A Fast Taboo Search Algorithm for the Job Shop Problem

Eugeniusz Nowicki • Czeslaw Smutnicki
Technical University of Wrocław, Institute of Engineering Cybernetics, ul. Janiszewskiego 11/17, 50-372 Wrocław, Poland

A fast and easily implementable approximation algorithm for the problem of finding a minimum makespan in a job shop is presented. The algorithm is based on a taboo search technique with a specific neighborhood definition which employs a critical path and blocks of operations notions. Computational experiments (up to 2,000 operations) show that the algorithm not only finds shorter makespans than the best approximation approaches but also runs in shorter time. It solves the well-known 10 × 10 hard benchmark problem within 30 seconds on a personal computer.
(*Scheduling; Heuristics; Job-shop; Taboo Search*)

1. Introduction

The job shop scheduling problem (French 1982) can be briefly described as follows. There are a set of jobs and a set of machines. Each job consists of a sequence of operations, each of which uses one of the machines for a fixed duration. Once started, the operation cannot be interrupted. Each machine can process at most one operation at a time. A schedule is an assignment of operations to time intervals on the machines. The problem is to find a schedule of minimal time to complete all jobs.

The job shop scheduling problem is considered as a particularly hard combinatorial optimization problem (Lawler et al. 1982). Since it has practical applications, the problem has been studied by many authors, and several *optimization algorithms* and *approximation algorithms* have been proposed. The optimization algorithms are based chiefly on the branch and bound scheme (Lageweg et al. 1977, Carlier and Pinson 1989, Brucker et al. 1990, Applegate and Cook 1991). While considerable progress has been made in this approach, practitioners still find such algorithms unattractive. They are time-consuming, and the size of problems which can be solved within a reasonable time limit is small (up to 100 operations). Moreover, their implementation requires a certain level of programmer sophistication. On the other hand, approximation algorithms, which are a quite good alternative, use several various approaches classified as follows: (a) dispatching priority rules (see the review by French 1982); (b) the shifting bottleneck approach (Adams et al. 1988, Applegate and Cook 1990), (c) a geometric approach (see the review by Shmoys et al. 1991), (d) job insertions (Werner and Winkler 1992), (e) a local neighborhood search (Spachis and King 1979), (f) simulated annealing (Matsuo et al. 1988, van Laarhoven et al. 1992), (g) taboo search (Taillard 1989, 1992; Eck and Pinedo 1989), and (h) a genetic approach. Some new research directions are also outlined by Storer et al. (1992). Algorithms (a)–(d) are constructive (create a schedule), while (e)–(g) are iterative (improve a given schedule). Currently the best results have been obtained via algorithms (b), (f), and (g). Usually, the implementation of approximation algorithms is simple; however, some of them are nontrivial (see the algorithm by Adams et al. 1988 and its implementation by Applegate and Cook 1991).

In this paper we propose the use of a very fast and easily implementable algorithm based on a specific neighborhood definition in the taboo search (TS) approach. A substantially small neighborhood is defined using so called blocks of operations on a critical path. We show experimentally (having tested on commonly used benchmarks with up to 2,000 operations) that the

0025-1909/96/4207/0797$01.25

algorithm performs much better than other known approximation algorithms including those based on TS. Moreover, it was successfully run on random benchmarks up to 10,000 operations on a PC. Some modifications of the classical TS technique are also proposed.

2. Problem Definition

There are a set of jobs $J = \{1, \ldots, n\}$, a set of machines $M = \{1, \ldots, m\}$, and a set of operations $O = \{1, \ldots, o\}$. Set O is decomposed into subsets corresponding to the jobs. Job j consists of a sequence of o_j operations indexed consecutively by $(l_{j-1} + 1, \ldots, l_{j-1} + o_j)$, which should be processed in that order, where $l_j = \Sigma_{i=1}^{j} o_i$, is the total number of operations of the first j jobs, $j = 1, \ldots, n$ ($l_0 = 0$), and $\Sigma_{i=1}^{n} o_i = o$. Operation i must be processed on machine $\mu_i \in M$ during an uninterrupted processing time $\tau_i > 0$, $i \in O$. We assume that any successive operations of the same job are going to be processed on different machines.[1] Each machine can process at most one operation at a time. A feasible schedule is defined by start times $S_i \geq 0$, $i \in O$, such that the above constraints are satisfied. The problem is to find a feasible schedule that minimizes the makespan $\max_{i \in O}(S_i + \tau_i)$.

It is convenient for the analysis to represent the problem by using a graph. First, observe that the set of operations O can be naturally decomposed into subsets $M_k = \{i \in O : \mu_i = k\}$, each of them corresponding to operations which should be processed on machine k, and let $m_k = |M_k|$, $k \in M$. The processing order of operations on machine k is defined by permutation $\pi_k = (\pi_k(1), \ldots, \pi_k(m_k))$ on M_k, $k \in M$; $\pi_k(i)$ denotes the element of M_k which is in position i in π_k. Let Π_k be the set of all permutations on M_k. The processing order of operations on machines is defined by m-tuple $\pi = (\pi_1, \ldots, \pi_m)$, where $\pi \in \Pi = \Pi_1 \times \Pi_2 \times \cdots \times \Pi_m$. For the processing order π, we create the digraph $G(\pi) = (O, R \cup E(\pi))$ with a set of nodes O and a set of arcs $R \cup E(\pi)$, where

$$R = \bigcup_{j=1}^{n} \bigcup_{i=1}^{o_j - 1} \{(l_{j-1} + i, l_{j-1} + i + 1)\} \quad \text{and}$$

$$E(\pi) = \bigcup_{k=1}^{m} \bigcup_{i=1}^{m_k - 1} \{(\pi_k(i), \pi_k(i+1))\}.$$

Arcs from set R represent the processing order of operations in jobs, whereas arcs from set $E(\pi)$ represent the processing order of operations on machines. Each node $i \in O$ in the digraph has weight τ_i, and each arc has weight zero. Note that any node in $G(\pi)$ has at most two immediate successors and at most two immediate predecessors. The processing order π is feasible only if graph $G(\pi)$ does not contain a cycle. It is well-known that any feasible processing order π generates a feasible schedule S_i, $i \in O$. Moreover, start time S_i equals the length of the longest path coming to the vertex i (but without τ_i) in $G(\pi)$, $i \in O$; if a node has no predecessors then the length is zero. Makespan $C_{\max}(\pi)$ for the processing order π equals the length of the longest path (critical path) in $G(\pi)$. Now we can rephrase the job shop problem as that of finding a feasible processing order $\pi \in \Pi$ which minimizes $C_{\max}(\pi)$.

Denote a critical path in $G(\pi)$ by $u = (u_1, \ldots, u_w)$, where $u_i \in O$, $1 \leq i \leq w$ and w is the number of nodes in this path. The critical path u depends on π, but for simplicity in notation we will not express it explicitly. The critical path is naturally decomposed into subsequences $B_1, \ldots, B_r$ called blocks in π on u (see among others Grabowski et al. 1986), where:

(1) $B_j = (u_{a_j}, u_{a_j+1}, \ldots, u_{b_j})$, $j = 1, \ldots, r$ and $1 = a_1 \leq b_1 < b_1 + 1 = a_2 \leq b_2 < b_2 + 1 = a_3 \leq \cdots \leq a_r \leq b_r = w$;

(2) B_j contains operations processed on the same machine $\mu(B_j) \stackrel{\text{def}}{=} \mu_{u_{a_j}}$, $j = 1, \ldots, r$;

(3) two consecutive blocks contain operations processed on different machines, i.e., $\mu(B_j) \neq \mu(B_{j+1})$, $j = 1, \ldots, r - 1$.

Simply, one can say that the block is a maximal subsequence of u, which contains operations processed on the same machine. In order to illustrate introduced notions, let us consider the following example.

EXAMPLE. The problem has three jobs, 12 operations, and two machines, $n = 3$, $m = 2$, $o = 12$. Job 1 consists of a sequence of seven operations (1, 2, 3, 4, 5, 6, 7), job 2 consists of a sequence of three operations (8, 9, 10),

[1] This assumption is made for simplicity of description and is not restrictive. Two successive operations of the same job processed on the same machine can be separated by a fictitious operation processed on a fictitious machine, during an infinitesimal processing time.

and job 3 consists of a sequence of two operations (11, 12). Operations must be processed on machines $\mu_1 = \mu_3 = \mu_5 = \mu_7 = \mu_8 = \mu_{10} = \mu_{11} = 1$, $\mu_2 = \mu_4 = \mu_6 = \mu_9 = \mu_{12} = 2$ with processing times $\tau_1 = \tau_3 = \tau_4 = \tau_8 = \tau_9 = \tau_{11} = \tau_{12} = 2$, $\tau_2 = \tau_5 = \tau_6 = \tau_7 = \tau_{10} = 1$. Consider certain feasible processing order $\pi = (\pi_1, \pi_2)$, where $\pi_1 = (1, 8, 3, 11, 5, 10, 7)$ and $\pi_2 = (2, 4, 9, 12, 6)$. The digraph $G(\pi)$ is shown in Figure 1, whereas the Gantt chart in Figure 2. $G(\pi)$ contains the single critical path $u = (1, 8, 3, 4, 9, 12, 6, 7)$, $w = 8$. This path is decomposed into $r = 3$ blocks $B_1 = (1, 8, 3)$, $B_2 = (4, 9, 12, 6)$, $B_3 = (7)$ and $\mu(B_1) = \mu(B_3) = 1$, $\mu(B_2) = 2$, $a_1 = 1$, $b_1 = 3$, $a_2 = 4$, $b_2 = 7$, $a_3 = 8$, $b_3 = 8$.

Figure 1 **The diagraph $G(\pi) = (O, R \cup E(\pi))$ of 3-job, 2-machine, 12-operation instance, where $\pi = (\pi_1, \pi_2)$ $\pi_1 = (1, 8, 3, 11, 5, 10, 7,)$, $\pi_2 = (2, 4, 9, 12, 6)$. Job 1 consists of a sequence of seven operations 1, 2, . . . , 7; job 2 of a sequence of three operations 8, 9, 10; and job 3 of a sequence of two operations 11, 12. Operations 1, 3, 5, 7, 8, 10, and 11 are processed by machine 1, and 2, 4, 6, 9, and 12 by machine 2. Thick arrows denote arcs in R, dotted arrows arcs in $E(\pi)$.**

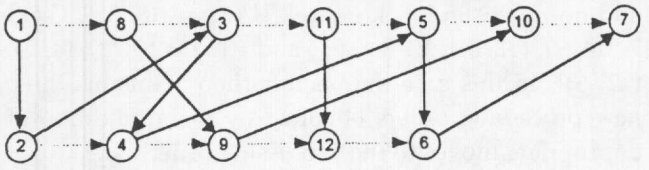
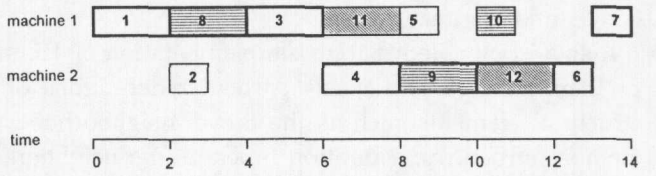


Figure 2 **Gantt Chart. Thick boxes denote operations in blocks $B_1 = (1, 8, 3)$ on machine 1, $B_2 = (4, 9, 12, 6)$ on machine 2, and $B_3 = (7)$ on machine 1.**



3. Application of the Taboo Search Technique to the Job Shop Problem

The taboo search (TS) is a metaheuristic approach designed to find a near-optimal solution of combinatorial optimization problems. This method has been suggested primarily by Glover et al. (1985) and further refined and developed by Glover (1986, 1989, 1990). TS can be briefly sketched as follows. At the beginning a fundamental notion called the *move* is defined. The move is a function which transforms a solution into another solution. For any solution, a subset of moves applicable to it is defined. This subset generates a subset of solutions called the *neighborhood*. TS starts from an *initial solution*. At each step the neighborhood of a given solution is searched in order to find a neighbor (usually the best in the neighborhood). This move, leading us to the best neighbor, is performed and then the newly obtained solution is set as the primal for the next step. In order to prevent cycling and to lead the search to "good" regions of the solution space the search history is kept in the memory and employed in the search. There are at least two classes of the memory: a *short-term memory* for the very recent history and a *long-term memory* for distant history. Among many memory structures established in TS (Glover 1989), a class of the short-term memory called the *taboo list* plays a basic role. This list does not permit to turn back to the solutions visited in the previous maxt steps, where maxt is a given number. In practice the list stores forbidden moves (or their attributes) or attributes of forbidden solutions rather than forbidden solutions. Quite often, the taboo list T is interpreted as a limited queue of length maxt containing forbidden moves; whenever a move from a solution to its neighbor is made we put the inverted move at the end of T and remove the first element from T if the queue is overloaded. Of course, it may happen that an interesting move is taboo. Nevertheless in order to perform such a move, an *aspiration function* must be defined. This function evaluates the profit in taking a forbidden move. If this profit is acceptable, then the taboo status of the move is dropped and the move can be performed. The *stopping rule* can be defined as an instance: (a) we found a solution which is close enough to the given lower bound of the goal function value; (b) we performed maxiter moves without improving the best solution obtained so far, where maxiter is a given number; and (c) the time limit ran out.

Other advanced and more sophisticated elements of TS recommended for hard optimization problems are discussed in details in papers of Glover (1990) and Glover and Laguna (1993).

It should be noted that any implementation of TS is problem-oriented and needs particular definitions of structural elements such as the move, neighborhood, memory structures, aspiration function, the neighborhood searching strategy, stopping rules, the initial solution, and values of several tuning parameters such as maxt, maxiter, and the level of aspiration. The final numerical characteristics of an algorithm (the performance, speed of convergence, and running time) depend both on structural elements and tuning parameters.

Up until now, TS has been applied to several combinatorial optimization problems such as timetabling, routing, and scheduling. TS for the job-shop problem has been discussed by Taillard (1989, 1992) and Eck and Pinedo (1989). In this paper, we propose another TS algorithm for the job-shop. Its structural elements will be defined in subsequent subsections while results of computational tests will be discussed in §4.

3.1. Neighborhood

There are several definitions of a move based mostly on interchanges of adjacent or non-adjacent pairs of operations on a machine. Van Laarhoven et al. (1992) and Taillard (1989) define move v for the processing order π by a pair (x, y) of operations: (i) x and y are successive operations on some machine, and (ii) x and y are successive operations on some critical path in $G(\pi)$. Then they define the neighborhood as processing orders obtained from π by application of all such moves. The size of this neighborhood depends on the number of critical paths in π and the number of operations on each critical path, and it can be relatively large. Obviously the CPU time of any TS implementation depends in principle on the computational complexity of the single neighborhood search (and thus on the neighborhood size). Taillard (1989) has reduced the amount of calculations for a large neighborhood by applying a lower bound on the makespan instead of calculating the makespan explicitly for selecting the best neighbor. Instead of this approach we prefer reducing the neighborhood size (based on neighborhoods defined by Matsuo et al. 1988).

The main idea of the reduction consists in removing some moves from the set introduced by Van Laarhooven et al. (1992), for which it is known a priori (without computing the makespan) that they will not immediately improve $C_{\max}(\pi)$. We will consider a set of moves called as "interchanges near the borderline of blocks on a single critical path." More precisely we consider only a single (arbitrarily selected) critical path u in $G(\pi)$ and blocks $B_1, \ldots, B_r$ defined for u. We swap the first two (and last two) operations in every block $B_2, \ldots, B_{r-1}$, each of which contains at least two operations. In the first block B_1 we swap only the last two operations, and via symmetry in the last block we swap only the first two. Then, formally, we define the set of moves from π as $V(\pi) = \cup_{j=1}^{r} V_j(\pi)$, where

$$V_1(\pi) = \begin{cases} \{(u_{b_1-1}, u_{b_1})\} & \text{if } a_1 < b_1 \text{ and } r > 1, \\ \varnothing & \text{otherwise.} \end{cases}$$

$$V_j(\pi) = \begin{cases} \{(u_{a_j}, u_{a_j+1}), (u_{b_j-1}, u_{b_j})\} & \text{if } a_j < b_j, \\ \varnothing & \text{otherwise.} \end{cases}$$

$$j = 2, \ldots, r-1$$

$$V_r(\pi) = \begin{cases} \{(u_{a_r}, u_{a_r+1})\} & \text{if } a_r < b_r \text{ and } r > 1, \\ \varnothing & \text{otherwise.} \end{cases}$$

The set of moves is not empty only if the number of blocks is greater than one ($r > 1$) and if there exists at least one block with a number of elements greater than one. Let $Q(\pi, v) \in \Pi$ denote the processing order obtained by the application of move v to the processing order π. Neighborhood $H(\pi)$ of π is defined as all processing orders obtained by applying moves from $V(\pi)$, i.e.,

$$H(\pi) = \{Q(\pi, v) : v \in V(\pi)\}.$$

For the example stated in §2 we propose a single move (8, 3) in block B_1, two moves (4, 9), (12, 6) in block B_2, and no moves in block B_3, i.e., $V_1(\pi) = \{(8, 3)\}$, $V_2(\pi) = \{(4, 9), (12, 6)\}$, $V_3(\pi) = \varnothing$ and $V(\pi) = \{(8, 3), (4, 9), (12, 6)\}$. In this case the neighborhood contains three new processing orders obtained by the application of appropriate moves to the processing order π, i.e.,

$H(\pi) = \{Q(\pi, (8, 3)), Q(\pi, (4, 9)), Q(\pi, (12, 6))\}$

where

$Q(\pi, (8, 3)) = ((1, 3, 8, 11, 5, 10, 7), (2, 4, 9, 12, 6))$,

$Q(\pi, (4, 9)) = ((1, 8, 3, 11, 5, 10, 7), (2, 9, 4, 12, 6))$,

and

$Q(\pi, (12, 6)) = ((1, 8, 3, 11, 5, 10, 7), (2, 4, 9, 6, 12))$.

Let us analyze properties of neighborhoods created by interchanges of adjacent operations in the feasible processing order π. Denote by $H''(\pi)$ the set of processing orders each of which were obtained from π by an interchange of a pair of successive operations on a machine. $H''(\pi)$ contains exactly $\Sigma_{k=1}^{m} (m_k - 1)$ processing orders (some of them can be infeasible). Let $H'(\pi)$ be the neighborhood employed by van Laarhoven et al. (see beginning of this section). By definition, $H(\pi)$ is a subset of $H'(\pi)$ and $H'(\pi)$ is a subset of $H''(\pi)$ (i.e., $H(\pi) \subset H'(\pi) \subset H''(\pi)$). Assuming that $G(\pi)$ has only a single critical path u, then $H'(\pi)$ contains $\Sigma_{k=1}^{r} (|B_k| - 1)$ neighbors, whereas $H(\pi)$ contains

$$\min\{1, |B_1| - 1\} + \sum_{k=2}^{r-1} \min\{2, |B_k| - 1\} + \min\{1, |B_r| - 1\}$$

neighbors, where $|B_k| = b_k - a_k + 1$ is the cardinality of kth block. If $G(\pi)$ has several critical paths, then the size of $H'(\pi)$ will increase while the size of $H(\pi)$ will remain the same. One can say that the size of $H(\pi)$ is significantly smaller than that of $H'(\pi)$ if we have several "long" blocks or several critical paths. Moreover, if there is only a single block on u ($r = 1$) then $H'(\pi)$ has at least $|B_1| - 1 = w - 1$ neighbors (if $w \geq 2$), whereas $H(\pi)$ is empty.

It has been shown that for any feasible $\pi^0 \in \Pi$ a finite sequence $\pi^0, \pi^1, \ldots, \pi^k$ exists such that π^k is an optimal processing order and $\pi^{i+1} \in H'(\pi^i)$, $i = 0, \ldots, k - 1$ (connectivity property). Moreover, $H'(\pi)$ contains only feasible processing orders, and for any $\alpha \in H''(\pi) \backslash H'(\pi)$, α is infeasible or $C_{\max}(\alpha) \geq C_{\max}(\pi)$ (van Laarhoven et al. 1992). In other words those processing orders which do not belong to $H'(\pi)$ are "less interesting" from the improvement point of view. Since $H(\pi)$ is a subset of $H'(\pi)$, then it contains only feasible processing orders as well. Now we will show that the processing orders which do not belong to $H(\pi)$ (but can belong to $H'(\pi)$) are also "not promising" for undergoing immediate improvements.

THEOREM. *For any processing order* $\alpha \in H'(\pi) \backslash H(\pi)$, $C_{\max}(\alpha) \geq C_{\max}(\pi)$.

Proof of the theorem is given in Appendix A. The proposed neighborhood $H(\pi)$ allows us to formulate certain sufficient conditions for processing order optimality.

PROPERTY. *If* $V(\pi) = \emptyset$ *then* π *is the optimal processing order*.

Proof of the property is given in Appendix B. Note that $V(\pi) = \emptyset$ if either all operations from u are processed on the same machine or all operations from u belong to the same job. Experimental results from §4 show that the use of the property stops searching in about 20 percent of instances.

Finally, note that the connectivity property does not hold for $H(\pi)$.

3.2. Taboo List

Let $T = (T_1, \ldots, T_{\text{maxt}})$ be the taboo list of a fixed length maxt, where $T_j \in O \times O$ is the forbidden move, $1 \leq j \leq$ maxt. The taboo list is initiated with zero elements $T_j = (0, 0)$, $1 \leq j \leq$ maxt. A move $v = (x, y)$ is added to taboo list T (denoted $T \oplus v$) in the following standard way: we shift taboo list T to the left and put v on the position maxt (i.e., we set $T_j := T_{j+1}, j = 1, \ldots,$ maxt $- 1$ and $T_{\text{maxt}} := v$). Such a form of T is used only for simplicity of notation. In practice, T is implemented as the circular list, which implies that operator $\oplus$ can be performed in $O(1)$ time.

For a move $v = (x, y)$, a move $\bar{v} = (y, x)$ will be called an *inverse move*. If only a move v has been performed from π then the inverse move $\bar{v}$ becomes forbidden and will be added to T, i.e. $T := T \oplus \bar{v}$.

3.3. Neighborhood Searching Strategy

Let π, $V(\pi)$, $H(\pi)$, and C^* be the processing order, set of moves, neighborhood, and the best known makespan. Assume that the set of moves $V(\pi)$ is not empty. We classify these moves into three categories: unforbidden (U), forbidden but profitable (FP), forbidden and

nonprofitable (FN). Moves from set $V(\pi)\backslash T$ are U-moves, whereas those from set $V(\pi) \cap T$ are forbidden. A forbidden move is profitable if it leads to a makespan shorter than C^*. This is, FP-moves are defined by set

$$A = \{v \in V(\pi) \cap T : C_{\max}(Q(\pi, v)) < C^*\}.$$

The remaining moves, i.e., from set $(V(\pi) \cap T)\backslash A$, are FN-moves.

It is quite natural that a move to be performed should be selected among U- and FP-moves. Usually among them, the move yielding the minimal makespan is the one performed (we do exactly the same thing). If there are neither U-moves nor FP-moves, the problem of selection is treated by many authors marginally (e.g., select a FN-move at random or erase the whole T), although it has been discussed in details in the original TS proposals of Glover (1989). Note that our neighborhood is substantially smaller than those defined by other authors. Therefore, the situation in which only FN-moves are available appears more often than in other algorithms. In this context the strategy of selecting an FN-move is important. Moreover, recent studies have confirmed that for hard problems any reasonable TS implementation should contain an appropriate rule for making such a choice, see, i.e., Hubscher and Glover (1992).

In our TS implementation, we propose selecting the "oldest" move (exactly the same as in Glover 1989) and additionally making a special modification of the taboo list based on replications of the "youngest" move. More precisely, if $V(\pi)$ contains a single move then there is no problem of the selection—this move should be chosen. Otherwise, the problem of the selection is solved as follows. First, modify the taboo list: repeat $T := T \oplus T_{\text{maxt}}$ until $V(\pi)\backslash T \neq \emptyset$ (this can be performed in $O(\text{maxt})$ time). Thereafter the set $V(\pi)\backslash T$ contains exactly a single move which can be chosen next. One can say that this modification of T is a form of the dynamic control of a taboo list length. To illustrate, consider the set of moves $V(\pi) = \{(4, 5), (1, 2)\}$ containing only FN-moves, and a taboo list

$$T = \{(2, 3), (1, 2), (6, 5), (4, 5), (2, 7)\}$$

of the length maxt = 5. The proposed rule will select the move $v' = (1, 2)$, and the modified taboo list will take the form of

$$T = \{(6, 5), (4, 5), (2, 7), (2, 7), (2, 7)\}.$$

At the end, the inverse move $\overline{v'} = (2, 1)$ will be added to T, which implies the final form of the taboo list

$$T' = T \oplus \overline{v'} = \{(4, 5), (2, 7), (2, 7), (2, 7), (2, 1)\}.$$

In view of the above considerations we have formulated the following Neighborhood Searching Procedure (NSP). NSP starts with a processing order π, a non-empty $V(\pi)$, a current taboo list T, a best known makespan C^*, and returns move v', the new processing order π' and the modified taboo list T'.

Step 1. Find the set of FP-moves

$$A = \{v \in V(\pi) \cap T : C_{\max}(Q(\pi, v)) < C^*\}.$$

If $(V(\pi)\backslash T) \cup A \neq \emptyset$ then select $v' \in (V(\pi)\backslash T) \cup A$ so that

$$C_{\max}(Q(\pi, v')) = \min \{C_{\max}(Q(\pi, v)) : v \in (V(\pi)\backslash T) \cup A\}$$

and go to 3.

Step 2. If $|V(\pi)| = 1$ then select $v' \in V(\pi)$. Otherwise repeat $T := T \oplus T_{\text{maxt}}$ until $V(\pi)\backslash T \neq \emptyset$. Then select $v' \in V(\pi)\backslash T$.

Step 3. Set $\pi' := Q(\pi, v')$ and $T' := T \oplus \overline{v'}$.

NSP will be used as an element of algorithms presented in the next section.

3.4. Algorithms

It is quite simple to design, using NSP, a classical TS algorithm. Let us start with a given primal processing order π and with a primary empty taboo list. At each iteration we find the set of moves $V(\pi)$. Next, applying NSP we select move $v' \in V(\pi)$, which determines neighbor $\pi' = Q(\pi, v')$, and thereafter we create the new taboo list T'. The processing order π' and taboo list T' are set to be primal for the next iteration. The best found makespan C^* and associated processing order π^* are currently updated. The algorithm stops in either two cases: when the optimal processing order has been found (see property in §3.1) or when the number of iterations without improving the best makespan is greater than a constant maxiter a priori fixed. The above considerations lead us to the following Taboo Search Algorithm (TSA), which begins with the processing order π^* (found by any heuristic algorithm), C^*

$= C_{max}(\pi^*)$, $T = \emptyset$, $\pi = \pi^*$, iter = 0 and returns the best processing order π^* and $C^* = C_{max}(\pi^*)$.

Step 1. Set iter := iter + 1. Find the set of moves $V(\pi)$. If $V(\pi) = \emptyset$ then *STOP*; π^* is optimal.

Step 2. Find the move $v' \in V(\pi)$, the neighbor $\pi' = Q(\pi, v')$ and the modified taboo T' by applying NSP. Set $\pi := \pi'$, $T := T'$.

Step 3. If $C_{max}(\pi) < C^*$ then set $\pi^* := \pi$, $C^* := C_{max}(\pi)$, iter := 0 and go to 1.

Step 4. If iter $\leq$ maxiter then go to 1; otherwise *STOP*.

It is quite obvious that maxiter (limit on unimproved iterations), maxt (length of taboo list), and π^* (primal processing order) have an influence on the efficiency and effectiveness of this scheme. The increasing of maxiter yields a higher possibility of finding a better solution; however, we observe this only in a period of primal values. A further increase of this parameter has little influence on C^* but evidently increases the running time. Moreover, some instances get into the cycle during iterations, which is a well-known disadvantage of TS. As regards maxt, we observe that the best way is to set this value individually for each instance. The primal solution has a week influence on the final C^*. TSA that was started from a "badly" selected primal solution provides a result comparable with that obtained from a "good" one (under slightly increased number of iterations). In this context, the improvement of TSA seems to be possible only by including some long-term memory elements (Glover 1990, Glover and Laguna 1993).

Observe that due to a small neighborhood, TSA works very fast. Therefore one can propose to use TSA as a part of a more complex system which manages searching and leads it to presumable good regions of the solution set. The commonly used technique based on various starting solutions seems to be the best idea. In this order a parameterized heuristic or a list of various heuristics can be used. The essential disadvantage of this approach is the loss of information about previous runs of TSA. Therefore we suggest using the best solution found during the previous run as the new starting solution. Unfortunately this approach (without any modification) loses all history of searching; i.e., we always start with an empty taboo list and a full set of moves. Consequently, we propose the modified (final) version of the taboo search algorithm called Taboo Search Algorithm with Back Jump Tracking (TSAB), which does not have the above mentioned fault. The main idea of TSAB is related to a strategy that resumes the search from unvisited neighbors of solutions previously generated (Glover 1990).

During the run of TSA we store in list L the last maxl best processing orders, where maxl is a fixed number. More precisely, if we have found in some iteration *iter* the processing order π such that $C_{max}(\pi) < C^*$ (see Step 3), we store the triple $(\pi, V(\pi)\backslash v', T)$ in list L, where v' is a move which will be performed from π. Note that $V(\pi)$ and v' will be found in iteration iter + 1 (see Steps 1 and 2). This triple is stored only if $V(\pi)\backslash v' \neq \emptyset$. Just after TSA has been finished we start it again with the processing order, set of moves, and the taboo list located in the last position in list L. In this case TSA is started from Step 2, and the process of storing the best processing orders is continued. TSAB is stopped if list L is empty.

For a precise description of TSAB, we define list L and an additional operator. Let $L = (L_1, \ldots, L_l)$ be the list with current list length l, $l \leq$ maxl where maxl is the upper bound on the list length, and $L_j = (\pi, V, T)$ is a stored triple: a processing order, set of moves, and a taboo list, $1 \leq j \leq l$. The operator for adding the triple (π, V, T) to list L (denotation $L \uplus (\pi, V, T)$) is defined as follows: if $l <$ maxl then we put (π, V, T) in position $l + 1$ in L (i.e., we set $l := l + 1$ and $L_l := (\pi, V, T)$); otherwise we shift list L to the left and put (π, V, T) in the position maxl (i.e., we set $L_j := L_{j+1}$, $j = 1, \ldots,$ maxl $- 1$ and $L_{maxl} = (\pi, V, T)$). Note that this operator has the similar significance as adding an operator on the taboo list.

TSAB begins with the processing order π^* (found by any heuristic algorithm), $\pi = \pi^*$, $C^* = C_{max}(\pi^*)$, $T = \emptyset$, $L = \emptyset$ (i.e., $l = 0$), iter = 0 and save := true. The flag save controls the process of storing on the list L. TSAB returns the best processing order π^* and $C_{max}(\pi^*) = C^*$.

Step 1. Set iter := iter + 1. Find the set of moves $V(\pi)$. If $V(\pi) = \emptyset$ then *STOP*; π^* is optimal.

Step 2. Find the move $v' \in V(\pi)$, the neighbor $\pi' = Q(\pi, v')$, and the modified taboo T' by applying NSP. If save = true and $V(\pi)\backslash v' \neq \emptyset$ then set $L := L \uplus (\pi, V(\pi)\backslash v', T)$.
Set $\pi := \pi'$, $T := T'$ and save := false.

Step 3. If $C_{max}(\pi) < C^*$ then set $\pi^* := \pi$, $C^* := C_{max}(\pi)$, iter := 0, save := true and go to 1.

Step 4. If iter $\leq$ maxiter then go to 1.

Step 5. If $l \neq 0$ then set $(\pi, V(\pi), T) := L_l$, $l := l - 1$, iter := 1, save := true and go to 2; otherwise *STOP*.

Observe that TSAB has no long-term memory for preventing long cycles in a single path search (Steps $1 \cdots 4$, Steps $1 \cdots 4$, $\cdots$). It has been made consciously. By the experimental observation, any transformation which disturbs the real makespan value (e.g., by adding a penalty associated with a move) have "bad" influence on the search process. Therefore instead of preventing, we only detect a cycle, and if it occurs, we end this path. More precisely, Step 4 takes the form "If (iter $\leq$ maxiter) and (IsCykle(iter) = false) then go to 1." Function IsCykle(iter) returns true at iteration iter if there exists the period δ, $1 \leq \delta \leq \max\delta$, such that $C^{i-\delta} = C^i$, $i = \text{iter} + 1 - \max c \cdot \max\delta, \ldots, \text{iter}$, where C^i is the makespan found at iteration i, and $\max\delta$, $\max c$ are given parameters. Otherwise, it returns *false*. In other words, the function detects repetitions of makespans, with a period of length $\delta \leq \max\delta$, observed during the last $\max c \cdot \max\delta$ iterations. It should be noted that IsCykle(iter) can be found in $O(1)$ time. The increase of $\max c$ yields stronger conviction of a cycle existence but simultaneously increases the number of unnecessary performed iterations (if a cycle exists). The increasing of $\max\delta$ allow us to detect longer cycles; however, it increases the storage space required for saving the history. Then $\max c$ and $\max\delta$ are set experimentally as a compromise between the amount of computations and "certainty" of a cycle existence.

Further considerations deal with the computational complexity of algorithm TSAB. The following time-consuming elements are performed in a single iteration of TSAB: finding $V(\pi)$, applying NSP, applying operator $\uplus$ to list L, and storing processing order π^* (these last two elements are performed quite rarely and can be done in $O(o)$ time). Let the computational complexity of NSP be analysed. Graph $G(\alpha)$, where $\alpha = Q(\pi, v)$, can be constructed from $G(\pi)$ in $O(1)$ time, $v \in V(\pi)$. Graph $G(\alpha)$ contains o nodes and $\sum_{j=1}^{n}(o_j - 1) + \sum_{k=1}^{m}(m_k - 1) = 2o - n - m$ arcs. Then makespan $C_{\max}(\alpha)$ can be found in $O(o)$ time by using the standard Bellman's algorithm. Since we have set $\max t \ll o$ in our implementation, NSP can then be performed in $O(ho)$ time, where $h = |V(\pi)|$ is the number of neighbors in neighborhood $H(\pi)$. Just after $C_{\max}(\alpha)$ has been calculated we can find and store the critical path of $G(\alpha)$ in $O(o)$ time. Therefore, passing to Step 1 of TSAB (from Step 3 or 4), we have the critical path already calculated. Next, blocks of operations and set $V(\pi)$ can be found in $O(w)$ time, where $w \ll o$. Finally, the computational complexity of a single iteration of TSAB is $O(ho)$. In our experiments we observed h between 5 and 9 for instances up to 2,000 operations (for small o, $o \leq 200$, h is closer to 5 while for greatest o, $o \geq 500$, h is closer to 9). The number of iterations of TSAB depends on each particular instance, on parameters maxiter and maxl, and is a priori unknown.

3.5. Initial Solution

The initial solution for TSAB can be found by any heuristic method. Further on we present an approximation algorithm employed in our computations. This algorithm is based on an insertion technique which has been primarily applied to flow-shop problems (Nawaz et al. 1983) and then refined and applied to job-shop problems (Werner and Winkler 1992).

We construct the processing order, starting from a primal partial order and inserting in each step a single operation. In more detail, we start from the order containing all operations of the job with the greatest sum of processing times. Next we successively insert operations of remaining jobs in the sequence of nonincreasing operation processing times. Denote by $\sigma = (\sigma_1, \ldots, \sigma_m)$ the partial processing order, where $\sigma_k = (\sigma_k(1), \ldots, \sigma_k(m_k'))$, $m_k' \leq m_k$ is a partial processing order of operations on kth machine, $k = 1, \ldots, m$. For σ we define the graph $G'(\sigma) = (O, R \cup E'(\sigma))$, where

$$E'(\sigma) = \bigcup_{k=1}^{m} \bigcup_{i=1}^{m_k'-1} \{(\sigma_k(i), \sigma_k(i+1))\}.$$

Let $d_i(\sigma)$ denote the length of the longest path passing through vertex i in $G'(\sigma)$. Let x be an operation which is inserted into σ (obviously x is not in σ). Denote by $z = \mu_x$ the machine on which operation x should be processed. Consider set Π_z' of $m_z' + 1$ permutations obtained from σ_z by the insertion of operation x on consecutive positions, i.e., before operation $\sigma_z(1)$ and immediately after operations $\sigma_z(j)$, $j = 1, \ldots, m_z'$. Set Π_z' generates $m_z' + 1$ new partial processing orders on machines having fixed order σ_k on kth machine, $k = 1, \ldots, m$, $k \neq z$, and varied (from Π_z') on zth machine. Since some of

these orders can be infeasible, then let Π' denote the subset containing only feasible orders. It is easy to show that for any feasible σ and x, set Π' is not empty (contains at least one feasible order). For each processing order $\sigma' \in \Pi'$ the longest path $d_x(\sigma')$ passing through vertex x is calculated; the processing order which yields the minimum length is taken as the partial processing order for the next step.

In view of the above considerations we formulate the following INSertion Algorithm (INSA). Without giving up anything of our previous generalization, we can set that the job with the greatest sum of processing times has the index 1. For the sake of simplified notation we assume that the number of operations of each job equals the number of machines and that each job has precisely one operation on each machine (i.e., $o_j = m$, $l_j = jm$, $j \in J$, $m_k = n$, $k \in M$, and $o = nm$).

Step 0. Set $m'_l := 1$ and $\sigma_{\mu_l}(1) := l$ for $l = 1, \ldots, l_1$.

Step 1. Find permutation ω of $\{l_1 + 1, \ldots, o\}$ according to nonincreasing processing times of operations.

Step 2. For $j = 1$ to $o - l_1$ perform Step 3.

Step 3. For partial processing order σ, operation $x = \omega(j)$ and machine $z = \mu_x$ find set Π'_z and set Π'. Find $\sigma' \in \Pi'$ such that $d_x(\sigma') = \min_{\alpha \in \Pi'} d_x(\alpha)$. Set $\sigma := \sigma'$ and $m'_z := m'_z + 1$.

The computational complexity of INSA is $O(n^3m^2)$. The effectiveness of the algorithm is comparable with a combination of 10 straight priority dispatching rules tested by Adams et al. (1988).

4. Computational Results

Algorithm TSAB has been implemented in Pascal and C on a personal computer AT 386DX. In the implementation, we set parameters maxc = 2 and maxδ = 100 for the function IsCycle(iter).

TSAB has been tested in several commonly used instances of various sizes and hardness levels.

(a) Three instances denoted as FS1, FS2, FS3 with $o = 36 \cdots 100$ ($m \times n = 6 \times 6$, 10×10, 5×20) due to Fisher and Thompson (1963), and two instances AD5, AD6 with $o = 100$ (10×10) due to Adams et al. (1988),

(b) Forty instances of eight different sizes denoted as A1$\cdots$I5 with $o = 50 \cdots 300$ (10×10, 10×15, 10×20, 10×30, 5×10, 5×15, 5×20, 15×15) due to Lawrence (1984),

(c) Eighty instances of eight different sizes with $o = 225 \cdots 2{,}000$ (15×15, 15×20, 20×20, 15×30, 20×30, 15×50, 20×50, 20×100) due to Tailard (1993),

(d) Forty instances of four different sizes with $o = 2{,}500 \cdots 10{,}000$ (5×500, 10×500, 5×1000, 10×1000) randomly generated.

Denotations FS1, FS2, FS3, A1$\cdots$I5 are taken from van Larhoven et al. (1992). FS2 is the well-known instance (10 machines 10 jobs) which required almost 25 years before an optimal solution could be reached. Instances FS1, D1$\cdots$5, F1, F5, G1$\cdots$5, H1$\cdots$5 are reported by several authors as "easy," whereas others are "hard." Group (c) contains "particularly hard" cases selected by Taillard (1993) among a large number of randomly generated instances.

We compared our algorithm TSAB with five recently best approximation algorithms: (ABZ) the Shifting Bottleneck Procedure SBII of Adams et al. (1988); (AC) a combination of Bottle5 and Shuffle Procedures of Applegate and Cook (1991); (MSS) a controlled search using the simulated annealing algorithm of Matsuo et al. (1988); (LAL) the classical simulated annealing algorithm of van Laarhoven et al. (1992); and (TA) a taboo search algorithm of Taillard (1989). From the above listed papers we conclude that each algorithm was tested on instances (a)–(b). However, whereas detailed results were included for ABZ, MSS, and LAL; the authors of AC and TA presented only selected results or conclusions. On the other hand Taillard tested TA, AC and ABZ (in the version Bottle implemented by Applegate and Cook) on instances (c). Results of our comparisons will be discussed in detail in the next two subsections.

4.1. Results for Instances (a)–(b)

We compare all algorithms from two points of view: efficiency and effectiveness. Tables 1–3 have been prepared based on the lists of best found makespan and running times reported by the authors of ABZ, MSS and LAL. ABZ and MSS were run on a VAX 780, while LAL was run on a VAX 785. The authors of the LAL algorithm suggest that VAX 785 is two times faster than 780 and therefore adjusted (multiplying by two) the running times of LAL enclosed in Tables 1–3 correspond with those obtained from 780. We found the speed of our PC between the speed of VAX 780 and 785 (about 1.5 times faster than 780 model).

Table 1 **Compact Results for Hard Problem Instances of Fisher and Thompson (1963), Adams et al. (1988), and Lawrence (1984)**

Algorithm	EX	Av CPU [sec]	B: E: W
ABZ	27	549[a]	21: 4: 2
MSS	24	271[a]	13: 9: 2
LAL	25	14,706[a]	15: 8: 2
ABZ + MSS + LAL	27		11: 13: 3
TSAB	27	124[b]	

EX: the number of compared hard problems (FS1 . . . I5 excluding "easy" instances FS1, D1 . . . 5, F1, F5, G1 . . . 5, H1 . . . 5);

Av CPU: the average CPU time; [a] on a VAX-780, [b] on a Personal Computer AT 386DX;

B: E: W: the number of examples for which TSAB found better (B), equal (E) or worse (W) makespan than Algorithm H, $H \in$ {ABZ, MSS, LAL, ABZ + MSS + LAL}.

Parameters maxl, maxt, maxiter, and the heuristic method of finding the initial solution were chosen experimentally in order to ensure a compromise between the running time and solution quality. Finally we tested two versions of our algorithm, denoted by TSAB and TSAB′, where

TSAB: a single run of algorithm TSAB from §3.4 with INSA (used for finding the initial solution), fixed parameters maxl = 5, maxt = 8, and the parameter maxiter varying during the run. We set maxiter = 2,500 at the beginning and if the makespan was improved during the run (Step 3), and we set maxiter = 2,500 − 400∗(maxl − l) if the back jump tracking to the position l on the list L (Step 5) was performed.

TSAB′: the best result selected among three runs of algorithm TSAB from §3.4 with maxl = 5, maxiter = 3,000 (maxiter) = 3,000 − 500∗(maxl − l) for back jump tracking, respectively). There were two runs with INSA and maxt = 8, 10, and a single run with classical SPT rule (as the method of finding the initial solution) and maxt = 8.

The compact results of comparisons TSAB versus ABZ, MSS, LAL and ABZ + MSS + LAL (the best solution of the three) on "hard" instances are given in Table 1. (In "easy" instances all algorithms show excellent performance.) Note that TSAB is better than other algorithms both in CPU time and in the number of the best makespans. Detailed results of comparison are given in Tables 2 and 3. Note that makespan 930 (optimal) of the notorious instance FS2 was found within 30 seconds. TSAB found optimal solutions of 30 instances (from 45) and proved optimality (see property) of 20 (from 30) cases. Moreover, TSAB′ found optimal solu-

Table 2 **Results for Problem Instances of Fisher and Thompson (1963) (FS1: 6 machines 6 jobs, FS2: 10 machines 10 jobs, FS3: 5 machines 20 jobs) and Problem Instances of Adams et al. (1988) (AB5, AB6: 10 machines 10 jobs)**

	Makespan						CPU [sec]				Improvements [%]		
Problem	OPT	ABZ	MSS	LAL[a]	TSAB	TSAB′	ABZ[b]	MSS[b]	LAL[c]	TSAB[d]	ABZ	MSS	LAL
FS1	55	55*	–	55*	55*		1	–	104	0	0	–	0
FS2	930	930*	946	930*	930*		851	988	115,544	30	0	1.69	0
FS3	1,165	1,178	–	1,165*	1,165*		80	–	125,518	3	1.10	–	0
AB5	1,234	1,239	–	–	1,238	1,236	1,503	–	–	4	0.08	–	–
AB6	943	943*	–	–	945	943*	1,101	–	–	29	−0.21	–	–

OPT: optimal value of the makespan (or the best found – in brackets) found by branch-and-bound algorithm of Brucker et al. (1991);

[a] the best makespan found over five runs;

[b] the CPU time on a VAX-780;

[c] the average CPU time over five runs (on a VAX-780);

[d] the CPU time on the personal computer AT 386DX;

* the algorithm found optimal solution without proving optimality;

° the algorithm found optimal solution and proves its optimality;

Improvement: the value 100·(H − TSAB)/H, where $H \in$ {ABZ, MSS, LAL};

TSAB′: the best makespan found over three runs of TSAB.

Table 3 **Results for Problem Instances of Lawrence (1984)[a]**

	Makespan						CPU [sec]				Improvements [%]		
Problem	OPT	ABZ	MSS	LAL[a]	TSAB	TSAB′	ABZ[b]	MSS[b]	LAL[c]	TSAB[d]	ABZ	MSS	LAL
						10 machines, 10 jobs							
A1	945	978	959	956	946	945*	240	156	1,372	64	3.27	1.36	1.05
A2	784	787	784*	785	784*		192	94	1,440	3	0.38	0	0.13
A3	848	859	848*	861	848*		225	106	1,346	66	1.28	0	1.51
A4	842	860	842*	848	842*		240	116	1,660	60	2.09	0	0.71
A5	902	914	907	902*	902*		289	100	1,334	150	1.31	0.55	0
						10 machines, 15 jobs							
B1	(1,050)	1,084	1,071	1,063	1,055	1,047	362	206	3,982	21	2.68	1.49	0.75
B2	927	944	927*	938	954	927*	419	184	4,326	9	−1.06	−2.91	−1.71
B3	1,032	1,032*	1,032*	1,032*	1,032°		225	20	4,186	1	0	0	0
B4	935	976	973	952	948	939	434	200	4,196	184	2.87	2.57	0.42
B5	977	1,017	991	992	988	977*	430	180	4,266	155	2.85	0.30	0.40
						10 machines, 20 jobs							
C1	1,218	1,224	1,218*	1,218*	1,218°		744	54	8,684	16	0.49	0	0
C2	(1,270)	1,291	1,274	1,269	1,259	1,236	837	286	9,070	66	2.48	1.17	0.79
C3	(1,276)	1,250	1,216	1,224	1,216°		901	306	8,708	107	2.72	0	0.65
C4	(1,202)	1,239	1,196	1,218	1,164	1,160	892	268	8,816	493	6.05	2.68	4.43
C5	1,355	1,355*	1,355*	1,355*	1,355°		551	8	7,912	2	0	0	0
						10 machines, 30 jobs							
D1	1,784	1,784*	–	1,784*	1,784°		38	–	3,034	1	0	–	0
D2	1,850	1,850*	–	1,850*	1,850°		29	–	3,504	2	0	–	0
D3	1,719	1,719*	–	1,719*	1,719°		26	–	3,760	0	0	–	0
D4	1,721	1,721*	–	1,721*	1,721°		28	–	3,772	4	0	–	0
D5	1,888	1,888*	–	1,888*	1,888°		22	–	3,336	1	0	–	0
						5 machines, 10 jobs							
F1	666	666*	–	666*	666°		1	–	246	0	0	–	0
F2	655	669	655*	655*	655*		12	4	234	8	2.09	0	0
F3	597	605	597*	606	597*		32	34	258	11	1.32	0	1.49
F4	590	593	590*	590*	593	590*	45	34	242	0	0	−0.51	−0.51
F5	593	593*	–	593*	593°		1	–	236	0	0	–	0
						5 machines, 15 jobs							
G1	926	926*	–	926*	926°		1	–	572	0	0	–	0
G2	890	890*	–	890*	890*		2	–	752	0	0	–	0
G3	863	863*	863*	863*	863°		5	2	584	0	0	0	0
G4	951	951*	–	951*	951°		1	–	566	0	0	–	0
G5	958	958*	–	958*	958°		1	–	486	0	0	–	0
						5 machines, 20 jobs							
H1	1,222	1,222*	–	1,222*	1,222°		2	–	1,254	0	0	–	0
H2	1,039	1,039*	–	1,039*	1,039°		1	–	1,310	0	0	–	0
H3	1,150	1,150*	–	1,150*	1,150°		1	–	1,128	0	0	–	0
H4	1,292	1,292*	–	1,292*	1,292°		1	–	924	0	0	–	0
H5	1,207	1,207*	–	1,207*	1,207°		3	–	1,472	0	0	–	0
						15 machines, 15 jobs							
I1	1,268	1,305	1,292	1,293	1,275	1,268*	735	624	10,692	623	2.30	1.32	1.39
I2	(1,424)	1,423	1,435	1,433	1,422	1,407	837	578	10,574	443	0.07	0.91	0.77
I3	(1,232)	1,255	1,231	1,215	1,209	1,196	1,079	672	10,960	165	3.67	1.79	0.49
I4	1,233	1,273	1,251	1,248	1,235	1,233*	669	660	11,532	325	2.99	1.28	1.04
I5	(1,238)	1,269	1,235	1,234	1,234	1,229	899	616	10,746	322	2.76	0.08	0

[a] The legend for this table appears on the bottom of Table 2.

tions of 7 (out of 15 remaining) cases. For instances C2, C3, C4, I2, I3, I5, which had been unsolved by a branch-and-bound algorithm of Brucker et al. (1991), TSAB found better makespans and for C3 proved optimality. TSAB' also found better makespans for instances B1, C2, C4, I3, which had been unsolved by currently the best branch-and-bound algorithm of Applegate and Cook (1991).

Algorithm AC, which is a combination of modified ABZ (Bottle5) and a post-optimization procedure (Shuffle), generates better makespans than ABZ; however, it requires significantly greater CPU time. Taillard (1989) showed that TA is better than AC (and bottle procedures) with respect to CPU time and makespan. This allowed us to restrict our attention to the performance of TA. Although TA has been tested in instances (a)–(b), only results for FS2, A1$\cdots$A5 and B1, C2, C4, I3 have been published. The number of iterations observed in 15 runs of TA necessary to find the optimal makespan of FS2 were between $2\cdot10^6$ and $35\cdot10^6$ (using only the short-term memory). The best result $2\cdot10^5$ iterations was chosen from among 10 runs of TA (using the short- and long-term memory). The mean makespan obtained after 10^4 iterations was less than 3% above the optimal one. Note that TSAB found an optimal makespan of FS2 within less than $0.75\cdot10^4$ iterations. The performance of TA on instances A1$\cdots$A5 was presented in the form of the mean of makespans as a function of running time on a VAX 785. The mean of the optimal makespans is 864.2 and TA found 873 after 150 sec and 866 after 1000 sec. TSAB found 864.4 after 150 sec on a PC (see Table 3). Finally note that TSAB' found better makespans than TA for C2, C4, I3 and equally good ones for B1 (see Taillard, 1989).

Detailed characteristics of TSAB are provided in Table 4. The reduction of a primal makespan reaches in extreme case 18 per cent, while the number of improvements of a solution during the search is greater than 75. It means that TSAB quickly and efficiently reduces the makespan. The number of iterations necessary to find the best solution is less than $6.5\cdot10^4$. The length of list L (parameter maxl) has an important influence on the final result of "hard" instances. Parameter maxl correlates with maxiter in the sense that a decreasing maxiter should be compensated by an increasing maxl in order to guarantee the same performance of TSAB. Our selection is a compromise reached experimentally.

In our research we also examined other methods of finding an initial solution (among others SPT, LPT, EFT, RANDOM). In each case TSAB provides a makespan comparable with those generated by INSA after a small number (100$\cdots$500) of iterations performed within a few seconds.

In order to isolate the impact of the neighborhood factor on the acceleration of the search we performed an additional test on hard instances (b). We compare TSAB with its analog version working on the neighborhood $H'(\pi)$. For each group of instances A1$\cdots$5, B1$\cdots$5, C1$\cdots$5, and I1$\cdots$5 the performance of the algorithms was presented in the form of the mean of makespans as a function of running time on AT 386DX. At each group the speed of convergence of TSAB with neighborhood $H(\pi)$ is significantly better than that with neighborhood $H'(\pi)$. The superiority increases with increasing n. To illustrate the experiment, the results of the comparison for instances C1$\cdots$5 are shown in Figure 3.

4.2. Results for Instances (c)–(d)

Instances (c) were solved by TA, AC, and ABZ (Taillard 1989). Taillard reports that neither ABZ nor AC can solve instances with $o > 600$ due to running time and the explosion of the size of the search tree. For instances with $o \leq 600$, TA is shown to be better than AC and ABZ with respect to CPU time and the quality of generated makespans. Therefore, we discuss only a comparison of TSAB with TA. As in Taillard (1992), we are chiefly interested in the solution performance not seen on the running time. Therefore, without tuning, we set maxl = 5, maxt = 8, maxiter = 10,000 if the algorithm has improved the makespan (Step 3) and maxiter = 6,000 if back jump tracking (Step 5) has been performed. The primal solution has been found by INSA. The time per single iteration is approximately $4.5\cdot10^{-5}\cdot o$ sec on a PC 386DX. The time per single iteration of TA on a VAX 785 is $16.7\cdot10^{-5}\cdot o$ sec. In comparing computer speeds, we conclude that TSAB performs a single iteration at least four times faster than TA.

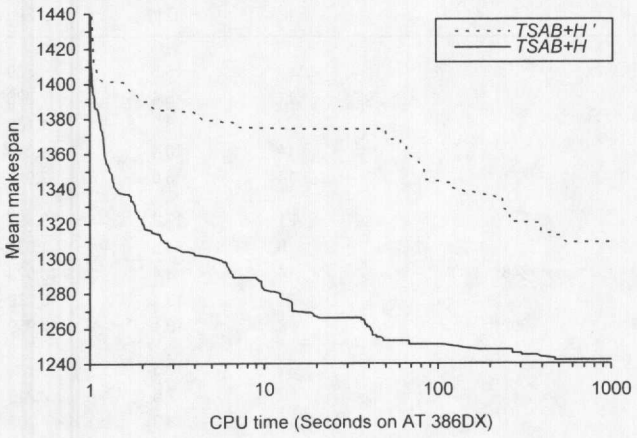
The speed of convergence of TSAB and TA is presented in Table 5. APRD is the average percentage rel-

Table 4 Results of Improvements

Problem	INSA	Makespan by TSAB for maxl						IP	RED [%]	ITER
		0	1	2	3	4	5			
FS1	59	57	55*					3	6.8	580
FS2	994	938	930*					11	6.4	7,326
FS3	1,269	1,165*						22	8.2	1,183
AB5	1,381	1,238						14	10.4	1,019
AB6	1,012	947	945					14	6.6	7,948
A1	1,077	974	956	946				21	12.2	19,695
A2	821	784*						9	4.5	1,031
A3	926	853	853	853	853	848*		14	8.4	18,023
A4	971	850	842*					21	13.4	15,039
A5	1,003	917	907	907	907	902*		28	10.1	37,959
B1	1,179	1,061	1,055					34	10.5	3,732
B2	1,032	968	954					19	7.6	2,246
B3	1,132	1,032*						18	8.8	208
B4	1,021	949	949	949	948			23	7.2	30,001
B5	1,147	1,001	997	995	995	988		48	13.9	27,677
C1	1,397	1,226	1,218*					58	12.8	2,106
C2	1,466	1,268	1,268	1,259				48	14.1	8,971
C3	1,485	1,221	1,216*					72	18.1	16,630
C4	1,385	1,218	1,164					76	16.0	63,469
C5	1,463	1,355*						27	7.4	282
D1	1,966	1,784*						39	9.3	84
D2	1,982	1,850*						33	6.7	135
D3	1,767	1,719*						8	2.7	9
D4	1,844	1,721*						45	6.7	302
D5	1,967	1,888*						18	4.0	48
F1	666*							0	0.0	0
F2	722	658	655*					16	9.3	5,353
F3	681	617	605	605	597*			14	12.3	7,546
F4	659	593						8	10.0	21
F5	593*							0	0.0	0
G1	950	926*						3	2.5	6
G2	976	890*						13	8.8	23
G3	868	863*						1	0.6	1
G4	951*							0	0.0	0
G5	958*							0	0.0	0
H1	1,293	1,222*						8	5.5	11
H2	1,044	1,039*						1	0.5	1
H3	1,154	1,150*						1	0.4	1
H4	1,328	1,292*						1	2.7	1
H5	1,323	1,207*						29	8.8	90
I1	1,445	1,303	1,303	1,279	1,278	1,278	1,275	43	11.8	64,473
I2	1,726	1,450	1,450	1,422				75	17.6	41,183
I3	1,307	1,246	1,209					33	7.5	15,579
I4	1,393	1,274	1,237	1,237	1,235			55	11.3	31,991
I5	1,387	1,261	1,234					50	11.0	30,417

INSA: the initial makespan for TSAB found by INSA (see §3.5);
IP: the number of improvements of makespan in TSAB;
RED: the value 100·(INSA − TSAB)/INSA;
ITER: the number of iterations performing by TSAB (up to the best makespan);
* solution is optimal.

Figure 3 **Performance of TSAB with Neighborhoods H and H' on Instances C1 · · ·5.**



ative difference $100 \cdot (C^H - C^*)/C^*$ over 10 instances of a given size, where $H \in \{\text{TSAB, TA}\}$ and C^* is the reference makespan (the best that was found by Tailard 1993, obtained in 3 or 4 runs of TA with $5 \cdot 10^5 \cdots 10^7$ iterations per run). For our algorithm we scan APRD after performing a given number of iterations (from 10^3 to 10^5). APRD of TA, scanned in some irregular numbers of iterations, is taken from Taillard (1989). Note that TSAB generates in every case faster and better makespans than TA. As an instance, see case 30 jobs 20 machines for which APRD 3.5% is obtained by TA after performing at least $2 \cdot 10^5$ iterations, while TSAB found APRD 3.7% after $2 \cdot 10^4$ and 1.9% after 10^5 iterations. Particularly good results were found for larger problems with $o \geq 1000$. For these instances reference makespans were reached by TSAB in a single run after performing $2 \cdot 10^4$ iterations. Note that, originally, reference makespans were obtained in three or four runs of TA with $5 \cdot 10^5$ iterations per run.

In Table 6 we present instances for which TSAB found better makespans than the reference ones. For this test TSAB was run twice with maxt = 8 and 10, and with maxl, maxiter having the same values as the above. We found 33 better results among 61 instances for which optimality has not been proved. We proved optimality of 10 results (from 33 found) using property (eight cases) and single-machine preemptive lower bound (two cases).

Table 5 **Average Percentage Relative Difference (APRD) Between the Makespan Given by Algorithm H, $H \in$ {TSAB, TA}, and the Reference Makespan [a] for Problem Instances of Taillard (1993)[b]**

	APRD of TSAB for iter								APRD of TA (iter is given in brackets)		
n/m	0	1,000	2,000	5,000	10,000	20,000	50,000	100,000			
15/15	13.9	5.4	3.5	2.8	2.2	1.3	1.1	0.8	5.2 (3,790)	1.7 (42,423)	0.9 (142,989)
20/15	16.5	5.3	4.5	3.9	3.2	2.4	1.4	0.9	7.3 (5,544)	2.6 (64,844)	1.5 (215,935)
20/20	15.8	6.7	5.4	3.6	2.4	1.9	1.4	1.2	6.6 (8,941)	1.6 (93,176)	0.9 (343,927)
30/15	18.8	7.1	5.7	3.6	2.8	1.6	0.7	0.6	11.9 (8,174)	2.1 (105,695)	1.1 (426,762)
30/20	20.3	8.8	7.3	5.6	4.5	3.7	2.5	1.9	13.6 (14,169)	3.5 (201,542)	1.4 (762,446)
50/15	16.4	4.7	3.1	1.2	0.3	0.1	0.0	0.0			
50/20	17.1	5.9	4.1	2.1	0.6	−0.8	−1.5	−2.0			
100/20	13.2	4.9	3.2	1.1	0.4	0.0	−0.1	−0.1			

[a] the best makespans reported by Taillard (1993);
[b] APRD found over 10 instances of the given size;
n/m: the number of jobs/the number of machines;
iter: the number of iterations performed by the algorithm (zero means that only INSA has been run).

Table 6 Best Makespans Found by TSAB for Problem Instances of Taillard (1993)

n/m/k	TA[a]	TSAB[b]	*n/m/k*	TA[a]	TSAB[b]	*n/m/k*	TA[a]	TSAB[b]
15/15/2	1,252	1,244	20/20/6	1,679	1,657	50/20/1	2,921	2,868°
15/15/3	1,223	1,222	20/20/9	1,635	1,629	50/20/2	3,002	2,902
15/15/5	1,234	1,233	30/15/1	1,770	1,766	50/20/3	2,835	2,755°
15/15/8	1,221	1,220	30/15/2	1,853	1,841	50/20/4	2,775	2,702*
15/15/9	1,289	1,282	30/15/3	1,855	1,832	50/20/5	2,800	2,725°
15/15/10	1,261	1,259	30/15/7	1,822	1,815	50/20/6	2,914	2,845°
20/15/2	1,381	1,377	30/15/8	1,714	1,700	50/20/7	2,895	2,841
20/15/4	1,355	1,345	30/15/9	1,824	1,811	50/20/8	2,835	2,784*
20/15/8	1,432	1,413	30/15/10	1,723	1,720	50/20/9	3,097	3,071°
20/15/9	1,361	1,352	30/20/8	2,005	2,001	50/20/10	3,075	2,995°
20/15/10	1,373	1,362	50/15/5	2,689	2,679°	100/20/10	5,213	5,183°

n/m/k: the number of jobs/the number of machines/the index of instance;
[a] the best makespan found over 3–4 runs for various initial solutions;
[b] the best makespan found over 2 runs with maxt = 8, 10 (the same initial solution);
* the algorithm found optimal solution without proving optimality;
° the algorithm found optimal solution and proves its optimality.

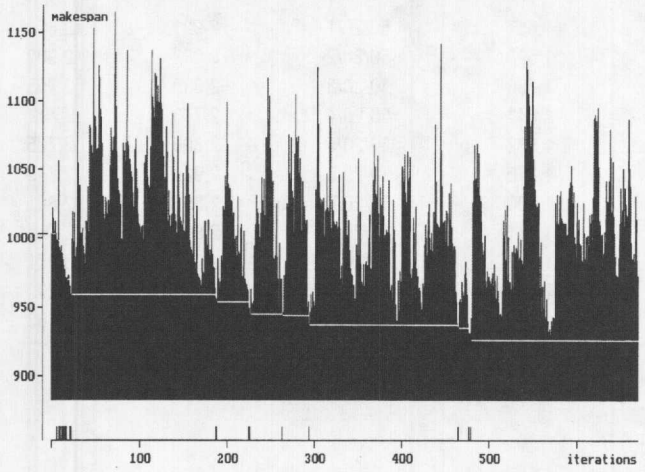
Another set of large-size instances (d) were generated similarly as in Taillard (1993). TSAB were tuned in the same way as in §4.1 (with SPT instead of INSA), and required approximately $9 \cdot 10^{-5} \cdot o$ sec per single iteration on a PC (due to large-memory implementation). All tested instances were solved optimally. The mean number of iteration necessary to find the optimal makespan is 30 for the instances 5×500, 50 for $5 \times 1{,}000$, about $2 \cdot 10^3$ for instances 10×500, and about $7.5 \cdot 10^3$ for those $10 \times 1{,}000$.

5. Conclusions

We have presented a method based on a taboo search technique for minimizing the maximum makespan in the job shop scheduling problem. Due to the special structure of the neighborhood, the method works faster and more efficiently than other known algorithms, even though the connectivity property does not hold for the applied neighborhood. This makes it possible to solve optimally, or almost optimally, medium and large-size job shop problems on a Personal Computer in seconds or at most a few minutes. The proposed method is easily implementable and has several parameters which can "tune" the method to the type of problem instances (primal values of tuning parameters are recommended). The approach can be easily extended to other scheduling problems (such as flow shop and open shop) with the minimum value of the maximum penalty criterion.

Seen in the broader context, the proposed method differs in some elements from the original TS proposals. First, the method does not keep track of the quality of *all* unselected neighbors, but employs only a *subset* of such neighbors. Second, solutions recorded in the list L (*elite solutions*) are selected in a specific way, which theoretically allows us to collect solutions relatively close to each other, and located near a single local optimum; this suggest that some region(s) of the solution space could be searched very intensively whereas other, probably more worthwhile, could be disregarded. Indeed, in the experiments we observed in L some sequences of succesive solutions, Figure 4, however, in the stable phase of the search they had at most three elements (hence maxl we set to 5). Moreover, our method restarted from a close solution usually performed quite different path search and tended to a "good" distant region. A version of TSAB such that only distant best solutions were taken to the list L (the distance were measured by the number of iterations) provided generally worse results, although an improvement was observed for some instances. On the other hand, the

Figure 4 **TSAB, first 600 iterations on instance A5. At iteration 600 the list L contains solutions found at iterations 263, 294, 465, 477, 479 (see markers on the horizontal axis).**



problem of selection of elite solutions to L should be considered in the context of our method behavior in a single path search. In all tested instances, the makespan for successive solutions is similar, and this is rather unusual for TS; see Figure 4. Changes are sharp, with high amplitude, and quite often the makespan exceeds the initial value (1003 in this figure). There are no clear differences between the phase of best makespan decreasing exponentially and the stable phase. Taking all arguments into account, it seems that there are no reasonable criteria for creating a selection which would be good enough for all instances. It means that any proposed criterion should be verified experimentally. Based on our experience, we proposed such criteria, tested them, and obtained quite promising results. Nevertheless, it remains an open space for further research.[2]

[2] The authors wish to thank the anonymous referees for their comments. Special thanks are addressed to Fred Glover for his valuable suggestions referring to TS technique. This research was partially supported by the International Institute of Applied System Analysis, Laxenburg, Austria.

Appendix A

Proof of Theorem

Let $\pi \in \Pi$ be a feasible processing order and $u = (u_1, \ldots, u_w)$ be the selected critical path in $G(\pi)$, which generates blocks $B_1, \ldots, B_r$ and the neighborhood $H(\pi)$. Set $U = \{u_1, \ldots, u_w\}$. Consider $\alpha \in H'(\pi) \backslash H(\pi)$. Denote by (x, y) the pair of operations in π, which have been interchanged in order to obtain α, i.e., $\alpha = Q(\pi, (x, y))$. Then we have

(A1) *x and y are successive operations on some machine q in π_q* ($q = \mu_x = \mu_y$), *which means that* $(x, y) \in E(\pi)$, *and*

(A2) *x and y are successive operations on critical path* $u' = (u'_1, \ldots, u'_{w'})$ *in* $G(\pi)$.

Generally u' can be different from u. It will be shown that in $G(\alpha)$ there exists a path z not shorter than $\sum_{i=1}^{w} \tau_{u_i} = \sum_{i=1}^{w'} \tau_{u'_i} \overset{\text{def}}{=} \tau$ which means that $C_{\max}(\alpha) \geq \tau = C_{\max}(\pi)$. In order to prove this consider separately four cases: "$x \notin U, y \notin U$," "$x \in U, y \notin U$," "$x \notin U, y \in U$," and "$x \in U, y \in U$."

Case "$x \notin U, y \notin U$." The interchange of operations x, y preserves arcs $(u_1, u_2), (u_2, u_3), \ldots, (u_{w-1}, u_w)$ in $G(\alpha)$. Hence graph $G(\alpha)$ contains path $z = u$ with the length τ. (The path z need not be the critical path of this graph.)

Case "$x \in U, y \notin U$." Let $x = u_k$ for certain k, $1 \leq k \leq w$. If $k = w$ then by (A1) the graph $G(\pi)$ contains a path (u, y) longer than τ, which means that u is not the critical path (contradiction). Therefore it must be that $k < w$. Next, observe that $y \neq u_{k+1}$ (since $y \notin U$), $(u_k, y) \in E(\pi)$ (by A1), and $(u_k, u_{k+1}) \in E(\pi) \cup R$. Then we have $(u_k, u_{k+1}) \in R$. Now, assume that $k > 1$ and consider the following two subcases: "$\mu_{u_{k-1}} \neq \mu_{u_k}$" and "$\mu_{u_{k-1}} = \mu_{u_k}$."

(i) "$\mu_{u_{k-1}} \neq \mu_{u_k}$." By the definition of $G(\pi)$ it follows that $(u_{k-1}, u_k) \in R$. Hence, graph $G(\alpha)$ contains path $z = u$ with the length τ.

(ii) "$\mu_{u_{k-1}} = \mu_{u_k}$." Since $(u_{k-1}, y) \in E(\alpha)$ and $(y, u_k) \in E(\alpha)$ then graph $G(\alpha)$ contains path $z = (u_1, \ldots, u_{k-1}, y, u_k, \ldots, u_w)$ with the length $\tau + \tau_y$.

If $k = 1$ then $(y, u_1) \in E(\alpha)$ and graph $G(\alpha)$ contains path $z = (y, u)$ with the length $\tau + \tau_y$.

Case "$x \notin U, y \in U$." The proof can be done by symmetry to the previous case.

Case "$x \in U, y \in U$." Let $x = u_k$ for certain k, $1 \leq k \leq w$ and $y = u_l$ for certain $l \neq k$, $1 \leq l \leq w$. Note that $l = k + 1$. Consider block B_e, $1 \leq e \leq r$, such that $x \in B_e$ and $y \in B_e$ (by the definition of blocks, e exists). If we have only one block (i.e., $e = r = 1$ and $u = B_1$) then graph $G(\alpha)$ contains path $z = (u_1, \ldots, u_{k-1}, y = u_{k+1}, x = u_k, u_{k+2}, \ldots, u_w)$ of length τ. Otherwise, consider the following three subcases: "$1 < e < r$," "$e = 1$" and "$e = r$."

(i) "$1 < e < r$." Since $\alpha \notin H(\pi)$ then $(x, y) \notin V_e(\pi)$. Hence $a_e < k$ and $k + 1 < b_e$. Therefore graph $G(\alpha)$ contains path

$$z = (B_1, B_2, \ldots, B_{e-1}, u_{a_e}, u_{a_e+1}, \ldots, u_{k-1}, y = u_{k+1}, x = u_k, u_{k+2}, \ldots, u_{b_e}, B_{e+1}, \ldots, B_r)$$

of length τ.

(ii) "$e = 1$." Since $\alpha \notin H(\pi)$ then $(x, y) \notin V_1(\pi)$. Hence $k + 1 < b_1$. Therefore graph $G(\alpha)$ contains path

$$z = (u_{a_1}, u_{a_1+1}, \ldots, u_{k-1}, y = u_{k+1}, x = u_k, u_{k+2}, \ldots, u_{b_1}, B_2, \ldots, B_r)$$

of length τ.

(iii) "$e = r$." The proof can be done by symmetry to (ii). □

Appendix B

Proof of Property

By definition $V(\pi)$ can be empty in one of two following cases: "$a_j = b_j, j = 1, \ldots, r$" or "$r = 1$."

Case "$a_j = b_j, j = 1, \ldots, r$." It means that $(u_j, u_{j+1}) \in R, j = 1, \ldots, w - 1$, and therefore all operations from u are operations of the same job k (i.e., $w = o_k$ and $u = (u_1, \ldots, u_w) = (l_{k-1} + 1, \ldots, l_{k-1} + o_k)$). Hence, we obtain that $C_{\max}(\pi)$ is equal to the job-based bound $\sum_{i=l_{k-1}+1}^{l_{k-1}+o_k} \tau_i$.

Case "$r = 1$." It means that all operations from $u = B_1$ are processed on the same machine $q = \mu(B_1)$ (i.e., $\{u_1, \ldots, u_w\} = M_q$). Therefore $C_{\max}(\pi)$ is equal to the machine-based bound $\sum_{i \in M_q} \tau_i$. □

References

Adams, J., E. Balas, and D. Zawack, "The Shifting Bottleneck Procedure for Job Shop Scheduling," *Management Sci.*, 34, 3 (1988), 391–401.

Applegate D. and W. Cook, "A Computational Study of the Job-Shop Scheduling Problem," *ORSA J. Comput.*, 3, 2 (1991), 149–156.

Brucker P., B. Jurish, and B. Sieviers, "A Fast Branch & Bound Algorithm for the Job-Shop Problem," Report, Universität Osnabrück, Germany, 1991.

——, and B. Jurish, "Job-shop (C codes)," *European J. Oper. Res.*, 57 (1992), 132–133.

Carlier J. and E. Pinson, "An Algorithm for Solving the Job-Shop Problem," *Management Sci.*, 35, 2 (1989), 164–176.

Eck B. and M. Pinedo, "Good Solution to Job Scheduling Problems Via Tabu Search," Presented at Joint ORSA/TIMS Meeting, Vancouver, Canada, May 10, 1989.

Fisher H. and G. L. Thompson, "Probabilistic Learning Combinations of Local Job-Shop Scheduling Rules," in J. F. Muth and G. L. Thompson (Eds.), *Industrial Scheduling*, Prentice-Hall, Englewood, Chichester, UK, 1963.

French S., *Sequencing and Scheduling: An Introduction to the Mathematics of the Job-Shop*, Horwood, Chichester, UK, 1982.

Glover F., C. McMilan, and B. Novick, "Interactive Decision Software and Computer Graphics for Architectural and Space Planning," *Annals of Oper. Res.*, 5, (1985), 557–573.

——, "Future Path for Integer Programming and Links to Artificial Intelligence," *Computers and Oper. Res.*, 13, 5 (1986), 533–549.

——, "Tabu Search—Part I," *ORSA J. Computing*, 1, 3 (1989), 190–206.

——, "Tabu Search—Part II," *ORSA J. Computing*, 2, 1 (1990), 4–32.

——, and M. Laguna, "Tabu Search," in C. Reeves (Ed.), *Modern Heuristic Techniques for Combinatorial Problems*, Blackwell Scientific Publishing, 1993.

——, E. Taillard, and D. de Werra, "A Users Guide to Tabu Search," *Annals of Oper. Res.*, 41 (1993), 3–28.

Grabowski J., E. Nowicki, and S. Zdrzalka, "A Block Approach for Single Machine Scheduling with Release Dates and Due Dates," *European J. Oper. Res.*, 26 (1986), 278–285.

Hubscher R. and F. Glover, "Applying Tabu Search with Influential Diversification to Multiprocessor Scheduling," Graduate School of Business, University of Colorado, Denver, CO, 1992.

Van Laarhoven P. J. M., E. H. L. Aarts, and J. K. Lenstra, "Job Shop Scheduling by Simulated Annealing," *Oper. Res.*, 40, 1 (1992), 113–125.

Lageweg B. J., J. K. Lenstra, and A. H. G. Rinnooy Kan, "Job-Shop Scheduling by Implicit Enumeration," *Management Sci.*, 24 (1977), 441–450.

Lawler E. L., J. K. Lenstra, and A. H. G. Rinnooy Kan, "Recent Developments in Deterministic Sequencing and Scheduling: A Survey," in M. A. H. Dempster, J. K. Lenstra, and A. H. G. Rinnooy Kan (Eds.), *Deterministic and Stochastic Scheduling*, Reichel, Dordrecht, The Netherlands, 1982.

Lawrence S., "Resource Constrained Project Scheduling: An Experimental Investigation of Heuristic Scheduling Techniques (Supplement)," Graduate School of Industrial Administration, Carnegie Mellon University, Pittsburgh, PA, 1984.

Matsuo H. C., C. J. Suh, and R. S. Sullivan, "A Controlled Search Simulated Annealing Method for the General Jobshop Scheduling Problem," Working Paper 03-04-88, Department of Management, The University of Texas at Austin, Austin, TX, 1988.

Nawaz M., E. E. Enscore, JR., and I. Ham, "A Heuristic Algorithm for the m-machine, n-job Flow-shop Sequencing Problem," *OMEGA International J. Management Sci.*, 11, 1 (1983), 91–95.

Shmoys D. B., C. Stein, and J. Wein, "Improved Approximation Algorithms for Shop Scheduling Problems," *Proc. Second ACM-SIAM Symposium on Discrete Algorithms*, January, 1991.

Spachis A. S. and J. R. King, "Job-Shop Scheduling Heuristics with Local Neighborhood Search," *International J. Production Res.*, 17, 6 (1979), 507–526.

Storer R. H., S. David Wu, and R. Vaccari, "New Search Spaces for Sequencing Problems with Application to Job Shop Scheduling," *Management Sci.*, 10, 10 (1992), 1495–1509.

Taillard E., "Parallel Taboo Search Technique for the Jobshop Scheduling Problem," Working Paper ORWP 89/11 (revised version October 1992), Departement de Mathematiques, Ecole Polytechnique Federale De Lausanne, Lausanne, Switzerland, 1989.

——, "Benchmarks for basic scheduling problems," *European J. Oper. Res.*, 64 (1993), 278–285.

Werner F. and A. Winkler, "Insertion Techniques for the Heuristic Solution of the Job Shop Problem," Report, Technische Universität "Otto von Guericke," Magdeburg, Germany, 1992.

Accepted by Luk Van Wassenhove; received May 13, 1993. *This paper has been with the authors* 3 *months for* 1 *revision.*