Job Shop Scheduling by Local Search

R.J.M. Vaessens [1]
E.H.L. Aarts [2,1]
J.K. Lenstra [1,3]

1. Eindhoven University of Technology, Department of Mathematics and Computing Science, P.O. Box 513, 5600 MB Eindhoven; Email: {robv, jkl}@win.tue.nl
2. Philips Research Laboratories, P.O. Box 80000, 5600 JA Eindhoven; Email: aarts@prl.philips.nl
3. CWI, P.O. Box 94079, 1090 GB Amsterdam

Abstract

We survey solution methods for the job shop scheduling problem with an emphasis on local search. Both deterministic and randomized local search methods as well as the proposed neighborhoods are discussed. We compare the computational performance of the various methods in terms of their effectiveness and efficiency on a standard set of problem instances.

Key words: job shop scheduling, local search, iterative improvement, shifting bottleneck heuristic, simulated annealing, taboo search, variable-depth search, genetic algorithms, constraint satisfaction.

In the job shop scheduling problem we are given a set of jobs and a set of machines. Each machine can handle at most one job at a time. Each job consists of a chain of operations, each of which needs to be processed during an uninterrupted time period of a given length on a given machine. The purpose is to find a schedule, that is, an allocation of the operations to time intervals on the machines, that has minimum length.

The problem is difficult to solve to optimality. For example, a relatively small instance with 10 jobs, 10 machines and 100 operations due to Fisher and Thompson[29] remained unsolved until 1986. Many solution methods have been proposed, ranging from simple and fast dispatching rules to sophisticated branch-and-bound algorithms.

We survey algorithms for the job shop scheduling problem with an emphasis on local search. During the last decade many different types of local search algorithms for job shop scheduling have been developed, and some of them have proved to be very effective.

The paper is structured as follows. In Section 1 several models for the job shop scheduling problem are presented. In Section 2 the complexity of the problem and methods for its solution are reviewed. Section 3 introduces local search and Section 4 discusses

representations and neighborhoods for the problem. Sections 5 and 6 describe constructive and iterative algorithms with local search, respectively; Section 7 describes some other techniques. Section 8 contains computational results, and Section 9 gives some concluding remarks.

1 The job shop scheduling problem

The job shop scheduling problem is formally defined as follows. Given are a set O of l operations, a set $\mathcal{M}$ of m machines, and a set $\mathcal{J}$ of n jobs. For each operation $v \in O$ there is a processing time $p(v) \in \mathbb{N}$, a unique machine $M(v) \in \mathcal{M}$ on which it requires processing, and a unique job $J(v) \in \mathcal{J}$ to which it belongs. On O a binary relation A is defined, which represents *precedences* between operations: if $(v, w) \in A$, then v has to be performed before w. A induces a total ordering of the operations belonging to the same job; no precedences exist between operations of different jobs. Furthermore, if $(v, w) \in A$ and there is no $u \in O$ with $(v, u) \in A$ and $(u, w) \in A$, then $M(v) \neq M(w)$.

A schedule is a function $S : O \rightarrow \mathbb{N} \cup \{0\}$ that for each operation v defines a start time $S(v)$. A schedule S is *feasible* if

$$\begin{array}{ll} \forall v \in O: & S(v) \geq 0, \\ \forall v, w \in O, (v, w) \in A: & S(v) + p(v) \leq S(w), \\ \forall v, w \in O, v \neq w, M(v) = M(w): & S(v) + p(v) \leq S(w) \text{ or } S(w) + p(w) \leq S(v). \end{array}$$

The *length* of a schedule S is $\max_{v \in O} S(v) + p(v)$, i.e., the earliest time at which all operations are completed. The problem is to find an *optimal* schedule, i.e., a feasible schedule of minimum length.

A feasible schedule is *left-justified* if no operation can start earlier without changing the processing order on any machine. It is *active* if no operation can start earlier without delaying another operation. Note that at least one optimal schedule is active and that each active schedule is left-justified.

An instance of the problem can be represented by means of a *disjunctive graph* $G = (O, A, E)$[59]. The vertices in O represent the operations, the arcs in A represent the given precedences between the operations, and the edges in $E = \{\{v, w\} \mid v, w \in O, v \neq w, M(v) = M(w)\}$ represent the machine capacity constraints. Each vertex $v \in O$ has a weight, equal to the processing time $p(v)$.

For each $E' \subseteq E$, an *orientation* on E' is a function $\Omega : E' \rightarrow O \times O$ such that $\Omega(\{v, w\}) \in \{(v, w), (w, v)\}$ for each $\{v, w\} \in E'$; we write $\Omega(E') = \{\Omega(e) \mid e \in E'\}$. A *partial orientation* is an orientation on $E' \neq E$ and a *complete orientation* is one on E. An orientation Ω on E' is *feasible* if the digraph $(O, A \cup \Omega(E'))$ is acyclic. It represents for each machine its *machine ordering*, i.e., the order in which it processes its operations. Each feasible schedule S uniquely determines a feasible complete orientation, which is denoted by Ω_S.

Conversely, for each feasible complete orientation Ω, there is a unique left-justified feasible schedule, which is denoted by S_Ω. For all $v \in O$, $S_\Omega(v)$ equals the length of a

longest path in the digraph $(O, A \cup \Omega(E))$ up to and excluding v. The length of S_Ω equals the length of a longest path in the digraph. Finding an optimal left-justified schedule is now equivalent to finding a feasible complete orientation that minimizes the longest path length in the corresponding digraph.

The search for such a schedule can be restricted to the set of active schedules. However, given a feasible complete orientation Ω, it is not clear at first sight whether S_Ω is active: activeness depends on the sizes of the processing times, while left-justifiedness does not. For this reason one often considers the larger set of left-justified schedules.

2 Complexity and algorithms

We give a brief review of results on the computational complexity of job shop scheduling, of the lower bounds and enumeration schemes that are used in branch-and-bound methods, and of the approximative approaches that yield upper bounds on the optimum. Techniques of the latter type that proceed by local search are discussed in the rest of the paper.

2.1 Complexity

Some very special cases of the problem can be solved in polynomial time, but their immediate generalizations are *NP*-hard. The results are summarized in Table 1, where l_j denotes the number of operations of the jth job. Note that, due to result (4b), it is *NP*-hard to find a job shop schedule that is shorter than 5/4 times the optimum.

Table 1: The complexity of job shop scheduling

solvable in polynomial time	*NP*-hard (in the strong sense*)
(1a) $m = 2$, all $l_j \leq 2$	(1b) $m = 2$, all $l_j \leq 3$; $m = 3$, all $l_j \leq 2$
(2a) $m = 2$, all $p(v) = 1$	(2b)* $m = 2$, all $p(v) \leq 2$; $m = 3$, all $p(v) = 1$
(3a) $n = 2$	(3b) $n = 3$
(4a) length ≤ 3	(4b)* length ≤ 4

(1a) Jackson[41]; (1b) Lenstra, Rinnooy Kan and Brucker[49]; (2a) Hefetz and Adiri[38]; (2b) Lenstra and Rinnooy Kan[48]; (3a) Akers[3] and Brucker[12]; (3b) Sotskov[63]; (4a,b) Williamson, Hall, Hoogeveen, Hurkens, Lenstra, Sevast'janov and Shmoys[68].

2.2 Lower bounds

Optimization algorithms for the problem employ some form of tree search. A node in the tree is usually characterized by a partial orientation Ω on a subset $E' \subset E$. The question is then how to compute a lower bound on the length of any feasible schedule corresponding to a completion of Ω.

Németi[54] and many subsequent authors obtained a lower bound by simply disregarding $E \setminus E'$ and computing the longest path length in the digraph $(O, A \cup \Omega(E'))$.

Bratley, Florian and Robillard[11] obtained the stronger *single-machine bound* by relaxing the capacity constraints of all machines except one. Given a machine M', they propose to solve the job shop scheduling problem on the disjunctive graph $(O, A \cup \Omega(E'), \{\{v, w\} \mid M(v) = M(w) = M'\} \setminus E')$. This is a single-machine problem, where the arcs in $A \cup \Omega(E')$ define release and delivery times for the operations on M' and precedence constraints between them. Lageweg, Lenstra and Rinnooy Kan[46] pointed out that many other lower bounds appear as special cases of this bound. For example, relaxing the capacity constraint of M' gives Németi's bound, and allowing preemption gives the bound used in current branch-and-bound codes. The bound itself is *NP*-hard to compute but can be found fairly efficiently[5,52,45,14]. It has been strengthened by Carlier and Pinson[16], who compute larger release and delivery times, and by Tiozzo[66] and Dauzère-Pérès and Lasserre[20], who observe that the arcs also define delays between precedence-related operations; Balas, Lenstra and Vazacopoulos[8] develop an algorithm for computing the bound subject to these delayed precedences.

Fisher, Lageweg, Lenstra and Rinnooy Kan[28] investigated surrogate duality relaxations, in which either the machine capacity constraints or the precedence constraints among the operations of a job are weighted and aggregated into a single constraint. Balas[7] described a first attempt to obtain bounds by polyhedral techniques. Applegate and Cook[4] review the valid inequalities studied before and gave some new ones. The computational performance of surrogate duality and polyhedral bounds reported until now is disappointing in view of what has been achieved for other hard problems.

2.3 Enumeration schemes

The traditional enumeration scheme is due to Giffler and Thompson[32]. It generates all active schedules by constructing them from front to back. At each node a machine on which the earliest possible completion time of any unscheduled operation is achieved is determined, and all unscheduled operations that can start earlier than this point in time on that machine are selected in turn.

Recent branch-and-bound algorithms use more flexible enumeration schemes. Carlier and Pinson[15,16,17] and Applegate and Cook[4] branch by selecting a single edge and orienting it in either of two ways. Brucker, Jurisch and Sievers[13] follow Grabowski's 'block approach'. All these authors apply the preemptive single-machine bound and a host of elimination rules. For details we refer to the literature.

The celebrated 10×10 instance of Fisher and Thompson[29] is within easy reach of these methods, but 15×15 instances seem to be the current limit. The main deficiency of the existing optimization algorithms for job shop scheduling is the weakness of the lower bounds. The situation is much brighter with respect to finding good upper bounds.

2.4 Upper bounds

Upper bounds on the optimum are usually obtained by generating a schedule and computing its length. An obvious first step is to apply a dispatch rule and to schedule the operations according to some priority function. Haupt[37] surveys such rules. They tend to exhibit an erratic behavior; the procedure 'bidir' proposed by Dell'Amico and Trubian[23] is one of the safer alternatives. The next step is then to try to improve the schedule by some sort of local search.

An entirely different approach is taken by Sevast'janov[61]. Using Steinitz' vector sum theorem, he develops polynomial-time algorithms for finding an upper bound with an absolute error that is independent of the number of jobs. Shmoys, Stein and Wein[62] improve on his results.

3 Local search

Local search employs the idea that a given solution may be improved by making small changes. Solutions are changed over and again, and better and better solutions are found.

We need the following notions. There is a set F of *feasible solutions*. Two functions are defined on F. The *cost function* is a mapping $c : F \rightarrow \mathbb{R}$, which in most cases is closely related to the function that is to be optimized. The *neighborhood function* is a mapping $N : F \rightarrow 2^F$, which defines for each solution $x \in F$ a *neighborhood* $N(x) \subseteq F$. Each solution in $N(x)$ is called a neighbor of x. Roughly speaking, the execution of a local search algorithm defines a walk in F such that each solution visited is a neighbor of the previously visited one.

A solution $x \in F$ is called a *local minimum* with respect to a neighborhood function N if $c(x) \leq c(y)$ for all $y \in N(x)$. The basic algorithm to find a local minimum is called *iterative improvement*. Starting at some initial feasible solution, its neighborhood is searched for a solution of lower cost. If such a solution is found, the algorithm is continued from there; otherwise, a local minimum has been found.

The quality of the local minimum depends on the initial solution, on the neighborhood function, and on the method of searching the neighborhoods. An initial solution may be obtained by generating it randomly or by applying a heuristic rule. The choice of a good neighborhood is often difficult. There is a clear trade-off between small and large neighborhoods: if the number of neighbors is larger, the probability of finding a good neighbor may be higher, but looking for it takes more time. There are several alternatives for searching the neighborhood: one may take the first neighbor found of lower cost (*first improvement*), or take the best neighbor in the entire neighborhood (*best improvement*), or take the best of a sample of neighbors, provided it is an improving one.

Often, one problem remains. The local optima obtained may be of poor quality. Therefore, several variants of iterative improvement have been proposed. The main variants can be divided into threshold algorithms, taboo search algorithms, variable-depth search algorithms, and genetic algorithms.

In *threshold algorithms*, a neighbor of a given solution becomes the new current solution if the cost difference between the current solution and its neighbor is below a certain threshold. One distinguishes three kinds of threshold algorithms.

In classical *iterative improvement* the thresholds are 0, so that only true improvements are accepted. In *threshold accepting*[26] the thresholds are nonnegative. They are large in the beginning of the algorithm's execution and gradually decrease to become 0 in the end. General rules to determine appropriate thresholds are lacking. In *simulated annealing*[44,18] the thresholds are positive and stochastic. Their values equal $-T \ln u$, where T is a control parameter (often called 'temperature'), whose value gradually decreases in the course of the algorithm's execution according to a 'cooling schedule', and u is drawn from a uniform distribution on (0,1]. Each time a neighbor is compared with the current solution, u is drawn again. Under certain mild conditions simulated annealing is guaranteed to find an optimal solution asymptotically.

In *taboo search*[33,34,35] one selects from a subset of *permissible neighbors* of the current solution a solution of minimum cost. In basic taboo search a neighbor is permissible if it is not on the 'taboo list' or satisfies a certain 'aspiration criterion'. The taboo list is recalculated at each iteration. It is often implicitly defined in terms of forbidden moves from the current solution to a neighbor. The aspiration criterion expresses possibilities to overrule the taboo status of a neighbor.

In *variable-depth search*[50] one starts from an initial solution and generates a sequence of subsequent neighbors by making relatively small changes. From this sequence one solution is selected to serve as the initial solution for a next sequence. Often, each time the sequence is extended, the set from which a neighbor is to be chosen is restricted by a sort of taboo list; this list may again be defined implicitly in terms of forbidden moves. Each time a new sequence is started, the list is emptied.

Genetic algorithms[39] are based on an extended notion of neighborhood function. A *hyperneighborhood function* is a mapping $N_{\mathrm{h}} : F^s \rightarrow 2^F$ with $s \geq 2$, which defines for each s-tuple $\mathbf{x} = (x_1, \ldots, x_s) \in F^s$ a set $N_{\mathrm{h}}(\mathbf{x}) \subseteq F$ of neighboring solutions. Each solution in $N_{\mathrm{h}}(\mathbf{x})$ is called a hyperneighbor of $\mathbf{x}$. At each iteration a set of solutions, often called 'population', is given. From this population several subsets of size s consisting of 'parents' are selected, and for each such subset some hyperneighbors, called 'offspring', are determined by operations called 'recombination' and 'mutation'. This set of hyperneighbors and the current population are then combined and reduced to a new population by selecting a subset of solutions.

Various solution methods combine local search with constructive, enumerative or iterative techniques. Such hybrid algorithms for job shop scheduling are dealt with below. We will encounter constructive rules that apply local search to partial solutions, combinations of local search with partial enumeration or backtracking, and nested forms of local search. In the latter case, local search is applied at several levels with different neighborhoods, so that the search can explore different regions of the solution space.

4 Solution representations and neighborhood functions

A crucial ingredient of a local search algorithm is the definition of a neighborhood function in combination with a solution representation. Below, several basic representations and neighborhood functions are introduced for the job shop scheduling problem. For most threshold and taboo search algorithms, only left-justified or active schedules are represented. This is done by specifying the start times of the operations or, equivalently, the corresponding machine orderings of the operations. Other representations are used too, especially in combination with genetic algorithms.

To be able to define the neighborhood functions, we need some extra notions. Given an instance and an operation v, $jp(v)$ and $js(v)$ denote the immediate predecessor and successor of v in the precedence relation A, provided they exist. Given a feasible schedule S and an operation v, $mp_S(v)$ and $ms_S(v)$ denote the immediate predecessor and successor of v in the orientation Ω_S, provided they exist. If the schedule S is clear from context, we delete the subscript S. Furthermore, $jp^2(v)$ denotes $jp(jp(v))$, provided it exists, and a similar notation is used for js, mp_S and ms_S. Two operations v and w are *adjacent* when $S(v) + p(v) = S(w)$. A *block* is a maximal sequence of size at least one, consisting of adjacent operations that are processed on the same machine and belong to a longest path. An operation of a block is *internal* if it is neither the first nor the last operation of that block.

Several neighborhood functions have been proposed in the literature. Most of these are not defined on a schedule S itself but on the corresponding orientation Ω_S. If Ω_S is changed into another feasible orientation Ω', $S_{\Omega'}$ is the corresponding neighbor of S. In this way neighbors of a given schedule are always left-justified.

The following properties[6,51,55] are helpful in obtaining reasonable neighborhood functions.

1. Given a feasible orientation, reversing an oriented edge on a longest path in the corresponding digraph results again in a feasible orientation.

2. If reversing an oriented edge of a feasible orientation Ω that is not on a longest path results in a feasible orientation Ω', then $S_{\Omega'}$ is at least as long as S_Ω.

3. Given a feasible orientation Ω, reversing an oriented edge (v, w) between two internal operations of a block results in a feasible schedule at least as long as S_Ω.

4. Given is a feasible orientation Ω. Let v and w be the first two operations of the first block of a longest path, and let w be an internal operation. Reversing (v, w) results in a feasible schedule at least as long as S_Ω. The same is true in case v and w are the last two operations of the last block of a longest path and v is internal.

In view of these properties, the simplest neighborhood functions are based on the reversal of exactly one edge of a given orientation. Van Laarhoven, Aarts and Lenstra[67] propose a neighborhood function N_1, which obtains a neighbor by interchanging two adjacent operations of a block. Matsuo, Suh and Sullivan[51] use a neighborhood function $N_{1\mathrm{a}}$ with the same interchanges, except those involving two internal operations. Nowicki and Smutnicki[55] use a neighborhood function $N_{1\mathrm{b}}$, excluding from $N_{1\mathrm{a}}$ the interchange

of the first two operations of the first block when the second one is internal and the interchange of the last two operations of the last block when the first is internal. For N_{1b}, neither a schedule with only one block nor one with only blocks of size one has a neighbor; note that such schedules are optimal.

Dell'Amico and Trubian[23] propose several neighborhood functions that may reverse more than one edge. Their neighborhood function N_2 obtains, for any two operations v and $w = ms(v)$ on a longest path, a neighboring orientation by permuting $mp(v)$, v and w, or by permuting v, w and $ms(w)$, such that v and w are interchanged and a feasible orientation results. Their neighborhood function N_{2a} excludes from N_2 the solutions for which both v and $w = ms(v)$ are internal. Their neighborhood function N_3 considers blocks of size at least two: a neighbor is obtained by positioning an operation v immediately in front of or after the other operations of its block, provided that the resulting orientation is feasible; otherwise, v is moved to the left or to the right as long as the orientation remains feasible.

While the above neighborhood functions are based on adjacent interchanges or *swaps*, Balas and Vazacopoulos[9] propose a neighborhood function N_4 that uses reinsertions or *jumps*. More precisely, N_4 considers any two operations v and w on the same machine such that v occurs prior to w on a longest path. A neighbor is obtained by inserting v immediately after w or w immediately before v. Sufficient conditions are derived under which these new schedules are feasible. If $mp(v)$ and $ms(w)$ are both on a longest path, they cannot improve the current schedule and are disregarded.

Adams, Balas and Zawack[2] propose a neighborhood function N_5, which may completely change one machine ordering. For every machine M' with an operation on a longest path, a neighbor is obtained by replacing the orientation on M' by any other feasible orientation.

The following neighborhood functions obtain a neighbor by changing several machine orderings at the same time. Relatively small modifications are made by the neighborhood function N_6 of Matsuo, Suh and Sullivan[51], which reorients at most three edges simultaneously. A neighbor is obtained by interchanging two adjacent operations v and $w = ms(v)$ of a block (except when they are both internal) and in addition by interchanging $jp^t(w)$ and $mp(jp^t(w))$ for some $t \geq 1$ and by interchanging $js(v)$ and $ms(js(v))$. The latter interchanges are executed only if certain additional conditions are satisfied; see their paper for details. Aarts, Van Laarhoven, Lenstra and Ulder[1] use a variant N_{6a} with $t = 1$.

Applegate and Cook[4] propose a neighborhood function N_7, which drastically changes the given orientation. Their neighborhood contains all feasible orientations that can be obtained by simultaneously replacing the orientation on $m - t$ machines by any other feasible orientation. Here, t is a small number depending on m.

Storer, Wu and Vaccari[64] use completely different representations of schedules. These are based on a modified version of the Giffler-Thompson algorithm (see Section 2.3). Suppose that at a certain point the earliest possible completion time of any unscheduled operation is equal to C and is achieved by operation v, and that T is the

earliest possible start time on machine $M(v)$. Then all unscheduled operations on $M(v)$ that can start no later than $T + \delta(C - T)$ are candidates for the next position on $M(v)$. Here, δ is a priori chosen in [0,1) (in experiments 0, 0.05 or 0.1); if δ approaches 1 all active schedules can be generated, while $\delta = 0$ gives only so-called *non-delay* schedules. Two representations are defined.

The representation R_8 represents a schedule by modified processing times for the operations. Using these, the modified Giffler-Thompson algorithm with the shortest processing time rule as selection rule uniquely determines a feasible orientation Ω, and S_Ω, computed with the original processing times, is the corresponding schedule. The neighborhood function N_8 now obtains a neighbor by increasing the processing times by amounts of time that are independently drawn from a uniform distribution on $(-\theta, \theta)$. Here, θ is a priori chosen (in experiments 10, 20 or 50). The representation R_9 represents a schedule by dividing the scheduling horizon into several time windows (in experiments 5, 10 or 20) and assigning one of a given set of dispatch rules to each window. The modified Giffler-Thompson algorithm determines a schedule by applying the dispatch rule of the corresponding window. The neighborhood function N_9 changes the dispatch rule for a window of a given schedule.

Genetic algorithms use two types of representations: the natural one, which is also used for the other algorithms, and the more artificial 'string representations'.

For the former type of representation Yamada and Nakano[69] propose a hyperneighborhood function N_{h1}. Given two schedules S and S', N_{h1} determines a neighbor using the Giffler-Thompson algorithm. When this algorithm has to choose from two or more operations, it takes, for a small $\epsilon > 0$, the operation that is first in S with probability $(1 - \epsilon)/2$, the operation that is first in S' with probability $(1 - \epsilon)/2$, and a random operation from the other available operations with probability ϵ.

Aarts, Van Laarhoven, Lenstra and Ulder[1] propose a hyperneighborhood function N_{h2}. Given two schedules S and S', N_{h2} determines a neighbor by repeating the following step $\lfloor nm/2 \rfloor$ times: choose a random arc (w, v) of S' and change S by reversing arc (v, w), provided it belongs to a longest path of S.

The latter type of representation encodes a schedule or its orientation into a string over a finite – usually binary – alphabet. Such representations facilitate the application of hyperneighborhood functions involving operations like 'crossover' and 'mutation'; see Goldberg[39, pp. 166–175]. There are a number of drawbacks, however. A schedule or orientation may have several representatives, or none. Conversely, a string does not have to represent a schedule, and if it does, it may be nontrivial to calculate the corresponding schedule. Although attempts have been made to circumvent these difficulties, the hyperneighborhood functions that operate on strings often have no meaningful effect in the context of the underlying problem. We will consider genetic algorithms using string representations in less detail.

5 Constructive algorithms with local search

In this section we discuss the *shifting bottleneck procedure* and its variants. These algorithms construct a complete schedule and apply local search to partial schedules on the way. A partial schedule is characterized by a partial orientation Ω on a subset $E' \in E$. Its length is defined as the longest path length in the digraph $(O, A \cup \Omega(E'))$.

The basic idea of the algorithms described here is as follows. The algorithm goes through m stages. At each stage, it orients all edges between operations on a specific machine. In this way, at the beginning of any stage all edges related to some machines have been oriented, while the edges related to the other machines are not yet oriented. Furthermore, at the end of each stage, it reoptimizes the current partial schedule. This is usually done by applying iterative best improvement using neighborhood function N_5, which revises the orientation on a machine scheduled before. Orienting or reorienting the edges related to one machine in an optimal way requires the solution of a single-machine problem, where the partial schedule defines release and delivery times and delayed precedence constraints. The algorithms discussed hereafter mainly differ by the order in which the m machines are considered, by the implementation of iterative best improvement, and by the single-machine algorithm used.

The original shifting bottleneck procedure SB1 of Adams, Balas and Zawack[2] orients at each stage the edges related to the *bottleneck machine*. This is the unscheduled machine for which the solution value to the corresponding single-machine problem is maximum; the delays between precedence-related operations are not taken into account. After scheduling a machine, iterative best improvement is applied during three cycles. In each cycle each scheduled machine is reconsidered once. The first cycle handles the machines in the order in which they were sequenced. After a cycle is completed, the machines are reordered according to decreasing solution values to the single-machine problems in the last cycle. When all of the machines have been scheduled, the cycles continue as long as improvements are found. Furthermore, after a phase of iterative best improvement, the orientations on several machines that have no operations on a longest path are deleted, and then these machines are rescheduled one by one.

Applegate and Cook[4] use almost the same algorithm. The main difference is that at each stage iterative improvement cycles continue until no improvement is found.

Dauzère-Pérès and Lasserre[20] were the first to take the delays between precedence-related operations into account. They develop a heuristic for the single-machine problem with delayed precedences and incorporate it into a shifting bottleneck variant.

Balas, Lenstra and Vazacopoulos[8] use their optimization algorithm for the single-machine problem with delayed precedences to determine the bottleneck machine in their procedure SB3. Their local search strategy differs from the one of Adams, Balas and Zawack[2] in some minor details; for instance, the number of cycles is limited to six. Again, after scheduling a new machine, they first apply iterative improvement, then delete the orientations on several non-critical machines, and reschedule these machines one by one. Their extended procedure SB4 takes the best solution of SB3 and a variant

of SB3 that reverses the order of the two reoptimizations procedures: first reschedule some non-critical machines, then apply regular iterative improvement.

Balas and Vazacopoulos[9] propose a rather different procedure SB-GLS. It reoptimizes partial schedules by applying their variable-depth search algorithm GLS (see Section 6.3) for a limited number of iterations using the jump neighborhood function N_4.

The shifting bottleneck procedure and its variants have been incorporated into other algorithms. Most of these employ some form of partial enumeration. Dorndorf and Pesch[25] embed a variant in a genetic algorithm; see Section 6.4.

Adams, Balas and Zawack[2] develop an algorithm PE-SB, which applies SB1 to the nodes of a partial enumeration tree. A node corresponds to a subset of machines that have been scheduled in a certain way. In each of its descendants one more machine is scheduled. The schedule is obtained by first solving the single-machine problem, with release and delivery times defined by the parent node, and then applying iterative improvement as in SB1. Descendants are created only for a few machines with highest solution values to the single-machine problem. A penalty function is used to limit the size of the tree. For details about the branching rule, the penalty function and the search strategy we refer the reader to the original paper.

Applegate and Cook[4] develop an algorithm Bottle-t, which employs partial enumeration in a different way. Bottle-t applies their shifting bottleneck variant described above as long as more than t machine are left unscheduled. For the last t machines it branches by selecting each remaining unscheduled machine in turn. The values $t = 4$, 5 and 6 were tested.

6 Iterative algorithms with local search

The algorithms presented in this section start from one or more given feasible schedules and manipulate these in an attempt to find better schedules. They can naturally be divided into threshold algorithms, taboo search algorithms, variable-depth search algorithms, and genetic algorithms.

6.1 Threshold algorithms

The basic threshold algorithms are iterative improvement, threshold accepting, and simulated annealing. We also consider some closely related variants. Unless stated otherwise, a schedule is represented in the ordinary way by the starting times or the orientation.

Iterative improvement is the simplest threshold algorithm. Aarts, Van Laarhoven, Lenstra and Ulder[1] test iterative improvement with the neighborhood functions N_1 and N_{6a}. To obtain a fair comparison with other algorithms they apply a *multi-start* strategy, i.e., they run the algorithm with several randomly generated start solutions until a limit on the total running time is reached, and take the best solution found over all individual runs.

The algorithm Shuffle of Applegate and Cook[4] uses the neighborhood function N_7. At each iteration, the schedule on a small number of heuristically selected machines remains fixed, and the schedule on the remaining machines is optimally revised by their branch-and-bound algorithm 'edge finder'. As initial solution they take the result of Bottle-5.

Storer, Wu and Vaccari[64] propose a variant of iterative improvement, called PS10, with representation R_8 and neighborhood function N_8. Given a solution, a fixed number of neighbors (in experiments 100 or 200) is determined, the best one of which becomes the new solution. They also test a standard iterative first improvement algorithm, called HSL10, with representation R_9 and neighborhood function N_9. Neighbors are generated randomly and the algorithm stops after a fixed number of iterations (in experiments 1000 or 2000).

Threshold accepting has only been implemented by Aarts, Van Laarhoven, Lenstra and Ulder[1]. Their algorithm TA1 uses the neighborhood function N_1. Threshold values are determined empirically.

Simulated annealing has been tested by several authors. Van Laarhoven, Aarts and Lenstra[67] use the neighborhood function N_1. Aarts, Van Laarhoven, Lenstra and Ulder[1] use N_1 (algorithm SA1) and N_{6a} (algorithm SA2).

Matsuo, Suh and Sullivan's[51] 'controlled search simulated annealing' algorithm SA-II is a bi-level variant, which also incorporates standard iterative improvement. Given a schedule S, a neighbor S' is selected using the neighborhood function N_6. S' is accepted or rejected by the simulated annealing criterion. In the latter case, S' is subjected to iterative improvement using N_6 again, and if the resulting local optimum improves on S, it is accepted as the new solution. Their method also differs from most other implementations of simulated annealing in that the acceptance probability for a schedule that is inferior to the current schedule is independent of the difference in schedule length.

6.2 Taboo search algorithms

The taboo search algorithm TS1 of Taillard[64] uses the neighborhood function N_1. After an arc (v, w) has been reversed, the interchange of w and its machine successor is put on the taboo list. Every 15 iterations a new length of the taboo list is randomly selected from a range between 8 and 14. The length of a neighbor is estimated in such a way that the estimate is exact when both operations involved are still on a longest path, and that it is a lower bound otherwise. Then, from the permissible neighbors the schedule of minimum estimated length is selected as the new schedule.

The algorithm TS2 of Barnes and Chambers[10] also uses N_1. Their taboo list has a fixed length. If no permissible moves exist, the list is emptied. The length of each neighbor is calculated exactly, not estimated. A start solution is obtained by taking the best from the active and non-delay schedules obtained by applying seven dispatch rules.

The algorithm TS3 of Dell'Amico and Trubian[23] uses the union of the neighborhoods generated by N_{2a} and N_3. The items on the taboo list are forbidden reorientations of arcs. Depending on the type of neighbor, one or more such items are on the list. The

length of the list depends on the fact whether the current schedule is shorter than the previous one and the best one, or not. Furthermore, the minimal and maximal allowable lengths of the list are changed after a given number of iterations. When all neighbors are taboo and do not satisfy the aspiration criterion, a random neighbor is chosen as the next schedule. A start solution is obtained by a procedure called 'bidir', which applies list scheduling simultaneously from the beginning and the end of the schedule.

Nowicki and Smutnicki's[55] algorithm TS-B combines taboo search with a backtracking scheme. In the taboo search part of their algorithm, the neighborhood function is a variant of N_{1b}, which only allows reorientations of arcs on a single longest path. The items on the taboo list are forbidden reorientations of arcs. The length of the list is fixed to 8. If no permissible neighbor exists, the following is done. If there is one neighbor only, which as a consequence is taboo, this one becomes the new schedule. Otherwise, the oldest items on the list are removed one by one until there is one non-taboo neighbor, and this one is chosen. A start solution is obtained by generating an active schedule using the shortest processing time rule or an insertion algorithm.

The backtracking scheme forces the taboo search to restart from promising situations encountered before. Suppose that at a certain point a new best schedule S is found. Let $R(S)$ be the set of feasible arc reversals in S, let T be the new taboo list, including the inverse of the reversal needed to obtain S, and let r be the reversal that will be made in the next iteration. Then, if $|R(S)| \geq 2$, the triple $(S, R(S) \setminus \{r\}, T)$ is stored on a list. This list has a maximum length of 5; if it is full, the oldest triple is deleted before a new one is stored. Each time the taboo search algorithm stops (by reaching a maximum number of iterations without improving the best schedule), the backtracking scheme initiates a new round of taboo search starting from the schedule, the set of reversals and the taboo list of the last stored triple. When the set of reversals has one element only, the triple is deleted from the list; otherwise, it is replaced by the same triple with the reversal that will be made in the next iteration excluded. Note that during this new round new triples can be added to the list.

6.3 Variable-depth search algorithms

Balas and Vazacopoulos'[9] *guided local search* algorithm GLS is based on the neighborhood function N_4. It differs from standard variable-depth search in the sense that trees are used instead of sequences. Each node of the tree corresponds to an orientation, and each child node is a neighbor of its parent. The number of children of a parent is restricted by a decreasing function of the level in the tree. The children are selected using estimates of the lengths of the associated schedules. When a node is obtained by inserting an operation just before or after another one, their relative order remains fixed in the orientation of all of its descendants. Other parts of the orientation are fixed too to ensure that each orientation occurs in at most one node in the tree. The size of the tree is kept small by limiting the number of children, by fixing parts of the orientations, and by bounding the depth of the tree by a logarithmic function of the number of operations. From the nodes in the tree one with smallest length is chosen as the root node

for the next tree, provided that it has a smaller length than the root of the current tree. Otherwise, a node is chosen that differs at least a certain number of reinsertions from the root, where a node with smaller length is chosen with higher probability. A start solution is generated by a randomized dispatch rule.

Balas and Vazacopoulos[9] propose two variants, which we call *iterated* and *reiterated* guided local search, or IGLS and RGLS-k. Starting from a solution generated by SB-GLS (see Section 5), IGLS repeats reoptimization cycles until no improvement is found. Each of these cycles removes the orientation on one machine, applies GLS for a limited number of trees, then adds the removed machine again, and applies GLS to the complete schedule for a limited number of trees. RGLS-k starts from a solution obtained by IGLS and repeats k cycles of the following type: remove the orientations on $\lfloor\sqrt{m}\rfloor$ randomly chosen machines, apply GLS for a limited number of trees, then add the removed machines again by applying SB-GLS (see Section 5), and finally apply IGLS.

6.4 Genetic algorithms

The genetic algorithm GA1 of Yamada and Nakano[69] determines, for every chosen pair of schedules of the current population, two hyperneighbors by using N_{h1}. From these four schedules two are selected for the next population: first the best schedule is chosen, and next the best unselected hyperneighbor is chosen.

Aarts, Van Laarhoven, Lenstra and Ulder[1] propose a genetic algorithm that incorporates iterative first improvement. In each iteration there is a population of solutions that are locally optimal with respect to either N_1 (algorithm GA-II1) or $N_{6\mathrm{a}}$ (algorithm GA-II2). The population is doubled in size by applying N_{h2} to randomly selected pairs of schedules of the population. Each hyperneighbor is subjected to iterative first improvement, using N_1 or $N_{6\mathrm{a}}$, and the extended population of local optima is reduced to its original size by choosing the best schedules. Then a next iteration is started. Start solutions are generated randomly, and iterative first improvement is applied to them before the genetic algorithm is started.

In the work of Davis[19], Falkenauer and Bouffouix[27] and Della Croce, Tadei and Volta[22] a string represents for each machine a preference list, which defines a preferable ordering of its operations. From such a list a schedule is calculated. Davis[19] and Falkenauer and Bouffouix[27] restrict themselves to non-delay schedules; Della Croce, Tadei and Volta[22] are able to represent other schedules as well. Falkenauer and Bouffouix[27] and Della Croce, Tadei and Volta[22] use the linear order crossover as hyperneighborhood function. See the original papers for details.

Nakano and Yamada[53] consider problem instances with exactly one operation for each job-machine pair. For each machine and each pair of jobs, they represent the order in which that machine executes those jobs by one bit. Thus, a schedule is represented by a string of $mn(n-1)/2$ bits. Since such a string may not represent a feasible orientation, they propose a method for finding a feasible string that is close to a given infeasible one. Two hyperneighbors are obtained by cutting two strings at the same point and exchanging their left parts.

Dorndorf and Pesch[25] propose a 'priority rule based genetic algorithm' GA-P, which uses the Giffler-Thompson algorithm. They use a string $(p_1, \ldots, p_{l-1})$, where p_i is a dispatch rule that resolves conflicts in the ith iteration of the algorithm. Two hyperneighbors are obtained by cutting two strings at the same point and exchanging their left parts. These authors also propose a second genetic algorithm, called GA-SB, which uses a shifting bottleneck procedure (see Section 5). A solution is represented by a sequence of the machines. A corresponding schedule is generated by a variant of SB1: each time it has to select an unoriented machine, it chooses the first unoriented machine in this sequence. The hyperneighborhood function used is the cycle crossover; see [36, p. 175]. In contrast to SB1, reoptimization is applied only when less than six machines are left unscheduled.

7 Other techniques

7.1 Constraint satisfaction

Constraint satisfaction algorithms consider the decision variant of the job shop scheduling problem: given an overall deadline, does there exist a feasible schedule meeting the deadline? Most algorithms of this type apply tree search and construct a schedule by assigning start times to the operations one by one. A *consistency checking* process removes inconsistent start times of not yet assigned operations. If it appears that a partial schedule cannot be completed to a feasible one, a *dead end* is encountered, and the procedure has to undo several assignments. Variable and value ordering heuristics determine the selection of a next operation and its start time. The algorithm stops when a feasible schedule meeting the deadline has been found or been proved not to exist. Note that it is also possible to establish lower bounds on the optimum with this technique.

Sadeh[60] developed an algorithm of this type, but its performance was poor. Nuijten and Aarts[56] designed new variable and value orderings and extensive consistency checking techniques. They restart the search from the beginning when a dead end occurs, and they also randomize the selection of a next operation and its start time. Their 'randomized constraint satisfaction' algorithm RCS performs quite well.

Pesch and Tetzlaff[58] describe a consistency checking process to test if a partial orientation can be extended to a feasible complete orientation within a given deadline. In order to find reasonable orientations of edges, they create subinstances of the given instance by deleting some of the jobs. Each such subinstance is solved to optimality by the branch-and-bound algorithm of Brucker, Jurisch and Sievers[13]. Next, they slightly increase the resulting optimal solution value to obtain a deadline for the subinstance and then apply consistency checking to identify edges that can be oriented in one way only. All oriented edges found in this way are aggregated into the original instance, except those that occur in a cycle. Finally the restricted original instance is solved to optimality, again by branch-and-bound.

7.2 Neural networks

Foo and Takefuji[30,31] describe a solution approach based on the deterministic neural network model with a symmetrically interconnected network, introduced by Hopfield and Tank[40]. The job shop scheduling problem is represented by a 2-dimensional matrix of neurons. Zhou, Cherkassky, Baldwin, and Olson[71] develop a neural network algorithm which uses a linear cost function instead of a quadratic one. For each operation there is one neuron in the network, and also the number of interconnections is linear in the number of operations. The algorithm improves the results of Foo and Takefuji both in terms of solution quality and network complexity. Altogether, applications of neural networks to the job shop scheduling problem are at an initial stage, and the reported computational results are poor up to now.

8 Computational results

The computational merits of job shop scheduling algorithms have often been measured by their performance on the notorious 10×10 instance FT10 of Fisher and Thompson[29]. Applegate and Cook[4] found that several instances of Lawrence[47] (LA21, LA24, LA25, LA27, LA29, LA38, LA40) pose a more difficult computational challenge. We have included the available computational results for these instances and, in addition, for two relatively easy instances (LA2, LA19) and for all remaining 15×15 instances of Lawrence (LA36, LA37, LA39). Each of these thirteen instances has exactly one operation for each job-machine pair.

Tables 2, 3, 4 and 5 present the computational results for most algorithms discussed in Sections 5, 6 and 7, as far as these are available. All results were taken from the literature, with the exception of the results for the algorithms of Applegate and Cook[4], which we obtained using their codes. Tables 2 and 3 give the individual results for the thirteen instances, Table 4 aggregates these results, and Table 5 contains the results for four algorithms that were only tested on instance FT10. Schedule lengths are printed in roman, computation times have been measured in CPU-seconds and are printed in italic, and blank spaces denote that no results are available.

In Tables 2 and 3, the values LB and UB are the best known lower and upper bounds on the optimal schedule lengths. For the instances LA21 and LA38, we ran the 'edge finder' algorithm of Applegate and Cook[4] to prove optimality of the best known solution values. The lower bound for LA29 was reported by Nuijten and Rogerie[57]. The best upper bounds are obtained by one or more of the algorithms in the tables, except the one for LA27, which was found by Carlier and Pinson[17], for LA29, which we found by running the 'shuffle' algorithm of Applegate and Cook[4] on a solution of Yamada and Nakano[70], and for LA40, which is due to Applegate and Cook[4]. Note that all instances but one have been solved to optimality. For each of the included algorithms, a superscript b followed by a number x indicates that the schedule lengths reported are the best ones obtained after x runs of the algorithm; the computation time is the total time over all

runs. A superscript m indicates that the schedule lengths are means over several runs; in this case, the computation time is the average over these runs. A superscript 1 refers to a single run.

For each algorithm and each instance, we computed the relative error, i.e., the percentage that the schedule length reported is above LB. Table 4 presents, for each algorithm, the mean and the standard deviation of these relative errors. Note that UB has already a mean relative error of 0.07. Table 4 also gives, for each algorithm, the total computation time, the computer used, and a computer independent total computation time. The latter values were computed using the normalization coefficients of Dongarra[24] and must be interpreted with care; their accuracy is at most up to two digits.

Figure 1 shows for each algorithm its mean relative error and its computer independent total computation time. Note that the time axis has a logarithmic scale.

The *shifting bottleneck* procedure SB1 of Adams, Balas and Zawack[2] is fast but gives poor results. The variants SB3 and SB4 of Balas, Lenstra and Vazacopoulos,[8] which take the delayed precedences into account, are more effective. It is worthwhile to combine the straight shifting bottleneck procedure with some form of partial enumeration, as is clear from the results obtained by algorithm PE-SB of Adams, Balas and Zawack[2] and by Bottle-5 and Bottle-6 of Applegate and Cook[4]. (Note that the values for Bottle-t given here differ from those reported in the original paper: we used the enumeration scheme as described in the paper; they implemented a different scheme.) Algorithm SB-GLS of Balas and Vazacopoulos[9] is in an entirely different category. Apparently, the performance of the shifting bottleneck procedure is significantly enhanced if variable-depth search with a fine-grained neighborhood function is used to reoptimize partial schedules.

Among *threshold algorithms* the best results are obtained by the simulated annealing algorithm of Aarts, Van Laarhoven, Lenstra and Ulder[1] and the iterative improvement algorithm Shuffle of Applegate and Cook[4].

Regarding *iterative improvement*, Aarts, Van Laarhoven, Lenstra and Ulder[1] report that their multi-start algorithm is inferior to threshold accepting and simulated annealing. Applegate and Cook's Shuffle algorithm works well, due to a neighborhood function that allows major changes in the schedule. We used our own outcomes of Bottle-5 as start solutions. The number t of machines to fix was chosen such that edge finder could rapidly fill in the remainder of the schedule. We set t=1 for FT10, LA2 and LA19, t=2 for LA21, LA24 and LA24, and t=5 for the other instances. The results for these values of t are reported under Shuffle1. We also carried out more time consuming runs with t=1 for FT10 and LA2-LA24, t=3 for LA29, LA36 and LA37, and t=4 for LA27, LA38, LA39 and LA40. The outcomes, reported under Shuffle2, are good but expensive. Storer, Wu and Vaccari[64] give very few computational results for their variants of iterative improvement. Their results for the instance FT10 are poor. It seems that their search strategy or their neighborhood function is not powerful enough.

The *threshold accepting* algorithm TA1 of Aarts, Van Laarhoven, Lenstra and Ulder[1]

Table 2: Results for six instances

algorithm	FT10		LA2		LA19		LA21		LA24		LA25	
n	10		10		10		15		15		15	
m	10		5		10		10		10		10	
LB	930		655		842		1046		935		977	
UB	930		655		842		1046		935		977	
shifting bottleneck												
SB1[1]	1015	*10*	720	*2*	875	*7*	1172	*2*	1000	*25*	1048	*28*
SB3[1]	981	*6*	667	*1*	902	*4*	1111	*11*	976	*11*	1012	*13*
SB4[1]	940	*11*	667	*1*	878	*9*	1071	*20*	976	*20*	1012	*23*
SB-GLS[1]	930	*13*	666	*1*	852	*12*	1048	*25*	941	*26*	993	*26*
PE-SB[1]	930	*851*	669	*12*	860	*240*	1084	*362*	976	*434*	1017	*430*
Bottle-4[1]	938	*7*	667	*1*	863	*10*	1094	*17*	983	*26*	1029	*22*
Bottle-5[1]	938	*7*	662	*8*	847	*65*	1084	*46*	983	*63*	1001	*48*
Bottle-6[1]	938	*8*			842	*201*	1084	*301*	958	*200*	1001	*100*
threshold algorithms												
Shuffle1[1]	938	*25*	655	*8*	842	*73*	1055	*955*	971	*421*	997	*74*
Shuffle2[1]	938	*25*	655	*8*	842	*73*	1046	*87478*	965	*65422*	992	*98*
TA1^{m5}	1003	*99*	693	*19*	925	*94*	1104	*243*	1014	*235*	1075	*255*
SA1^{m5}	969	*99*	669	*19*	855	*94*	1083	*243*	962	*235*	1003	*255*
SA2^{m5}	977	*99*	658	*19*	854	*94*	1078	*243*	960	*235*	1019	*255*
SA1$^{1}_{t\to\infty}$							1053		935		983	
SA^{m5}	985	*779*	663	*117*	853	*830*	1067	*1991*	966	*2098*	1004	*2133*
SA^{b5}	951	*3895*	655	*585*	848	*4150*	1063	*9955*	952	*10490*	992	*10665*
SA-II[1]	946	*987*	655	*3*	842	*115*	1071	*205*	973	*199*	991	*180*
taboo search												
TS1^{b5}	930						1047					
TS2^{b2}	930	*450*	655	*60*	843	*450*	1050	*480*	946	*480*	988	*480*
TS3^{m5}	948	*156*	655	*19*	846	*104*	1057	*199*	943	*182*	980	*192*
TS3^{b5}	935	*779*	655	*94*	842	*519*	1048	*994*	941	*909*	979	*958*
TS-B[1]	930	*30*	655	*8*	842	*60*	1055	*21*	948	*184*	988	*155*
TS-B^{b3}	930		655		842		1047		939		977	
variable-depth search												
GLS^{b4}	930	*153*	655	*33*	842	*134*	1047	*222*	938	*243*	982	*330*
IGLS[1]	930	*45*	655	*8*	842	*74*	1048	*112*	937	*175*	977	*224*
RGLS-5[1]	930	*247*	655	*8*	842	*269*	1046	*612*	935	*682*	977	*616*
genetic algorithms												
GA-II1^{m5}	978	*99*	668	*19*	863	*94*	1084	*243*	970	*235*	1016	*255*
GA-II2^{m5}	982	*99*	659	*19*	859	*94*	1085	*243*	981	*235*	1010	*255*
GA-II2$^{1}_{t\to\infty}$							1055		938		985	
GA2^{m5}	965	*628*	685	*284*	855	*651*	1113	*1062*	1000	*1045*	1029	*1052*
GA2^{b5}	946	*3140*	680	*1420*	850	*3255*	1097	*5310*	984	*5275*	1018	*5260*
GA-P[1]	960	*933*	681	*108*	880	*191*	1139	*352*	1014	*352*	1014	*350*
GA-SB$^{m2}_{40}$	938	*107*	666	*16*	863	*77*	1074	*135*	960	*137*	1008	*134*
GA-SB$^{m2}_{60}$					848	*161*	1074	*293*	957	*289*	1007	*229*
constraint satisfaction												
RCS^{b5}	930	*2955*	655	*110*	848	*1455*	1069	*7600*	942	*7385*	981	*7360*

schedule length in roman, *computation time in seconds in italic*

Table 3: Results for seven instances

algorithm	LA27		LA29		LA36		LA37		LA38		LA39		LA40	
n	20		20		15		15		15		15		15	
m	10		10		15		15		15		15		15	
LB	1235		1142		1268		1397		1196		1233		1222	
UB	1235		1153		1268		1397		1196		1233		1222	
shifting bottleneck														
SB1[1]	1325	*45*	1294	*48*	1351	*47*	1485	*61*	1280	*58*	1321	*72*	1326	*77*
SB3[1]	1272	*19*	1227	*21*	1319	*28*	1425	*26*	1318	*30*	1278	*25*	1266	*26*
SB4[1]	1272	*38*	1227	*39*	1319	*56*	1425	*53*	1294	*59*	1278	*51*	1262	*52*
SB-GLS[1]	1243	*30*	1182	*44*	1268	*55*	1397	*37*	1208	*56*	1249	*48*	1242	*56*
PE-SB[1]	1291	*837*	1239	*892*	1305	*735*	1423	*837*	1255	*1079*	1273	*669*	1269	*899*
Bottle-4[1]	1307	*31*	1220	*31*	1326	*23*	1444	*14*	1299	*46*	1301	*42*	1295	*22*
Bottle-5[1]	1288	*92*	1220	*91*	1316	*153*	1444	*56*	1299	*96*	1291	*134*	1295	*24*
Bottle-6[1]	1286	*666*	1218	*280*	1299	*321*	1442	*562*	1268	*182*	1279	*192*	1255	*154*
threshold algorithms														
Shuffle1[1]	1280	*98*	1219	*95*	1295	*171*	1437	*64*	1294	*104*	1268	*178*	1276	*43*
Shuffle2[1]	1269	*604*	1191	*15358*	1275	*3348*	1422	*1577*	1267	*17799*	1257	*6745*	1238	*150*
TA1[m5]	1289	*492*	1262	*471*	1385	*602*	1469	*636*	1323	*636*	1305	*592*	1295	*597*
SA1[m5]	1282	*492*	1233	*471*	1307	*602*	1440	*636*	1235	*636*	1258	*592*	1256	*597*
SA2[m5]	1275	*492*	1225	*471*	1308	*602*	1451	*636*	1243	*636*	1263	*592*	1254	*597*
$SA1^{1}_{t\to\infty}$	1249		1185						1208				1225	
SA[m5]	1273	*4535*	1226	*4408*	1300	*5346*	1442	*5287*	1227	*5480*	1258	*5766*	1247	*5373*
SA[b5]	1269	*22675*	1218	*22040*	1293	*26730*	1433	*26435*	1215	*27400*	1248	*28830*	1234	*26865*
SA-II[1]	1274	*286*	1196	*267*	1292	*624*	1435	*577*	1231	*672*	1251	*660*	1235	*603*
taboo search														
TS1[b5]	1240		1170						1202					
TS2[b2]	1250	*600*	1194	*600*	1278	*540*	1418	*540*	1211	*540*	1237	*540*	1228	*540*
TS3[m5]	1252	*254*	1194	*281*	1289	*238*	1423	*242*	1210	*257*	1254	*238*	1235	*237*
TS3[b5]	1242	*1271*	1182	*1407*	1278	*1192*	1409	*1211*	1203	*1283*	1242	*1189*	1233	*1183*
TS-B[1]	1259	*66*	1164	*493*	1275	*623*	1422	*443*	1209	*165*	1235	*325*	1234	*322*
TS-B[b3]	1236		1160		1268		1407		1196		1233		1229	
variable-depth search														
GLS[b4]	1236	*435*	1157	*627*	1269	*455*	1400	*268*	1208	*464*	1233	*577*	1233	*367*
IGLS[1]	1240	*210*	1164	*369*	1268	*179*	1397	*146*	1198	*299*	1233	*434*	1234	*332*
RGLS-5[1]	1235	*315*	1164	*1062*	1268	*920*	1397	*822*	1196	*1281*	1233	*1131*	1224	*1590*
genetic algorithms														
GA-II1[m5]	1303	*492*	1290	*471*	1324	*602*	1449	*636*	1285	*636*	1279	*592*	1273	*597*
GA-II2[m5]	1300	*492*	1260	*471*	1310	*602*	1450	*636*	1283	*636*	1279	*592*	1260	*597*
$GA\text{-}II2^{1}_{t\to\infty}$	1265		1217						1248				1233	
GA2[m5]	1322	*1555*	1257	*1550*	1330	*1880*	1526	*1872*	1282	*1887*	1332	*1870*	1297	*1853*
GA2[b5]	1308	*7775*	1238	*7550*	1305	*9400*	1519	*9360*	1273	*9435*	1315	*9350*	1278	*9265*
GA-P[1]	1378	*565*	1336	*570*	1373	*524*	1498	*520*	1296	*525*	1351	*525*	1321	*526*
$GA\text{-}SB^{m2}_{40}$	1272	*242*	1204	*241*	1317	*336*	1484	*350*	1251	*336*	1282	*327*	1274	*348*
$GA\text{-}SB^{m2}_{60}$	1269	*446*	1210	*453*	1317	*688*	1446	*666*	1241	*666*	1277	*687*	1252	*698*
constraint satisfaction														
RCS[b5]	1285	*13950*	1208	*5660*	1292	*11165*	1411	*12760*	1278	*14075*	1233	*13620*	1247	*12875*

schedule length in roman, *computation time in seconds in italic*

Table 4: Summary of results

algorithm	authors	mean relative error	standard deviation of relative error	total computation time	computer	computer independent total computation time
LB		0	0			
UB		0.07	0.27			
shifting bottleneck						
SB1[1]	Adams et al.[2]	8.10	2.50	*483*	VAX 780/11	*60*
SB3[1]	Balas et al.[8]	4.81	2.40	*222*	Sparc 330	*560*
SB4[1]	Balas et al.[8]	3.78	2.06	*432*	Sparc 330	*1,100*
SB-GLS[1]	Balas & Vazacopoulos[9]	1.03	0.98	*430*	Sparc 330	*1,100*
PE-SB[1]	Adams et al.[2]	3.56	2.01	*10,742*	VAX 780/11	*1,300*
Bottle-4[1]	Applegate & Cook[4]	4.69	2.11	*293*	Sparc ELC	*730*
Bottle-5[1]	Applegate & Cook[4]	3.95	2.38	*884*	Sparc ELC	*2,200*
Bottle-6[1]	Applegate & Cook[4]	3.03	1.89	*3,175*	Sparc ELC	*7,900*
threshold algorithms						
Shuffle1[1]	Applegate & Cook[4]	2.96	2.47	*2,307*	Sparc ELC	*5,800*
Shuffle2[1]	Applegate & Cook[4]	1.86	1.79	*198,685*	Sparc ELC	*500,000*
TA1^{m5}	Aarts et al.[1]	7.63	2.27	*4,971*	VAX 8650	*3,500*
SA1^{m5}	Aarts et al.[1]	3.30	1.58	*4,971*	VAX 8650	*3,500*
SA2^{m5}	Aarts et al.[1]	3.34	1.68	*4,971*	VAX 8650	*3,500*
SA1$^{1}_{t\to\infty}$	Aarts et al.[1]	0.78	1.02	*40,000*	VAX 8650	*30,000*
SA^{m5}	Van Laarhoven et al.[67]	3.03	1.75	*44,143*	VAX 785	*8,400*
SA^{b5}	Van Laarhoven et al.[67]	1.97	1.59	*220,715*	VAX 785	*42,000*
SA-II[1]	Matsuo et al.[51]	2.12	1.41	*5,378*	VAX 780/11	*670*
taboo search						
TS1^{b5}	Taillard[65]					
TS2^{b2}	Barnes & Chambers[10]	0.99	1.19	*5,820*	IBM RS 6000	*70,000*
TS3^{m5}	Dell'Amico & Trubian[23]	1.39	1.13	*2,598*	PC 386	*1,300*
TS3^{b5}	Dell'Amico & Trubian[23]	0.73	0.89	*12,989*	PC 386	*6,500*
TS-B[1]	Nowicki & Smutnicki[55]	0.91	0.73	*2,895*	AT 386 DX	*1,400*
TS-B^{b3}	Nowicki & Smutnicki[55]	0.27	0.47	*8,685*	AT 386 DX	*4,300*
variable-depth search						
GLS^{b4}	Balas & Vazacopoulos[9]	0.35	0.45	*4,306*	Sparc 330	*11,000*
IGLS[1]	Balas & Vazacopoulos[9]	0.30	0.56	*2,606*	Sparc 330	*6,500*
RGLS-5[1]	Balas & Vazacopoulos[9]	0.16	0.53	*9,554*	Sparc 330	*24,000*
genetic algorithms						
GA-II1^{m5}	Aarts et al.[1]	4.84	2.79	*4,971*	VAX 8650	*3,500*
GA-II2^{m5}	Aarts et al.[1]	4.39	2.43	*4,971*	VAX 8650	*3,500*
GA-II2$^{1}_{t\to\infty}$	Aarts et al.[1]			*40,000*	VAX 8650	*30,000*
GA2^{m5}	Della Croce et al.[22]	6.24	2.28	*17,189*	PC 486/25	*12,000*
GA2^{b5}	Della Croce et al[22].	4.96	2.33	*85,945*	PC 486/25	*61,000*
GA-P[1]	Dorndorf & Pesch[25]	7.92	3.73	*6,041*	DEC 3100	*9,700*
GA-SB$^{m2}_{40}$	Dorndorf & Pesch[25]	3.45	1.48	*2,787*	DEC 3100	*4,500*
GA-SB$^{m2}_{60}$	Dorndorf & Pesch[25]	3.15	1.28	*5,523*	DEC 3100	*8,800*
constraint satisfaction						
RCS^{b5}	Nuijten & Aarts[56]	1.98	2.25	*110,970*	Sparc ELC	*280,000*

relative error in percents, *computation time in seconds in italic*;
computer independent computation times are rough estimates

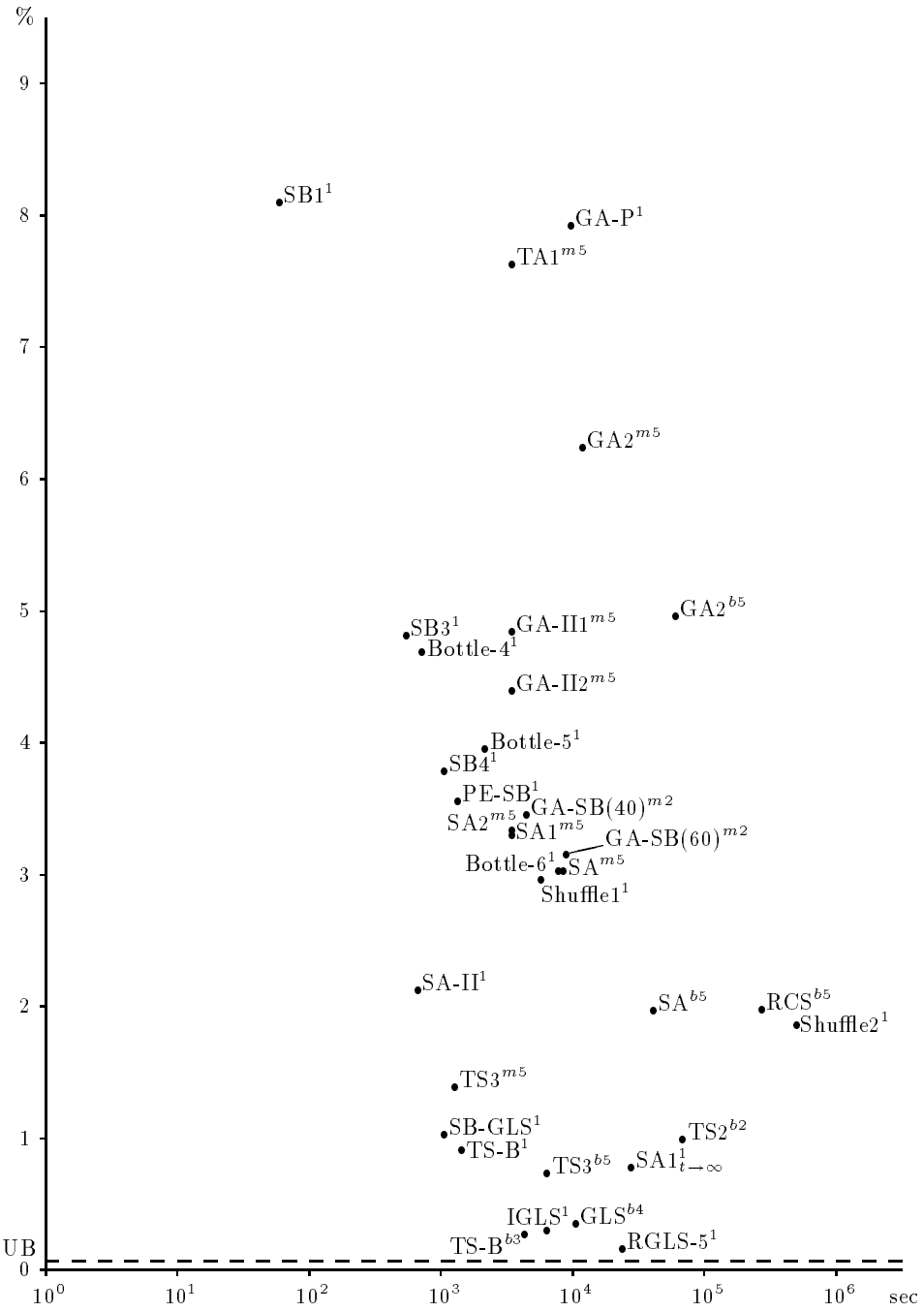

Figure 1: Relation between mean relative error and computer independent total computation time

Table 5: Miscellaneous results for FT10

algorithm	authors	length	time	computer
threshold algorithms				
PS10[1]	Storer et al.[64]	976		
HSL10[1]	Storer et al.[64]	1006		
genetic algorithms				
GA1[b600]	Yamada & Nakano[69]	930	*3600000*	Sparc 2
GA3[1]	Nakano & Yamada[53]	965		

schedule length in roman, *computation time in seconds in italic*

competes with their simulated annealing algorithm in case simulated annealing finds an optimal schedule. Otherwise, threshold accepting is outperformed by simulated annealing. Almost all instances in our table belong to the latter category.

The *simulated annealing* algorithm SA of Van Laarhoven, Aarts and Lenstra[67] produces reasonable results. Results of the same quality are obtained by the algorithms SA1 and SA2 of Aarts, Van Laarhoven, Lenstra and Ulder[1] with a standard cooling schedule; an extremely slow cooling schedule ($SA1_{t\to\infty}$) gives very good results. To compute the mean and the standard deviation of the relative errors for the latter cooling schedule, we estimated the values for the missing entries. It is remarkable that the standard cooling schedule behaves similarly for the neighborhood functions N_1 (SA1) and N_{6a} (SA2). Good results are obtained by the bi-level variant SA-II of Matsuo, Suh and Sullivan[51]. In comparison to other approximative approaches, simulated annealing may require large running times, but it yields consistently good solutions with a modest amount of human implementation effort and relatively little insight into the combinatorial structure of the problem type under consideration.

The advent of *taboo search* has changed the picture. Methods of this type produce excellent solutions in reasonable times, but these benefits come at the expense of a non-trivial amount of testing and tuning. Although few data are available, Taillard's[65] algorithm TS1 seems to perform extremely well. Also very good results are obtained by algorithm TS2 of Barnes and Chambers[10]. Dell'Amico and Trubian's[23] algorithm TS3 obtained even better results; apparently, their complicated neighborhood function is very effective. The algorithm TS-B of Nowicki and Smutnicki[55], which applies taboo search and traces its way back to promising but rejected changes, is one of the current champions for job shop scheduling. For our thirteen instances it achieves a mean relative error of only 0.27% for the best result out of three runs.

Like TS-B, the variable-depth search algorithm GLS of Balas and Vazacopoulos[9] combines conceptual elegance and computational excellence. GLS achieves a mean relative error of 0.35% for the best result out of four runs; it needs more time than TS-B, however. The iterated and reiterated variants IGLS and RGLS-k, which apply various reoptimization cycles to partial and complete solutions, perform still better. The best results reported so far have been obtained by RGLS-5: a single run achieves a mean relative error of only 0.16% and finds the optimum for eleven out of thirteen instances.

For many *genetic algorithms* no results for our instances are available. Sometimes only the result for FT10 is given. Yamada and Nakano[69] found a schedule of length 930 four times among 600 trials. They also tested their algorithm GA1 on four 20-job 20-machine instances, but their outcomes are on average 6.9% above the best known upper bounds. The results obtained by Aarts, Van Laarhoven, Lenstra and Ulder[1] are not very strong. Their algorithm GA-II2 (using neighborhood function N_{6a}) performs slightly better than GA-II1 (using N_1).

As for genetic algorithms using string representations, the results obtained by Della Croce, Tadei and Volta's[22] algorithm GA2 (published by Della Croce, Tadei and Rolando[21]) and by Nakano and Yamada's[53] algorithm GA3 are poor. The algorithm GA-P of Dorndorf and Pesch[25] is even worse. Their algorithm GA-SB, which incorporates a shifting bottleneck variant, produces reasonable results. Values are reported for runs with population sizes of 40 and 60.

The *constraint satisfaction* algorithm of Nuijten and Aarts[56] produces good results but needs a lot of time. Pesch and Tetzlaff's[58] set of test instances contains only three problems from our set. On eight 10×10 instances, their best parameter settings give significantly better solutions than algorithm PE-SB in about the same amount of time. For the *neural network* approaches no computational results are available that allow a proper comparison with other techniques.

9 Conclusion

9.1 Review

The local search algorithms discussed in this survey cover a broad range from straightforward to rather involved approaches. In general, the best results are obtained by taboo search and variable-depth search. The 'reiterated guided local search' algorithm of Balas and Vazacopoulos outperforms the other methods in terms of solution quality. The algorithm of Nowicki and Smutnicki, which combines taboo search with backtracking, is a close second and needs much less time. The recent shifting bottleneck algorithm of Balas and Vazacopoulos, which reoptimizes partial schedules by variable-depth search, is an effective and very fast alternative.

The other shifting bottleneck variants are not competitive anymore. For job shop scheduling, simulated annealing does not seem to be attractive either. It can yield very good solutions, but only if time is of no concern.

Genetic algorithms perform poorly up to now. Often the neighborhood function applied in combination with the schedule representation chosen does not generate meaningful changes and it is hard to find improvements. Only when some kind of local search is embedded at a second level, the computational results are reasonable.

Constraint satisfaction is a promising technique and needs further investigation. It is too early to make an assessment of the use of neural networks for job shop scheduling.

A word of caution is in order regarding the validity of our conclusions. We have col-

lected and compared the computational results reported on a set of benchmark instances. These problems are just on the borderline of being in reach of optimization algorithms. Further experiments on larger instances are required to improve our insights into the performance of the various breeds of the local search family.

9.2 Preview

There is still considerable room for improving local search approaches to the job shop scheduling problem. As shown in Figure 1, none of the existing algorithms achieves an average error of less than 2% within 1000 seconds total computation time.

We have observed that many approaches operate at two levels, with, for instance, schedule construction, partial enumeration or local search with big changes at the top level, and local search with smaller changes at the bottom level. Such hybrid approaches are in need of a more systematic investigation. It might also be interesting to design a three-level approach with neighborhoods of smaller size towards the bottom.

The flexibility of local search and the results reported here provide a promising basis for the application of local search to more general scheduling problems. An example of practical interest is the multiprocessor job shop, where each production stage has a set of parallel machines rather than a single one. Finding a schedule involves assignment as well as sequencing decisions. This is a difficult problem, for which no effective solution methods exist.

Applying local search to large instances of scheduling problems requires the design of data structures that allow fast incremental computations of, for example, longest paths. Johnson[42] has shown that sophisticated data structures play an important role in the application of local search to large traveling salesman problems.

Our survey has been predominantly of a computational nature. There are several related theoretical questions about the complexity of local search. A central concept in this respect is PLS-completeness[43]. Many of the neighborhood functions defined in Section 4 define a PLS-problem, which may be PLS-complete. There are also complexity issues regarding the parallel execution of local search. For example, for some of the neighborhood functions it may be possible to verify local optimality in polylog parallel time.

Acknowledgements

We are grateful to David Applegate and Bill Cook for making their codes available to us. We thank many colleagues for providing information about their computational work and for commenting on an earlier draft of this paper. This research was partially supported by Human Capital and Mobility Project ERB-CHRX-CT93-0087.

References

1. E.H.L. Aarts, P.J.M. van Laarhoven, J.K. Lenstra, and N.L.J. Ulder, 1994. A Computational Study of Local Search Algorithms for Job Shop Scheduling, *ORSA Journal on Computing 6*, 118-125.
2. J. Adams, E. Balas, and D. Zawack, 1988. The Shifting Bottleneck Procedure for Job Shop Scheduling, *Management Science 34*, 391-401.
3. S.B. Akers, 1956. A Graphical Approach to Production Scheduling Problems, *Operations Research 4*, 244-245.
4. D. Applegate and W. Cook, 1991. A Computational Study of the Job-Shop Scheduling Problem, *ORSA Journal on Computing 3*, 149-156.
5. K.R. Baker and Z.-S. Su, 1974. Sequencing with Due-Dates and Early Start Times to Minimize Maximum Tardiness, *Naval Research Logistics Quarterly 21*, 171-176.
6. E. Balas, 1969. Machine Sequencing via Disjunctive Graphs: an Implicit Enumeration Algorithm, *Operations Research 17*, 941-957.
7. E. Balas, 1985. On the Facial Structure of Scheduling Polyhedra, *Mathematical Programming Study 24*, 179-218.
8. E. Balas, J.K. Lenstra, and A. Vazacopoulos, 1995. The One-Machine Problem with Delayed Precedence Constraints and Its Use in Job Shop Scheduling, *Management Science 41*, 94-109.
9. E. Balas and A. Vazacopoulos, 1994. Guided Local Search with Shifting Bottleneck for Job Shop Scheduling, Management Science Research Report #MSRR-609, Graduate School of Industrial Administration, Carnegie Mellon University, Pittsburgh, Pennsylvania.
10. J.W. Barnes and J.B. Chambers, 1995. Solving the Job Shop Scheduling Problem Using Tabu Search, *IIE Transactions 27*, 257-263.
11. P. Bratley, M. Florian, and P. Robillard, 1973. On Sequencing with Earliest Starts and Due Dates with Application to Computing Bounds for the $(n/m/G/F_{\max})$ Problem, *Naval Research Logistics Quarterly 20*, 57-67.
12. P. Brucker, 1988. An Efficient Algorithm for the Job-Shop Problem with Two Jobs, *Computing 40*, 353-359.
13. P. Brucker, B. Jurisch, and B. Sievers, 1994. A Branch and Bound Algorithm for the Job-Shop Scheduling Problem, *Discrete Applied Mathematics 49*, 107-127.
14. J. Carlier, 1982. The One-Machine Sequencing Problem, *European Journal of Operational Research 11*, 42-47.
15. J. Carlier and E. Pinson, 1989. An Algorithm for Solving the Job-Shop Problem, *Management Science 35*, 164-176.
16. J. Carlier and E. Pinson, 1990. A Practical Use of Jackson's Preemptive Schedule for Solving the Job-Shop Problem, *Annals of Operations Research 26*, 269-287.
17. J. Carlier and E. Pinson, 1994. Adjustments of Heads and Tails for the Job-Shop Problem, *European Journal of Operational Research 78*, 146-161.
18. V. Černy, 1985. Thermodynamical Approach to the Traveling Salesman Problem, *Journal of Optimization Theory and Applications 45*, 41-51.
19. L. Davis, 1985. Job Shop Scheduling with Genetic Algorithms, in *Proceedings of an International Conference on Genetic Algorithms and Their Applications*, J.J. Grefenstette (ed.), Carnegie Mellon University, Pittsburgh, Pennsylvania, 136-140.
20. S. Dauzère-Pérès and J.B. Lasserre, 1993. A Modified Shifting Bottleneck Procedure for Job-Shop Scheduling, *International Journal of Production Research 31*, 923-932.

21. F. Della Croce, R. Tadei, and R. Rolando, 1993. Solving a Real World Project Scheduling Problem with a Genetic Algorithm, *Belgian Journal of Operations Research, Statistics and Computer Science 33*, 65-78.
22. F. Della Croce, R. Tadei, and G. Volta, 1995. A Genetic Algorithm for the Job Shop Problem, *Computers and Operations Research 22*, 15-24.
23. M. Dell'Amico and M. Trubian, 1993. Applying Tabu Search to the Job-Shop Scheduling Problem, *Annals of Operations Research 41*, 231-252.
24. J.J. Dongarra, 1993. Performance of Various Computers Using Standard Linear Equations Software, Report CS-89-85, Computer Science Department, University of Tennessee, Knoxville, Tennessee.
25. U. Dorndorf and E. Pesch, 1995. Evolution Based Learning in a Job Shop Scheduling Environment, *Computers and Operations Research 22*, 25-40.
26. G. Dueck and T. Scheuer, 1990. Threshold Accepting; a General Purpose Optimization Algorithm, *Journal of Computational Physics 90*, 161-175.
27. E. Falkenauer and S. Bouffouix, 1991. A Genetic Algorithm for Job Shop, in *Proceedings of the 1991 IEEE Internationial Conference on Robotics and Automation*, Los Alamitos, California, 824-829.
28. M.L. Fisher, B.J. Lageweg, J.K. Lenstra, and A.H.G. Rinnooy Kan, 1983. Surrogate Duality Relaxation for Job Shop Scheduling, *Discrete Applied Mathematics 5*, 65-75.
29. H. Fisher and G.L. Thompson, 1963. Probabilistic Learning Combinations of Local Job-Shop Scheduling Rules, in J.F. Muth and G.L. Thompson (eds.), *Industrial Scheduling*, Prentice Hall, Englewood Cliffs, New Jersey, 225-251.
30. Y.P.S. Foo and Y. Takefuji, 1988. Stochastic Neural Networks for Solving Job-Shop Scheduling: Part 1. Problem Representation, in *IEEE International Conference on Neural Networks*, IEEE San Diego section & IEEE TAB Neural Network Committee, San Diego, California, 275-282.
31. Y.P.S. Foo and Y. Takefuji, 1988. Stochastic Neural Networks for Solving Job-Shop Scheduling: Part 2. Architecture and Simulations, in *IEEE International Conference on Neural Networks*, IEEE San Diego section & IEEE TAB Neural Network Committee, San Diego, California, 283-290.
32. B. Giffler and G.L. Thompson, 1960. Algorithms for Solving Production Scheduling Problems, *Operations Research 8*, 487-503.
33. F. Glover, 1989. Tabu Search - Part I, *ORSA Journal on Computing 1*, 190-206.
34. F. Glover, 1990. Tabu Search - Part II, *ORSA Journal on Computing 2*, 4-32.
35. F. Glover, E. Taillard, and D. de Werra, 1993. A User's Guide to Tabu Search, *Annals of Operations Research 41*, 3-28.
36. D.E. Goldberg, 1989. *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, Reading, Massachusetts.
37. R. Haupt, 1989. A Survey of Priority Rule-Based Scheduling, *OR Spektrum 11*, 3-16.
38. N. Hefetz and I. Adiri, 1982. An Efficient Optimal Algorithm for the Two-Machines Unit-Time Jobshop Schedule-Length Problem, *Mathematics of Operations Research 7*, 354-360.
39. J.H. Holland, 1975. *Adaptation in Natural and Artificial Systems*, The University of Michigan Press, Ann Arbor, Michigan.
40. J.J. Hopfield and D.W. Tank, 1985. Neural Computation of Decisions in Optimization Problems, *Biological Cybernetics 55*, 141-152.
41. J.R. Jackson, 1956. An Extension of Johnson's Results on Job Lot Scheduling, *Naval Research Logistics Quarterly 3*, 201-203.

42. D.S. Johnson, 1990. Data Structures for Traveling Salesmen, in *SWAT90, 2nd Scandinavian Workshop on Algorithm Theory*, J.R. Gilbert, R. Karlsson (eds.), Springer, Berlin, 287-305.
43. D.S. Johnson, C.H. Papadimitriou, and M. Yannakakis, 1988. How Easy is Local Search?, *Journal of Computer and System Sciences 37*, 79-100.
44. S. Kirkpatrick, C.D. Gelatt, Jr., and M.P. Vecchi, 1983. Optimization by Simulated Annealing, *Science 220*, 671-680.
45. B.J. Lageweg, J.K. Lenstra, and A.H.G. Rinnooy Kan, 1976. Minimizing Maximum Lateness on One Machine: Computational Experience and Some Applications, *Statistica Neerlandica 30*, 25-41.
46. B.J. Lageweg, J.K. Lenstra, and A.H.G. Rinnooy Kan, 1977. Job-Shop Scheduling by Implicit Enumeration, *Management Science 24*, 441-450.
47. S. Lawrence, 1984. *Resource Constrained Project Scheduling: an Experimental Investigation of Heuristic Scheduling Techniques (Supplement)*, Graduate School of Industrial Administration, Carnegie Mellon University, Pittsburgh, Pennsylvania.
48. J.K. Lenstra and A.H.G. Rinnooy Kan, 1979. Computational Complexity of Discrete Optimization Problems, *Annals of Discrete Mathematics 4*, 121-140.
49. J.K. Lenstra, A.H.G. Rinnooy Kan, and P. Brucker, 1977. Complexity of Machine Scheduling Problems, *Annals of Discrete Mathematics 1*, 343-362.
50. S. Lin and B.W. Kernighan, 1973. An Effective Heuristic Algorithm for the Traveling-Salesman Problem, *Operations Research 21* 498-516.
51. H. Matsuo, C.J. Suh, and R.S. Sullivan, 1988. A Controlled Search Simulated Annealing Method for the General Jobshop Scheduling Problem, Working paper 03-04-88, Graduate School of Business, University of Texas, Austin.
52. G.B. McMahon and M. Florian, 1975. On Scheduling with Ready Times and Due Dates to Minimize Maximum Lateness, *Operations Research 23*, 475-482.
53. R. Nakano and T. Yamada, 1991. Conventional Genetic Algorithm for Job Shop Problems, *Proceedings of the Fourth International Conference on Genetic Algorithms*, R.K. Belew, L.B. Booker (eds.), San Diego, California, 474-479.
54. L. Németi, 1964. Das Reihenfolgeproblem in der Fertigungsprogrammierung und Linearplanung mit Logischen Bedingungen, *Mathematica (Cluj) 6*, 87-99.
55. E. Nowicki and C. Smutnicki, 1996. A Fast Taboo Search Algorithm for the Job Shop Problem, *Management Science*, to appear.
56. W.P.M. Nuijten and E.H.L. Aarts, 1996. A Computational Study of Constraint Satisfaction for Job Shop Scheduling, *European Journal of Operational Research*, to appear.
57. W.P.M. Nuijten and J. Rogerie, 1996. Post on newsgroup *sci.op-research*, January 22.
58. E. Pesch and U.A.W. Tetzlaff, 1996. Constraint Propagation Based Scheduling of Job Shops, *INFORMS Journal on Computing 8*, to appear.
59. B. Roy and B. Sussmann, 1964. Les Problèmes d'Ordonnancement avec Constraints Disjonctives, Note DS No. 9 bis, SEMA, Montrouge.
60. N. Sadeh, 1991. *Look-Ahead Techniques for Micro-Opportunistic Job Shop Scheduling*, PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania.
61. S.V. Sevast'janov, 1994. On Some Geometric Methods in Scheduling Theory: a Survey, *Discrete Applied Mathematics 55*, 59-82.
62. D.B. Shmoys, C. Stein, and J. Wein, 1994. Improved Approximation Algorithms for Shop Scheduling Problems, *SIAM Journal on Computing 23*, 617-632.
63. Y.N. Sotskov, 1991. The Complexity of Shop-Scheduling Problems with Two or Three Jobs, *European Journal of Operational Research 53*, 326-336.

64. R.H. Storer, S.D. Wu, and R. Vaccari, 1992. New Search Spaces for Sequencing Problems with Application to Job Shop Scheduling, *Management Science 38*, 1495-1509.
65. E. Taillard, 1994. Parallel Taboo Search Techniques for the Job Shop Scheduling Problem, *ORSA Journal on Computing 6*, 108-117.
66. F. Tiozzo, 1988. Building a Decision Support System for Operation Scheduling in a Large Industrial Department: a Preliminary Algorithmic Study, Internal report, Department of Mathematics and Informatics, University of Udine, Italy.
67. P.J.M. Van Laarhoven, E.H.L. Aarts, and J.K. Lenstra, 1992. Job Shop Scheduling by Simulated Annealing, *Operations Research 40*, 113-125.
68. D.P. Williamson, L.A. Hall, J.A. Hoogeveen, C.A.J. Hurkens, J.K. Lenstra, S.V. Sevast'janov, and D.B. Shmoys, 1996. Short Shop Schedules, *Operations Research*, to appear.
69. T. Yamada and R. Nakano, 1992. A Genetic Algorithm Applicable to Large-Scale Job-Shop Problems, in *Parallel Problem Solving from Nature, 2*, R. Männer, B. Manderick (eds.), North-Holland, Amsterdam, 281-290.
70. T. Yamada and R. Nakano, 1995. Job-Shop Scheduling by Simulated Annealing Combined with Deterministic Local Search, *Proceedings of the Metaheuristics International Conference*, I.H. Osman, J.P. Kelly (eds.), Hilton Breckenridge, Colorado, 344-349.
71. D.N. Zhou, V. Cherkassky, T.R. Baldwin, and D.E. Olson, 1991. A Neural Network Approach to Job-Shop Scheduling, *IEEE Transactions on Neural Networks 2*, 175-179.