (1,4,2,3) which has just one job late.

We can restrict our attention to sequences S of the form $S = (E, L)$, where a partial sequence E of early jobs is followed by a partial sequence L of late jobs. The partial sequence E can be regarded as a series of batches, where a batch is a subsequence of jobs from the same family preceded and succeeded by a job from a different family (unless it is the first or last batch in E). If there are r batches of early jobs in S, which are labelled $B_1, \ldots, B_r$, then

$$S = (B_1, \ldots, B_r, L).$$

Each batch B_b, for $1 \leq b \leq r$, can be viewed as a single composite job with processing time T_b and due date D_b. If batch B_b contains jobs $(\pi(u), f), \ldots, (\pi(v), f)$, then T_b and D_b are calculated as follows:

$$T_b = s_f + \sum_{h=u}^{v} p_{\pi(h),f}, \qquad D_b = \min_{\nu \in \{u,\ldots,v\}} \left\{ d_{\pi(\nu),f} + \sum_{h=\nu+1}^{v} p_{\pi(h),f} \right\}. \tag{1}$$

The following result justifies this choice of T_b and D_b.

Lemma 1 *All jobs of a batch are early if and only if the batch is completed by its due date.*

Proof. Let $C(B_b)$ be the completion time of batch B_b which contains jobs $(\pi(u), f), \ldots, (\pi(v), f)$, and let $C_{\pi(\nu),f}$ denote the completion time of job $(\pi(\nu), f)$, for $u \leq \nu \leq v$. We show that $C(B_b) \leq D_b$ if and only if each of the individual jobs $(\pi(u), f), \ldots, (\pi(v), f)$ is early.

First, suppose that $C(B_b) \leq D_b$. For any job $\pi(\nu)$ of B_b, it follows from the definition of D_b that

$$D_b \leq d_{\pi(\nu),f} + \sum_{h=\nu+1}^{v} p_{\pi(h),f}.$$

We deduce that job $(\pi(\nu), f)$ is early because

$$C_{\pi(\nu),f} = C(B_b) - \sum_{h=\nu+1}^{v} p_{\pi(h),f} \leq D_b - \sum_{h=\nu+1}^{v} p_{\pi(h),f} \leq d_{\pi(\nu),f}.$$

Conversely, suppose that $C(B_b) > D_b$, and that

$$D_b = d_{\pi(\nu),f} + \sum_{h=\nu+1}^{v} p_{\pi(h),f}$$

for some $\nu \in \{u, \ldots, v\}$. Then

$$C_{\pi(\nu),f} = C(B_b) - \sum_{h=\nu+1}^{v} p_{\pi(h),f} > D_b - \sum_{h=\nu+1}^{v} p_{\pi(h),f} = d_{\pi(\nu),f},$$

which shows that job $(\pi(\nu), f)$ is late. □

We can now state two theorems on the structure of an optimal solution. The first result specifies the ordering of early jobs within each family. In the second result, an optimal ordering of early batches is given.

Theorem 1 (Monma and Potts [6]) *There exists an optimal schedule where the early jobs within each family are sequenced in non-decreasing due date order (intragroup EDD property).*

Theorem 2 *There exists an optimal schedule where the early batches are sequenced in non-decreasing due date order with respect to the corresponding composite job due dates (intergroup EDD property).*

Proof. Lemma 1 establishes that all jobs of a batch are early if the batch itself is early. Moreover, a straightforward adjacent interchange argument for batches verifies the existence of an optimal schedule in which early batches are sequenced in EDD order. □

Throughout this paper, we assume that the jobs within each family have been renumbered so that $d_{1f} \leq \ldots \leq d_{n_f,f}$, for $1 \leq f \leq F$.

Using the above theorems as dominance conditions, an initial solution for a neighbourhood search heuristic can be constructed as follows.

Algorithm A

Step 1. Sequence the jobs in EDD order. Apply the algorithm of Moore [7] to minimize the number of late jobs, ignoring the fact that they belong to different families. A sequence with many set-ups and a relatively small number of early jobs is likely to be generated.

Step 2. To reduce the number of set-ups, batches which contain only one job are shifted forward and backward to batches which belong to the same family. For example, consider a sequence of early jobs which has the form

$$(\ldots,(i,f),\ldots,(r,g),(i+1,f),\ldots,(j-1,f),(s,h),\ldots,(j,f),\ldots).$$

If job (i,f) by itself forms a batch, then this job is shifted to the position just before $(i+1,f)$ if it is still early. This backward shift is attempted for all batches with one job. For the remaining batches with one job, a forward shift is tried; for example, if job (j,f) forms a batch, then this job is shifted to the position just after $(j-1,f)$ if all jobs starting from (s,h) remain early.

Reorder the batches using Theorem 2 so that the intergroup EDD property is satisfied. Add as many jobs as possible from the late set using Algorithm Insert which is described below.

Step 3. Incorporate sequentially an additional split. Consider a batch

$$B_b = ((h,f),\ldots,(i-1,f),(i,f),\ldots,(j,f)),$$

where $h < j$. A split is performed between jobs $(i-1,f)$ and (i,f), for $h < i \leq j$, when $d_{i-1,f}+p_{i,f} < d_{i,f}$ and $C_{i,f}+s_f < d_{i,f}$. The sub-batches $((h,f),\ldots,(i-1,f))$ and $((i,f),\ldots,(j,f))$ are resequenced using Theorem 2 so that the intergroup EDD property is satisfied. If $((i,f),\ldots,(j,f))$ is sequenced before B_{b+1}, then this split is disregarded.

Try to add as many jobs as possible from the late set using Algorithm Insert which is described below. If no improvement can be made, then the split is disregarded and batch B_b is reformed by combining the two sub-batches.

Step 4. Attempt to improve the resulting sequence by selecting a batch containing a single job (if such a batch exists) which has the longest set-up plus processing time, setting it to be late, and repeating Step 3.

An initial solution can also be generated by randomly choosing which jobs are early and how the batches are constructed.

Algorithm B

Step 1. For each job i, where $1 \leq i \leq N$, generate two random numbers r_{i1} and r_{i2}. If $r_{i1} \leq 0.5$, then job i is considered to be early; otherwise, it is late. If job i is early and $r_{i2} \leq 0.5$, then the corresponding family is split after job i. This process constructs several batches of early jobs.

Step 2. For each batch B_b of early jobs, compute its processing time T_b and due date D_b from (1). Apply the algorithm of Moore [7] to minimize the number of late batches.

Step 3. Improve the resulting sequence with Algorithm Insert.

Algorithm Insert, which is used in several steps of Algorithm A and B to add as many jobs as possible from the late set, is now described.

Algorithm Insert

Step 1. Let L be a list of late jobs, and let $E = (\pi(1), \ldots, \pi(n_e))$, where n_e is the number of early jobs. Set $e = 1$, and $t = 0$.

Step 2. Compute latest start times for the jobs of E using $v_{\pi(n_e)} = d_{\pi(n_e)} - p'_{\pi(n_e)}$, and $v_{\pi(i)} = \min\{v_{\pi(i+1)}, d_{\pi(i)}\} - p'_{\pi(i)}$ for $i = n_e - 1, n_e - 2, \ldots, 1$, where $p'_{\pi(i)} = p_{\pi(i)}$ if $i > 1$ and $\pi(i)$ and $\pi(i-1)$ belong to the same family; $p'_{\pi(i)} = p_{\pi(i)} + s_f$ otherwise, where f is the family to which job $\pi(i)$ belongs.

Step 3. Consider a job j from the late set, belonging to family f, which satisfies the intragroup EDD property of Theorem 1 when inserted immediately before $\pi(e)$. Compute $t' = t + p'_j$, where $p'_j = p_j$ if $e > 1$ and $\pi(e-1)$ is in family f; $p'_j = p_j + s_f$ otherwise. Also, set $v'_{\pi(e)} = v_{\pi(e)} - s_g$ if $e > 1$, and $\pi(e-1)$ and $\pi(e)$ both belong to some family g, where $g \neq f$; $v'_{\pi(e)} = v_{\pi(e)} + s_f$ if $\pi(e)$ belongs to family f, and either $e = 1$ or $e > 1$ and $\pi(e-1)$ does not belong to family f; $v'_{\pi(e)} = v_{\pi(e)}$ otherwise. If $t' \leq v'_{\pi(e)}$ and $t' \leq d_j$, insert job j immediately before job $\pi(e)$, unless job j can be inserted in a later position just before a batch with jobs of the same family f and it remains early (in which case the job will be inserted at a later stage). If job j is inserted, set $L = L - \{j\}$, $n_e = n_e + 1$, $\pi(i) = \pi(i-1)$ for $i = n_e, n_e - 1, \ldots, e+1$, $\pi(e) = j$, $v_{\pi(e+1)} = v'_{\pi(e)}$, and $v_{\pi(e)} = \min\{v_{\pi(e+1)}, d_{\pi(e)}\} - p'_{\pi(e)}$.

Repeat Step 3 for all jobs from L.

Step 4. Set $t = t + p'_{\pi(e)}$, where $p'_{\pi(e)}$ is defined in Step 2, and set $e = e + 1$. If $e \leq n_e$, then go to Step 3. For all remaining jobs in L, add as many of them as possible to the end of the sequence if they are early.

3 Neighbourhood Search

A neighbourhood search algorithm has the following structure. At each stage, a current solution is specified. A neighbour of this solution is generated by some suitable mechanism, and some acceptance rule is used to decide on whether it should replace the current solution. This process is repeated until some termination criterion is satisfied.

In any neighbourhood search algorithm, it is necessary to specify how an initial current solution is obtained. In line with our philosophy of using neighbourhood search to obtain good solutions at modest computational expense, it is preferable to have a reasonably good initial solution. Such a solution can be obtained using Algorithm A. However, for cases where a neighbourhood search algorithm is to be applied several times using different starting solutions, Algorithm B can be used whenever a new initial solution is required.

We now discuss how solutions are represented and the choice of neighbourhood. Unless there are strong reasons for deciding otherwise, it is usually best to adopt a natural representation. For our problem, a natural

representation is a sequence of early jobs. Also, for the efficient implementation of a neighbourhood search algorithm, structural knowledge of the problem should be exploited. Thus, for our problem, neighbourhood solutions should satisfy the EDD rules of Theorems 1 and 2. These intuitions are supported by the results of initial experiments in which a 'standard' approach for sequencing problems is adopted (see Potts and Van Wassenhove [9], for example): solutions are represented by complete sequences of jobs (including those that are late), and neighbourhood solutions are obtained by interchanging two jobs or moving a job to a different position in the sequence. The poor performance of such an approach is attributed to the fact that many neighbours do not satisfy the EDD properties.

We propose neighbourhoods in which one or more jobs of a batch are removed from the current sequence and replaced by some jobs that are previously late. For this replacement, we use Algorithm Insert which is described in the previous section. Step 3 of this algorithm ensures that only sequences of early jobs that satisfy the intragroup EDD property of Theorem 1 are generated.

To specify our neighbourhoods more precisely, let $S = (\pi(1), \ldots, \pi(n_e))$ denote the current sequence of early jobs, and let $S = (B_1, \ldots, B_r)$, where $B_1, \ldots, B_r$ are the batches for S. In each of our neighbourhood search algorithms, we perform tests with two neighbourhood structures.

In the *job* neighbourhood, a single job $\pi(i)$, for $i = 1, \ldots, n_e$, is removed from S. The search is done systematically, starting with the removal of $\pi(1)$. After Algorithm Insert is applied (where jobs of the original late set L are candidates for insertion, but the job that has just been removed is not), the resulting sequence is a neighbour. To construct the other neighbours, jobs $\pi(2), \ldots, \pi(n_e)$ are removed in turn from S, and then Algorithm Insert is applied.

In the *batch* neighbourhood, there are r neighbours corresponding to the systematic removal of batches $B_1, \ldots, B_r$. After the removal of a batch B_b, for $b = 1, \ldots, r$, Algorithm Insert is applied twice, first for inserting jobs of the original late set L, and second for reinserting jobs of B_b, but excluding the first job of this batch. We employ this second application of Algorithm Insert because there is a possibility that B_b contains more jobs than are in L, which would result in more late jobs than for the current solution under a single application of Algorithm Insert.

Note that the batch neighbourhood, which has size r, is generally smaller than the job neighbourhood, which has size n_e. In both cases, we refer to the complete search of the neighbourhood as a *level*.

In a *descent* method, a series of moves from one solution to another solution in its neighbourhood 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 would terminate. For our problem, however, neighbours frequently have the same objective function value as the current solution. Such neighbours are also accepted in our procedure. Note that this could result in cycling, although this did not pose a problem in our experiments. The stopping criterion is defined by a fixed number of levels, i.e., the complete neighbourhood is searched a fixed number of times. Although the resulting solution is normally a local optimum, it is not necessarily a global optimum. A classical remedy for this drawback is to perform multiple trials of the procedure, starting from different initial solutions, and select the best sequence as final solution. Such an approach is called *multi-start descent*.

Another possibility is to allow moves which lead to an increase in the objective function value. Consequently, the procedure can escape from a local optimum and continue its search for a global optimum. This idea is implemented in our simulated annealing and tabu search algorithms which are described in the following two sections.

4 Simulated annealing

In a simulated annealing procedure, solutions which improve upon the current solution value are accepted, while those which cause a deterioration in the objective function value are accepted according to a given probabilistic acceptance function. Following the lead of Kirkpatrick et al. [5] who suggest simulated annealing as a solution method for the traveling salesman problem, many papers are devoted to both the theoretical and computational aspects of this approach. A review is given by Eglese [2]. 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 in the objective function value, and t is a parameter which is known as the *temperature*.

Let l denote the number of levels that are considered, where each level i, for $1 \leq i \leq l$, has a temperature t_i. In our implementation of simulated annealing, t_i is derived from an acceptance probability K_i, as described by Potts and Van Wassenhove [9]. More precisely, K_i is equal to the probability of accepting a solution with one additional late job. We set $K_i = ((l-i)K_1 + (i-1)K_l)/(l-1)$, for $1 \leq i \leq l$, where K_1 and K_l are the initial and final acceptance probabilities. In order to compute the actual acceptance probability for any given increase in the objective function value at level i, the temperature t_i is found using $K_i = \exp(-1/t_i)$, thereby yielding $t_i = -1/\ln K_i$.

For each acceptance probability level, the complete neighbourhood is searched systematically as described in the previous section. We perform experiments with both the job and batch neighbourhoods defined above. The termination criterion is defined by fixing a priori the number of levels l.

5 Tabu search

A deterministic approach for escaping from a local optimum is offered by tabu search (see Glover [3]), which allows the possibility of accepting solutions which cause a deterioration in the objective function value. However, certain moves are forbidden or *tabu* for a few iterations. Although the main purpose of specifying tabu moves is to prevent cycling, an additional benefit is to direct the search into unexplored regions of the solution space.

Conventionally in tabu search, the complete neighbourhood must be searched at each iteration (level) to find the best non-tabu move. In our implementation, however, we truncate the search by accepting the first non-tabu neighbour which improves upon the objective function value of the current solution. Due to the computational expense of searching the complete neighbourhood when no improving non-tabu neighbour exists, it may be anticipated that the smaller batch neighbourhood is preferred to the job neighbourhood. Nevertheless, we perform experiments with both neighbourhood structures.

A tabu list is constructed, which characterizes which moves are tabu. We perform tests with two types of tabu list, both of which classify the past few current solutions as tabu. In the first type, when the search moves from a current solution to a neighbour, the set of early jobs for the current solution are stored in the list. If a subsequent neighbour contains a set of early jobs that appears on the list, then this neighbour is tabu. In the second type of tabu list, the jobs that become early when a move from the current solution to a neighbour is executed are added to the tabu list. This tabu list fixes jobs that are to remain early, so a neighbour is tabu if any of its late jobs appear in the tabu list.

The length of the tabu list determines the number of iterations during which an entry remains in the list. Because of the more specific characteristics of a tabu move in the first type of list, a relatively long list is

required: we set the length to be half of the total number of iterations that are performed. For the second type, the classical tabu list length of 7 is satisfactory.

To prevent the occasional loss of good solutions, an aspiration level criterion is incorporated. If the solution value of a tabu neighbour is better than that for all solutions already generated, then its tabu status is overridden.

The last parameter to be fixed is the stopping rule. The search terminates after the execution of a prespecified number of iterations.

6 A genetic algorithm

Genetic algorithms, which are 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, are given by Goldberg [4].

We now discuss how solutions are represented. Previous research on applying genetic algorithms to sequencing problems indicates that approaches in which solutions are represented as sequences tend not to perform well, which is partly due to the difficulty in applying the standard genetic operators to such representations. On the other hand, a binary representation in which some structural properties of the problem are used to convert the representation into a solution is likely to be far more effective. We adopt this latter approach.

Because the order of the early jobs within each family is known, and the order of the batches is easily computed from Theorem 2, knowledge of which jobs are early and how these jobs are composed into batches allows a solution to be constructed. We propose a binary representation in which there are two elements for each job. These two elements are adjacent in the chromosome; a chromosome thus consists of N groups, each comprising two elements. If the first element for a particular job is set to 1, then the job is considered early; otherwise, it is late. Also, if the second element is 1, then this job ends a batch; otherwise it is not the end job of a batch. An arbitrary chromosome resulting from this representation does not necessarily correspond to a feasible sequence. To overcome the problem of infeasibility, we adopt the following approach.

First, the representation is used to choose the early jobs (using the first element of each pair), and then partition them into batches (using the second element for each early job). For these batches, we compute the processing time and due date from (1). To achieve the maximum number of early batches, the algorithm of Moore [7] is applied, from which we evaluate the number of late jobs that is used in the fitness evaluation of the chromosome.

A population consists of M chromosomes. The first chromosome of the initial population is created specially: all 'early job' elements are set to 1 and all 'split' elements are set to 0. With this chromosome, some problems can be solved very quickly if no late jobs are detected. The other $M-1$ chromosomes of the initial population are created analogously to Step 1 of Algorithm B of Section 2, but more appropriate probabilities are used to determine if a job is early and if a split is incorporated. The number of early job elements set to 1 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 $P = \sum_{f=1}^{F} n_f s_f/2 + \sum_{i=1}^{N} p_i$ is an estimate for the total set-up plus processing

time. With a relatively small TF, more jobs can be early and more early job elements should be set to 1 in the chromosome. Thus, a probability of 1 − TF is used to decide if the early job element is set to 1. For the split elements, a probability of RDD × TF is used, where RDD = $\min\{(\max_{i=1,\ldots,N}\{d_i\} - \min_{i=1,\ldots,N}\{d_i\})/P, 1\}$, is the relative due date range. If RDD is relatively large, then there is more time available for additional set-ups and so more split elements should be set to 1. On the other hand, with a larger TF, the average due date is smaller, and then it is preferable to create many small batches by setting more split elements to 1 so that the impact on the number of late jobs is kept to a minimum when it becomes necessary to discard batches to ensure feasibility.

The general structure of the genetic algorithm can be summarized as follows.

Genetic algorithm

Step 1. Create an initial population and set it to be the current population. Evaluate fitness values for the current population.

Step 2. Generate a new population from the current population using the genetic operators reproduction, crossover and mutation.

Step 3. Evaluate the new population. Set the current population to be the new population. If the maximum number of generations has not been executed, then go to Step 2.

Step 4. Optionally apply an improvement procedure in which all elements of the final population are taken as an initial sequence for an improvement procedure that executes Step 4 (and uses Step 3) of Algorithm A of Section 2.

During *evaluation*, the number of late jobs is computed for each chromosome of the population using the method described above. Let $V_1, \ldots, V_M$ denote these values, and let $f_1, \ldots, f_M$ be corresponding numbers of early jobs, where $f_k = N - V_k$ for $k = 1, \ldots, M$. We regard $f_1, \ldots, f_M$ as fitness values in our genetic algorithm. Sometimes the minimum and maximum fitness values are identical. In that case, the algorithm is terminated. Termination also occurs if a chromosome is found which yields a sequence with number of late jobs equal to zero.

The *reproduction* operator is an artificial version of natural selection. Chromosomes with a larger fitness value have a higher probability of surviving to be 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 $F_1, \ldots, F_M$. The aim of scaling is to prevent premature convergence and, in the latter stages of the algorithm, to avoid random walks amongst mediocre solutions. We use linear scaling, which is 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 the minimum scaled fitness is zero. Thus,

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

We also test versions of our genetic algorithm that do not use scaling, where $a = 1$ and $b = 0$ in (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 $F_k / \sum_{h=1}^{M} F_h$ and the mating pool is of size M, the expected number of copies in the mating pool is $e_k = MF_k / \sum_{h=1}^{M} F_h$. Firstly, $\lfloor e_k \rfloor$ copies of each chromosome k are placed in the mating pool, and then the population is sorted in non-increasing 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.

In *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. After each crossover, the two old chromosomes are discarded.

The *mutation* operator prevents loss of diversity in the population by randomly altering elements in the chromosomes. The number of elements that are changed depends upon a mutation probability. Firstly, we compute the expected total number of elements to be mutated (total number of elements in the population multiplied by the mutation probability). This expected number of elements is selected at random in the population and each is mutated by setting 0 to 1 and vice versa.

7 Computational results

In this section, we report on computational experience with the different local search heuristics. For each method (descent, simulated annealing, tabu search and genetic algorithm), we present tables which show how the choice of neighbourhood and other parameters affect solution quality. Although many additional parameter setting tests were performed to obtain a 'good' implementation of each algorithm, only the most significant are reported. Having found suitable parameter settings for each method, we then present results that compare the different methods.

The heuristics were tested by coding them in ANSI-C and running them on a HP 9000/825. One hundred random problems were generated for different combinations of numbers of jobs and families: the values used are $N = 30, 40, 50$ and $F = 4, 6, 8, 10$. The jobs are distributed uniformly across families, subject to the constraint that a total of N jobs is required. The set-up and job processing times are uniformly distributed integers between 1 and 10. Due dates were generated based on different values for the relative due date range (RDD $\in \{0.2, 0.4, 0.6, 0.8, 1.0\}$) and the average tardiness factor (TF $\in \{0.2, 0.4, 0.6, 0.8, 1.0\}$). The due dates are uniformly distributed integers from the interval $[P(1 - TF - RDD/2), P(1 - TF + RDD/2)]$, where $P = \sum_{f=1}^{F} n_f s_f / 2 + \sum_{i=1}^{N} p_i$. Four problems were generated for each pair of RDD and TF values.

The local search algorithms that are described in the previous sections were applied to all test problems. Also,

using a branch and bound algorithm that is currently under development, the optimal solutions of all 30-job and some 40-job test problems are obtained. For problems that are not solved to optimality, the best known solution that is obtained from the branch and bound algorithm or from any of the local search methods is recorded. For each test set of 100 problems and each local search algorithm, the number of times the best known solution is found (NO) and the number of times the heuristic solution value deviates two or more jobs from the best known solution (D2) are tabulated. Generally, we use NO as our performance indicator when comparing the quality of solutions obtained with different algorithms. The other column gives the average computation time in seconds (ACT) on a HP 9000/825 computer.

The following abbreviations are used.

DN: the descent heuristic, where N is replaced by J or B depending on whether the job or batch neighbourhood is used.

SAN: simulated annealing, where N is replaced by J or B depending on whether the job or batch neighbourhood is used; the initial acceptance probability is $K_1 = 0.50$ and the final acceptance probability is $K_l = 0.0001$.

TSNL: tabu search, where N is replaced by J or B depending on whether the job or batch neighbourhood is used; and where L is replaced by E if the tabu list stores the complete early set (with list length equal to half the number of iterations), and L is replaced by J if the tabu list stores jobs which cannot be made late (with list length equal to 7).

GAXY: genetic algorithm, where X can be empty or replaced by S, indicating that linear scaling with a parameter $C = 1.9$ is applied; and where Y can be empty, or replaced by I indicating an improvement procedure (Step 4 of Algorithm A) is applied to all solutions of the final population, or replaced by L indicating that an additional N generations are created. A population size of $2N$ is used, and normally $2N$ generations are executed (except for the L version). The crossover operator is based on interchanging $N/5$ sections, each containing $N/10$ elements. The mutation probability is equal to 0.001.

For the first three algorithms, multi-start versions are tested. A number is appended between parentheses to indicate the number of trials from different initial solutions. Recall that for multi-start versions of the neighbourhood search algorithms, the first initial solution is constructed with Algorithm A, while the others are generated at random using Algorithm B.

To compare different versions of the same algorithm, results for $N = 50$ only are given, because for $N = 30$ and $N = 40$ differences in solution quality are less pronounced.

Descent

In Table 1, results are shown for the descent method with the two neighbourhood structures: job (DJ) and batch (DB). For the single-start versions DJ(1) and DB(1), 100 levels are performed. However, only 20 levels are executed for each initial solution in the multi-start versions DJ(5) and DB(5), so that approximately the same computation time is required as for the corresponding single-start algorithm.

Table 1. Results for descent													
N	F	DJ(1)			DB(1)			DJ(5)			DB(5)		
		NO	D2	ACT	NO	D2	ACT	NO	D2	ACT	NO	D2	ACT
50	4	58	15	6.9	61	16	2.6	72	3	7.5	66	8	3.1
	6	50	14	8.7	50	12	4.2	68	5	9.4	63	4	4.6
	8	49	20	10.4	50	11	5.2	64	0	10.9	63	3	5.8
	10	48	17	11.5	53	10	6.3	64	0	11.8	67	3	6.9

Results for the multi-start versions indicate their superiority over those with a single start. Although the batch neighbourhood requires less computation time because of size, its performance is worse. Increasing the number of levels in DB(5), so that about the same computation time is used as for DJ(5), does not yield superior quality solutions. For all versions, the required computation time is very sensitive to the number of families.

Simulated annealing

Table 2 presents the results for simulated annealing. For the single-start versions SAJ(1) and SAB(1), the number of levels is 100; for SAJ(5) and SAB(5), there are 20 levels for each of the five trials.

Table 2. Results for simulated annealing													
N	F	SAJ(1)			SAB(1)			SAJ(5)			SAB(5)		
		NO	D2	ACT	NO	D2	ACT	NO	D2	ACT	NO	D2	ACT
50	4	74	2	7.3	70	7	3.1	78	0	7.8	73	6	3.5
	6	69	2	9.2	63	4	5.0	73	1	9.7	66	1	5.7
	8	73	2	10.4	65	5	6.3	76	0	11.1	66	5	6.9
	10	72	0	11.5	64	2	7.6	75	0	11.9	63	0	8.0

For both neighbourhood structures, the results are better than those obtained with single-start descent (DJ(1) and DB(1)), and slightly better than those obtained with multi-start descent (DJ(5) and DB(5)) in most cases. The improvements obtained with multi-start are not as large as those with the descent method: for each test set, 4 or 5 problems are solved better, whereas multi-start descent results in improvements for more than 20 problems in each set. As with descent, the larger job neighbourhood tends to perform better than the batch neighbourhood, but at the expense of more computation time.

Tabu search

Table 3 gives results for the different tabu search methods. With the job neighbourhood, 100 iterations (levels) are executed, and there are 200 iterations with the batch neighbourhood.

Table 3. Results for tabu search													
N	F	TSJE(1)			TSJJ(1)			TSBE(1)			TSBJ(1)		
		NO	D2	ACT	NO	D2	ACT	NO	D2	ACT	NO	D2	ACT
50	4	65	8	6.7	66	10	6.5	74	5	5.4	78	5	4.8
	6	58	12	8.5	63	14	8.2	73	1	8.6	70	2	8.1
	8	58	15	9.4	59	12	9.2	70	5	10.4	74	1	9.5
	10	55	6	10.9	56	6	10.5	71	0	12.7	73	0	11.8

Contrary to descent and simulated annealing, the job neighbourhood performs worse than the batch neighbourhood. Note that the larger number of iterations does not explain this phenomenon, since even after 100 iterations with the batch neighbourhood better quality solutions are observed than with the job neighbourhood. The reason for the better performance of the batch neighbourhood is probably that more diversification is incorporated in the search through the solution space. By moving a batch to the late set, the likelihood of the resulting neighbour exhibiting significantly different characteristics to the original solution is higher than if a single job is moved.

The difference between the two tabu list structures is not great. The versions with a list where jobs are forbidden to be made late (TSJJ(1) and TSBJ(1)) have a slightly better performance except for TSBJ(1) when $F = 6$, and they also require less computation time.

Genetic algorithm

In Table 4, results for different versions of the genetic algorithm are shown.

Table 4. Results for the genetic algorithm													
N	F	GA			GAS			GASI			GASL		
		NO	D2	ACT	NO	D2	ACT	NO	D2	ACT	NO	D2	ACT
50	4	31	59	7.3	71	0	6.7	83	0	7.1	72	0	10.0
	6	25	64	8.2	81	0	8.0	87	0	8.1	82	0	11.8
	8	25	61	8.3	78	0	8.2	85	0	8.5	80	0	12.1
	10	25	60	8.8	84	0	8.8	88	0	8.9	85	0	12.6

A comparison of the results for GA with the others, shows that linear scaling is essential in order to obtain good results. The improvement step in GASI requires less computation time than performing an additional N generations in GASL, and it can generate results of slightly better quality.

To obtain an idea of the benefits of batching, we have calculated the average number of batches in the solutions obtained by GASI. This average equals 9, 11, 13 and 14 for F =4, 6, 8 and 10, respectively. This shows that in many problem instances several families are split into two or more batches. For some problems with a large relative range of due dates and a small tardiness factor, an optimal solution with no late jobs is generated by GASI. Such a solution may have a lot of small batches: for example, one problem with $F = 10$ has 30 batches.

Comparison of different local search techniques

The results of the tests that are described above provide guidelines for the design of local search heuristics. The neighbourhood search methods clearly benefit from performing several short trials from different starting solutions, rather than using a single long run. Also, all methods gain from starting with good solutions, and the performance of the genetic algorithm is enhanced by adding an improvement step at the end. Furthermore, descent and simulated annealing seem to produce better quality solutions under the larger job neighbourhood, whereas tabu search favours the smaller batch neighbourhood. For tabu search, the batch neighbourhood provides better diversification, and the computational expense of searching the larger job neighbourhood for the best neighbour at each iteration is avoided.

These findings from our preliminary tests were incorporated in 'good' implementations of descent, simulated annealing, tabu search and a genetic algorithm. The first three methods are multi-start versions: the number of trials is equal to $N/3$ for descent, and $N/10$ for simulated annealing and tabu search. Both descent and simulated annealing use the job neighbourhood: $N/10$ levels are executed during each trial of descent, and for simulated annealing there are $N/2$ levels per trial. The batch neighbourhood is used in tabu search which performs N iterations (levels) during each trial. For the genetic algorithm GASI, the same parameters as for Table 4 are used. Table 5 provides comparative computational results for these implementations.

Table 5. Comparative results for different local search methods

N	F	DJ($N/3$)			SAJ($N/10$)			TSBJ($N/10$)			GASI		
		NO	D2	ACT	NO	D2	ACT	NO	D2	ACT	NO	D2	ACT
30	4	86	0	0.9	85	0	1.1	86	0	1.0	85	0	1.8
	6	88	0	1.0	87	0	1.3	87	0	1.4	88	0	1.9
	8	78	0	1.1	84	0	1.4	84	0	1.6	83	0	2.0
	10	84	0	1.2	83	0	1.5	90	0	1.9	91	0	2.1
40	4	83	1	2.7	79	0	3.6	78	3	2.8	86	0	3.8
	6	81	0	3.2	80	0	4.3	78	1	4.1	93	0	4.2
	8	82	0	3.7	82	0	4.9	87	0	5.1	88	0	4.5
	10	77	0	3.9	75	0	5.3	87	0	5.9	89	0	4.6
50	4	79	1	6.7	78	0	9.6	77	4	6.7	83	0	7.1
	6	75	1	8.5	73	0	12.1	74	2	10.0	87	0	8.1
	8	75	0	9.3	73	0	13.5	77	1	12.4	85	0	8.4
	10	69	0	10.2	76	0	14.4	75	0	14.8	88	0	8.9

Although the previous tables indicated that the more sophisticated local search heuristics tend to improve on the performance of the classical multi-start descent method, this is not so clear-cut for simulated annealing and tabu search in Table 5. The better performance of DJ($N/3$) compared to DJ(5) of Table 1 for $N = 50$ is a consequence of performing a lot of short trials instead of a few trials with more iteration levels. Note that the use of short trials also avoids wasting computation time should cycling occur.

The small differences between SAJ($N/10$) and SAJ(5) in Table 2 show the stochastic character of simulated annealing. Although fewer levels are used (20 instead of 25 for $N = 50$) to obtain the results of Table 2, the performance is sometimes better with SAJ(5), as can be observed for $F = 8$.

By comparing the results of TSBJ(1) in Table 3 with those of TSBJ($N/10$), we observe that the effect of multiple starts is not as large as for descent and simulated annealing. For $N = 50$ and $F = 4$, the number of times the best known solution is found (NO=77) for TSBJ($N/10$) is one less than for TSBJ(1), even though more computation time is required.

The best quality solutions are obtained with the genetic algorithm, except for some of the 30-job problems

where the other methods sometimes generate more optimal solutions at smaller computational expense. As Table 4 shows, this good performance is partly attributed to the improvement step which is applied to all solutions of the final population.

From the D2 columns in Table 5, we observe that simulated annealing always finds a solution with a deviation of no more than one job from the best known solution. This is also the case for the genetic algorithm. The other methods sometimes generate a solution with a larger deviation, especially when the number of families is small ($F = 4$ and $F = 6$).

The computation time required by tabu search is very sensitive to the number of families. This is because the batch neighbourhood becomes larger as the number of families increases. The computation time also increases with the number of families for descent and simulated annealing; for the genetic algorithm, it is more stable.

8 Concluding remarks

This paper considers a single machine scheduling problem with multiple families, where a decision not to process jobs of a family together results in additional set-up time. The objective is to minimize the number of late jobs. Apart from the complexity results of Monma and Potts [6], we are not aware of any studies that tackle this problem. Various local search heuristics are studied. Our requirement is for these heuristics to have modest computational requirements so that they can be used for on-line scheduling. To achieve a satisfactory performance with limited computational resources, the methods should exploit known structural properties of the problem. With this proviso, we adopt 'standard' implementations unless initial experiments indicate that the quality of the resulting solutions is unacceptable.

For neighbourhood search algorithms, we represent solutions by sequences of early jobs. Two neighbourhood structures are proposed, and they are used in descent, simulated annealing and tabu search algorithms. We also develop a genetic algorithm which adopts a simple binary encoding of the solution.

Extensive computational tests show that all of the local search heuristics generate solutions of high quality (the best known solution value is obtained for over 75% of our test problems). This success is attributed to structural knowledge of the problem that is incorporated into the heuristics. For the neighbourhood search methods, our results show that multi-start versions are preferred to those in which a single long run is performed. Also, the larger job neighbourhood (based on setting one early job to be late) gives good results for descent and simulated annealing, whereas the smaller batch neighbourhood (based on setting one early batch to be late) is superior in tabu search.

Our genetic algorithm (GASI) generates the best quality solutions. The best known solution is obtained for more than 80% of the problems in each test set, and the maximum deviation from this best known solution is not greater than one job for any test problem. Also, the genetic algorithm requires less computation time than the other methods for large problem sizes.

Our conclusions on the relative merits of different local search algorithms apply to the particular representations of solutions and neighbourhood structures that we have adopted. An interesting future research topic would involve experimentation with different representations of solutions.

References

[1] Bruno, J., and Downey, P., "Complexity of task sequencing with deadlines, set-up times and changeover costs", *SIAM Journal on Computing* 7 (1978) 393–404.

[2] Eglese, R.W., "Simulated annealing: a tool for operational research", *European Journal of Operational Research* 46 (1990) 271–281.

[3] Glover, F., "Tabu search, Part I", *ORSA Journal on Computing* 1 (1989) 190–206.

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

[5] Kirkpatrick, A., Gelatt Jr., C.D., and Vecchi, M.P., "Optimization by simulated annealing", *Science* 220 (1983) 671–680.

[6] Monma, C.L., and Potts, C.N., "On the complexity of scheduling with batch setup times", *Operations Research* 37 (1989) 798–804.

[7] Moore, J.M., "An n job, one machine sequencing algorithm for minimizing the number of late jobs", *Management Science* 15 (1968) 102–109.

[8] Pirlot, M., "General local search heuristics in combinatorial optimization: a tutorial", *Belgian Journal of Operations Research, Statistics and Computer Science*, 32 (1992) 8–67.

[9] Potts, C.N., and Van Wassenhove, L.N., "Single machine tardiness sequencing heuristics", *IIE Transactions* 23 (1991) 346–354.

[10] Potts, C.N., and Van Wassenhove, L.N., "Integrating scheduling with batching and lot-sizing: a review of algorithms and complexity", *Journal of the Operational Research Society* 43 (1992) 395–406.