"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. VAN WASSENHOVE ***

94/45/TM

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

** Faculty of Mathematical Studies, University of Southampton, Southampton S09 5NH, UK.

*** Professor of Operations Management and Operations Research at INSEAD, Boulevard de Constance, 77305 Fontainebleau 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 SO9 5NH, 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 developed: 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 by the genetic algorithm; multi-start descent also performs quite well.

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 where jobs have to be sequenced for processing on a multi-tool machine. Whenever a change is made 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 different batching problems is given by Potts and Van Wassenhove [10].

In this paper we consider a single machine scheduling problem with sequence independent set-up times. We assume that all the jobs are available at time zero, and that each job belongs to a family, has a given processing time and a due date. The objective is to find a schedule which minimizes the number of late jobs.

Monma and Potts [6] consider a variety of single machine family scheduling problems under the assumption that sequence dependent family set-up times 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 the problem using branch and bound 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 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 problem of scheduling with family set-up times 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. 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 the i'th job in family f, and d_{if} its due date.

A sequence independent set-up time $s_f \geq 0$ 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 incurred if a job from family f is the first to be processed.

It is convenient to regard a sequence $S = (E, L)$ as a partial sequence E of early jobs 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 B_k is the k'th batch in S then

$$S = (B_1, B_2, ..., B_r, L).$$

Each batch B_k can be viewed as a single composite job with processing time T_k and due date D_k. If batch B_k contains jobs $\pi(u), \pi(u+1), \ldots, \pi(v)$ from family f, then T_k and D_k are calculated as follows:

$$T_k = s_f + \sum_{i=u}^{v} p_{\pi(i)f} \qquad \text{and} \qquad D_k = \min_{j \in \{u, u+1, \ldots, v\}} \{d_{\pi(j)f} + \sum_{h=j+1}^{v} p_{\pi(h)f}\}.$$

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

Proof. Let C_k be the completion time of the on-time batch containing $\pi(u), \pi(u+1), \ldots, \pi(v)$ of family f: $C_k \leq D_k$. We show for the case $C_k = D_k$ that all composing jobs $(\pi(u), \ldots, \pi(v))$ are on time.

For any job $\pi(\nu)$ of B_k, it follows from the definition of D_k that

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

Job $\pi(\nu)$ is on time because $C_{\pi(\nu)f} = C_k - \sum_{h=\nu+1}^{v} p_{\pi(h)f} \leq D_k - \sum_{h=\nu+1}^{v} p_{\pi(h)f} \leq d_{\pi(\nu)f}$. □

We can now state the following theorems.

Theorem 1 (Monma and Potts [6]) *There exists an optimal schedule where the on-time jobs within each family are ordered by the earliest due date rule.*
That is, for an optimal sequence with this property, if the i'th early job in family f precedes the j'th early job in f, then $d_{if} \leq d_{jf}$ (intragroup EDD property).

Theorem 2 *There exists an optimal schedule where the on-time batches are ordered by the earliest due date rule where the due dates of the composite jobs are considered.*
That is, if batch B_k precedes batch B_l in some optimal sequence, then $D_k \leq D_l$ (intergroup EDD property).

Proof. Lemma 1 establishes the fact that all jobs of a batch are on time it the batch itself is on time.
Moreover, a straightforward adjacent interchange argument for batches verifies the existence of an optimal sequence in which on-time 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 $1 \leq f \leq F$ and the families have been renumbered so that $D_1 \leq \ldots \leq D_F$.

Using the above 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 on-time 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.

Consider the sequence

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

where $\sigma(i,f)$ denotes the i'th early job from family f. The batch containing only one job $\sigma(i,f)$ is shifted to the position just before $\sigma(i+1,f)$ if it is still on time. This backward shift is attempted for all batches with one job. For the remaining batches with one job, a forward shift is tried, e.g. job $\sigma(j,f)$ is shifted to the position just after $\sigma(j-1,f)$ if all jobs starting from $\sigma(s,h)$ remain on time.

Reorder the batches so that the intergroup EDD property is satisfied. Add as many jobs as possible from the late set (see Algorithm *Insert* below).

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

$$B_k = (\sigma(i,f), \ldots, \sigma(j-1,f), \sigma(j,f), \ldots, \sigma(k,f))$$

A split is performed between $\sigma(j-1,f)$ and $\sigma(j,f)$ $(j = i+1, \ldots, k)$, when $d_{\sigma(j-1,f)} + p_{\sigma(j,f)} < d_{\sigma(j,f)}$ and $C_{\sigma(j,f)} + s_f < d_{\sigma(j,f)}$. The subbatch $(\sigma(j,f), \ldots, \sigma(k,f))$ is shifted at least after B_{k+1}, possibly even further corresponding to the EDD intergroup property. If it cannot be shifted after B_{k+1}, the split is disregarded. The subbatch $(\sigma(i,f), \ldots, \sigma(j-1,f))$ is possibly shifted forwards before B_{k-1}, again satisfying the EDD intergroup property.

Try to add as many jobs as possible from the late set (see Algorithm *Insert* below). If no improvement can be made, undo the additional split.

Step 4. Search for a batch containing just one job with the longest set-up plus processing time, make it late and repeat Step 3.

An initial solution can also be generated at random by chosing arbitrarily which jobs are early and how the batches are constructed.

Algorithm B

Step 1. For each job $i = 1, \ldots, N$ generate two random numbers r_{i1} and r_{i2}. If $r_{i1} \leq 0.5$ job i is considered as early; if $r_{i2} \leq 0.5$ the family is split after job i. In this way a number of batches of early jobs is constructed.

Step 2. Compute for each batch of early jobs the processing time T_k and due date D_k. Apply the algorithm of Moore [7] to minimize the number of late batches. From this number of late batches, the number of late jobs can be derived.

Step 3. Improve the resulting sequence with Algorithm Insert.

The algorithm 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. Let $E = (\pi(1), \ldots, \pi(n_e))$, where n_e is the number of jobs in the early set E. 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 $\pi(i)$ and $\pi(i-1)$ belong to the same family and $i > 1$; $p'_{\pi(i)} = p_{\pi(i)} + s_f$ otherwise, where f the family to which $\pi(i)$ belongs.

Step 3. Consider a job j from the late set, belonging to family f, which satisfies the intragroup EDD property when inserted immediately before $\pi(e)$. Calculate $t' = t + p'_j$ with $p'_j = p_j$ or $p'_j = p_j + s_f$ depending on whether job j and job $\pi(e)$ belong to the same family or not. If $t' \leq v_{\pi(e)}$ and $t' \leq d_j$ insert job j before job $\pi(e)$ except when the job can be inserted in a later position just before a batch with jobs of the same family f. Set $n_e = n_e + 1$ and $\pi(i) = \pi(i-1)$ for $i = n_e, n_e - 1, \ldots, e+1$; $\pi(e) = j$ 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 $t = t + p'_{\pi(e)}$, $e = e + 1$. If $e \leq n_e$ go to Step 3. For all remaining jobs in L, try to add them at the end of the sequence.

3 Neighbourhood Search

A neighbourhood search method requires an initial solution, which can be constructed by a heuristic (Algorithm A) or it can be chosen at random (Algorithm B). 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.

For the considered problem a neighbourhood can be constructed in various ways by taking a subbatch out of the sequence of early jobs and replacing it by jobs from the late set. For this replacement we use Algorithm Insert described in the previous section.

Starting from a sequence $(\pi(1), \ldots, \pi(k), \pi(k+1), \ldots, \pi(l), \pi(l+1), \ldots, \pi(n_e), L)$, where $\pi(k+1), \ldots, \pi(l)$ are jobs from the same family and L is the set of late jobs, the subbatch that is removed can take different forms. We consider two forms.

In the *job* neighbourhood, just one job $\pi(i)$ with $i = 1, \ldots, n_e$, is removed. The search is done systematically by starting with the removal of the job in the first position of the sequence. After Algorithm Insert, the resulting sequence is a neighbour. Then the job in the second position is removed, and so on, until the job in position n_e is removed.

In the *batch* neighbourhood each time a complete batch $(\pi(k+1),\ldots,\pi(l))$ is removed, starting from the first batch in the sequence and systematically proceeding to the last batch. The number of neighbours is thus smaller than for the job neighbourhood. Because a batch can contain many jobs, it is possible that the number of late jobs is smaller than the number of jobs just removed. Therefore, Algorithm Insert is first applied for inserting from the late set, and then again for reinsertion of the jobs from the removed batch except for the first job (i.e. for jobs $\pi(k+2),\ldots,\pi(l)$).

In both cases, we refer to the complete search of the neighbourhood as a *level*, because there is a relation with our implementation of simulated annealing (see following section). During such a level, the number of moves that is considered, is equal to the number of subsequences (jobs or batches) that is swapped to the late set.

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, the pure procedure stops. For this problem however, neighbours frequently have the same objective function value as the starting sequence. These neighbours are also accepted in our procedure. The stopping criterion is defined by a fixed number of levels, i.e. the number of times the complete neighbourhood is searched. Although the resulting solution is 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 to take the best sequence as final solution. Such an approach is called *multi-start descent.*

Another possibility is to allow interchanges which lead to an increase in the objective function value. Consequently, the procedure can escape from a local minimum and continue its search for a global minimum. 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 metaheuristic. 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. The parameter t is known as the *temperature.*

In our implementation of simulated annealing, the temperature is derived from an acceptance probability K as described by Potts and Van Wassenhove [9]. More precisely, K is equal to the probability of accepting an increase of one job. If l denotes the number of distinct acceptance probability levels considered, we set $K_{i+1} = K_i - (K_0 - K_f)/(l-1)$, where K_0 and K_f are the initial and final acceptance probabilities. In order to compute the actual acceptance probability for any given increase in the

objective function value, the temperature t is found using $K_i = \exp(-1/t)$, thereby yielding $t = -1/\ln K_i$.

For each acceptance probability level, the complete neighbourhood is searched systematically as described in the previous section. We have performed 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 non-improving moves. However, certain moves are forbidden or *tabu* for a few iterations to prevent a closed loop of movements between a few sequences.

Conventionally, the complete neighbourhood must be searched in each iteration to find the best possible move. During such a iteration each job or batch of the early sequence is swapped to the late set, followed by an application of the Algorithm Insert. Thus, it appears advantageous to use a small neighbourhood. So it is appropriate to use the batch neighbourhood, but we have also undertaken tests with the job neighbourhood. The search is also stopped as soon as a non-tabu neighbour is found which improves upon the current objective function value.

A tabu list is constructed, the purpose of which is to prevent the search heuristic from returning to a previously visited local minimum. There are several ways to characterize a tabu move. In the first version of our tabu search algorithm, the complete list of early jobs is stored on the tabu list. If a subsequent iteration results in the same set of early jobs, the sequence is not accepted. In the second version of our algorithm, the jobs that are made early during an iteration are added to the tabu list. This means that during subsequent iterations these jobs cannot be made late.

The length of the tabu list determines the number of iterations during which a move remains tabu. Because of the more specific characteristics of a tabu move in the first version, a longer tabu list is necessary to avoid cycling. For the second method, the classical tabu list size 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].

Our genetic algorithm uses an encoding scheme in which the chromosome no longer directly represents the solution sequence. Because the order of the early jobs in a family is known, as well as the order of the batches, it is sufficient to indicate which jobs are early and how these jobs are composed into batches. This can be achieved by binary encoding, in which there are two elements for each job. These two elements are adjacent in the chromosome; a chromosome thus consists of N groups of two elements each. If the first element for a job is set to 1, then the job is considered early; if the second element is 1, then this job ends a batch. An arbitrary chromosome resulting from this encoding scheme does not necessarily correspond to a feasible sequence. To overcome the problem of feasibility, we adopt the following approach.

Out of this encoding the early jobs (the first element of each pair) are composed into batches corresponding to the second element of each job. For these batches, the processing time T_k and due date D_k are computed as described in Section 2. The resulting problem of minimizing the number of late batches is solved with the algorithm of Moore [7]. From the number of late batches, the number of late jobs is derived which 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 $\mathrm{TF} = 1 - d_{\mathbf{avg}}/P$, where $d_{\mathbf{avg}} = \sum_{i=1}^{N} d_i/N$ is the average due date and $P = (\sum_{f=1}^{F} s_f n_f)/2 + \sum_{j=1}^{N} p_j$ is an estimate for the total 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 $1 - \mathrm{TF}$ is used to decide if the early job element is set to 1. For the split elements a probability of $\mathrm{RDD} \times \mathrm{TF}$ is used. If the relative due date range, $\mathrm{RDD} = (\max_{i=1,\ldots,N} d_i - \min_{i=1,\ldots,N} d_i)/P$, 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 make a lot of small batches by setting more split elements to 1. This is because the evaluation of the chromosome fitness is done by minimizing the number of late batches.

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

Genetic algorithm

Step 1. Create an initial population and set it as the current population. Evaluate 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. All elements of the final population are taken as an initial sequence for an improvement procedure which 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. Next, $V_1, \ldots, V_M$ are mapped to fitness values $f_1, \ldots, f_M$. This mapping is necessary because a genetic algorithm always works with a maximization problem: we use $f_k = N - V_k$ for $k = 1, \ldots, M$, so that f_k is actually the number of early jobs. 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 $F_1, \ldots, F_M$. This scaling prevents premature convergence and, in the latter stages of the algorithm, avoids random walks amongst the mediocre solutions. We use linear scaling, which is defined by

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

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

$$\sum_{k=1}^{M} F_k = \sum_{k=1}^{M} f_k. \tag{2}$$

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

$$\max_{k=1,\ldots,M}\{F_k\} = C \sum_{k=1}^{M} F_k/M. \tag{3}$$

Because of (3), there is a possibility that this scaling could produce negative fitness values. If the constants a and b that are derived from (1), (2) and (3) yield a negative minimum fitness value, then (3) is replaced by an alternative relationship which specifies that the the minimum scaled fitness is zero. Thus,

$$\min_{k=1,\ldots,M}\{F_k\} = 0. \tag{4}$$

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

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 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 $P = (\sum_{f=1}^{F} s_f n_f)/2 + \sum_{j=1}^{N} p_j$. Four problems are generated for each pair of RDD and TF values. The jobs are distributed uniformly across families, subject to the constraint that a total of N jobs is required.

For each test set of 100 problems and each local search algorithm, the number of times the best known solution (NO) is found and the number of times the heuristic solution value deviates two or more jobs from the best known solution (D2) are tabulated. The other column (ACT) gives the average computation time in seconds on a HP9000/825.

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 $K_0 = 0.50$ and

the final acceptance probability $K_f = 0.0001$.

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 (the length of the tabu list is equal to half the number of iterations), and L is replaced by J if the tabu list (with length equal to 7) stores jobs which cannot be made late.

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) to all solutions of the ending population or replaced by L indicating $3N$ generations. A population size of $2N$ is used and normally $2N$ generations are done (except for the L version). The crossover operator is based on interchanging $N/5$ sections of $N/10$ bits each. The mutation probability is equal to 0.001.

For each algorithm, a number is appended between parentheses to indicate the number of trials starting from different initial solutions. When multiple trials of the neighbourhood search algorithms are used, the first initial solution is constructed with the heuristic described in Section 2 (Algorithm A), while the others are generated at random (Algorithm B).

In the first four tables, results for $N = 50$ only are given because for $N = 30$ and $N = 40$ differences between the different methods are less pronounced.

Descent heuristics.

In Table 1, the results are shown for the descent method with the two neighbourhood structures: job (DJ) and batch (DB). In both cases, $l = 100$ levels are performed. The last two columns gives results for multi-start descent with five different initial solutions. Here only $l = 20$ levels are executed for each initial solution so that approximately the same CPU time is required.

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 with the multi-start method indicate its superiority over the single start method. The batch neighbourhood requires less CPU time because of the smaller neighbourhood but its performance is worse. The required CPU time is also very sensitive to the number of families.

Simulated annealing.

Table 2 presents the results for simulated annealing. The first two columns correspond to a single trial of $l = 100$ levels for the two neighbourhood structures; the last two to five trials of $l = 20$ levels each.

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 the descent method (DJ(1) and DB(1)) and slightly better than those obtained with the multi-start descent method (DJ(5) and DB(5)) except with the batch neighbourhood for $F = 10$. The improvements obtained with multi-start are not as large as 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 the descent method, the larger job neighbourhood tends to perform better than the batch neighbourhood, but at the expense of more CPU time.

Tabu search.

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

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. The reason for this better performance of the batch neighbourhood is probably that more diversification is incorporated in the search through the solution space. By swapping a batch to the late set instead of just a job, it is more likely that the resulting neighbour is quite different from the original element.

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

Genetic algorithm.

In Table 4, the results 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

The first column (GA) compared with the others, shows that linear scaling is essential in order to obtain good results. The improvement step (GASI) requires less CPU time than doing more generations (GASL) and it can obtain approximately the same results.

Comparison of different local search techniques.

In Table 5, we compare the results obtained using 'good' versions of the descent, simulated annealing and tabu search methods, with those obtained with a genetic algorithm.

The first three methods all employ multiple trials from different starting solutions; 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; for descent $N/10$ levels are executed during each trial and for simulated annealing $N/2$ levels. The batch neighbourhood is used in tabu search which does N iterations during each trial. For the genetic algorithm the same parameters as for Table 4 are used.

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

The more sophisticated local search heuristics tend to improve on the performance of the classical multi-start descent method, in particular for $F = 10$.

The better performance of DJ($N/3$) compared to DJ(5) of Table 1 ($N = 50$) is a result of doing a lot of short trials starting from different random sequences instead of a few trials with more iteration levels.

The small differences between SAJ($N/10$) and SAJ(5) of Table 2 show the stochastic character of simulated annealing. Although fewer levels are used ($l = 20$ instead of 25) to obtain the results of Table 2, there is a better performance for the test sets with $F = 8$ and 10.

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

The best performance is obtained with the genetic algorithm, except for $N = 30$, $F = 4$ where DJ($N/3$) and TSBJ($N/10$) result in a better NO=86 and less computation time is required. As Table 4 shows, this good performance is realised by applying the improvement step to all the elements of the ending population.

The required CPU time of tabu search is very sensitive to the number of families. This is because of the batch neighbourhood where there are more elements in the neighbourhood when there are more families. The computation time also increases with the number of families for descent and simulated annealing; for the genetic algorithm it is more stable.

When we look at the maximum deviation (D2) we see 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 result in a solution with a larger deviation, especially when the number of families is small ($F = 4, 6$).

8 Concluding remarks

Local search methods are studied for 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. We have developed two neighbourhood structures and used them in descent, simulated annealing and tabu search algorithms. Another method we propose, is a genetic algorithm with a simple binary encoding of the solution.

Extensive computational tests show that the larger neighbourhood size (based on swapping one job) gives good results for simulated annealing (SAJ($N/10$)) and also for multi-start descent (DJ($N/3$)) when a lot of short trials are done. However for tabu search (TSJJ) a lot of computation time is wasted during each iteration by searching this large neighbourhood and then performing in a rather small change in the solution sequence. By using a smaller neighbourhood (swapping a batch) which also allows a broader search (TSBJ($N/10$)), we get a better performance comparable with multi-start descent and simulated annealing.

Our genetic algorithm (GASI) generates solutions of very good quality. More than 80 % of the problems in each test set are solved to the best known solution; the maximum

deviation from this best known solution is in all sets not greater than one job. Also it requires less computation time than the other methods for large problem sizes.

References

1. Bruno, J. and Downey, P., "Complexity of task sequencing with deadlines, set-up times and changeover costs", *SIAM J. Computing,* Vol.7, No.4, pp. 393-404 (1978).

2. Eglese, R.W., "Simulated annealing: a tool for operational research", *European Journal of Operations Research,* 46, pp. 271-281 (1990).

3. Glover, F., "Tabu search, Part I", *ORSA Journal on Computing,* Vol.1, No.3, pp. 190-206 (1989).

4. Goldberg, D.E., *Genetic Algorithms in Search, Optimization and Machine Learning,* Addison-Wesley, Reading, MA (1989). *Operations Research,* Vol.37, No.6, pp. 865-892 (1989).

5. Kirkpatrick, A., Getlatt Jr., C.D., Vechi, M.P., "Optimization by simulated annealing", *Science,* 220, pp. 671-680 (1983).

6. Monma, C.L. and Potts, C.N., "On the complexity of scheduling with batch setup times", *Operations Research,* 37, pp. 798-904 (1989).

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

8. Pirlot, M., "General local search heuristics in combinatorial optimization: a tutorial," *Belgian Journal of Operations Research, Statistics and Computer Science,* Vol. 32, No. 1-2, pp. 8-67 (1992).

9. Potts, C. N., and Van Wassenhove, L. N., "Single Machine Tardiness Sequencing Heuristics," *IIE Transactions,* Vol. 23, No. 4, pp. 346-354 (1991).

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,* Vol. 43, No. 5, pp. 395-406 (1992).