ORSA Journal on Computing
Vol. 6, No. 2, Spring 1994

0899-1499/94/0602-0108 $01.25

Parallel Taboo Search Techniques for the Job Shop Scheduling Problem

ÉRIC D. TAILLARD / *DMA, École Polytechnique Fédérale de Lausanne, 1015 Lausanne, Switzerland; Email: taillard@dma.epfl.ch*

(Received: October 1991; revised: July 1992; accepted: December 1993)

We apply the global optimization technique called taboo search to the job shop scheduling problem and show that our method is typically more efficient than the shifting bottleneck procedure, and also more efficient than a recently proposed simulated annealing implementation. We also identify a type of problem for which taboo search provides an optimal solution in a polynomial mean time in practice, while an implementation of the shifting bottleneck procedure seems to take an exponential amount of computation time. Included are computational results that establish new best solutions for a number of benchmark problems from the literature. Finally, we give a fast parallel algorithm that provides good solutions to very large problems in a very short computation time.

The job shop scheduling problem, or $J \| C_{max}$ in the classification of Lawler et al.,[9] has been studied for a long time. Recent works include those of Carlier and Pinson,[4] Adams et al.,[1] van Laarhoven et al.,[8] and Applegate and Cook,[2] who propose, respectively, a branch and bound method, the shifting bottleneck procedure, an adaptation of simulated annealing (SA), and some extensions of the shifting bottleneck procedure for solving this problem.

The form of the problem may be roughly sketched as follows: we are given n jobs, each composed of several operations, that must be processed on m machines. Each operation uses one of the m machines for a fixed duration. The operations of a given job have to be processed in a given order. The problem is to find a schedule of the operations on the machines, taking the precedence constraints into account, that minimizes the make span (C_{max}), that is, the finish time of the last operation completed in the schedule. This problem was shown to be NP-hard by Lawler et al.[9] The difficulty of this problem may be illustrated by the fact that the optimal solution of an instance with 10 jobs and 10 machines, proposed by Fisher and Thompson,[5] was not found until 20 years after the problem was introduced.

Branch and bound methods have the advantage of providing an optimal solution to this problem. Unfortunately, the computing time becomes prohibitive when the number of operations exceeds a few hundred. By contrast, the method proposed by Applegate and Cook[2] is a faster heuristic method that provides good solutions, but without guaranteeing their optimality. Finally, the SA implementation of van Laarhoven et al.[8] provides better solutions, but the computing times are much higher. Its advantage is its very simple implementation.

As a foundation for our comparative study using taboo search, we especially thank D. Applegate and B. Cook for making available their implementation of the shifting bottleneck procedure (referred as *bottle* from now on) and those of the extensions of this procedure (*bottle-4*, *bottle-5*, and *shuffle* procedures). These extensions, discussed in the paper of Applegate and Cook,[2] provide better solutions than the original method but need much higher computation times.

This paper, to our knowledge, represents the first effort to apply the taboo search technique (TS) to the job shop problem. Our implementation investigates some new features of TS including random taboo list length and frequency-based memory. Section 1 gives the representation of the job shop problem in graph theoretical terms. Then, section 2 presents an implementation of TS for this problem. We discuss the effectiveness and the efficiency of this method in section 3. In section 4, we give first some ways of parallelizing TS and then give a general method for parallelizing randomized algorithms that is relevant for applying TS to the job shop problem. We discuss CPU times required by our method in section 5 and provide conclusions in section 6.

1. Representation of the Problem in Terms of Graph Theory

It is useful to represent the job shop problem in graph theoretical terms by creating a vertex for each operation. We number the vertices from 1 to N, where N denotes the total number of operations. Two fictitious vertices, 0 (beginning) and $N + 1$ (end), are then added. An arc (i,j) connects operations i and j if they belong to the same job, and i has to be processed immediately before j. The fictitious operation 0 is connected by an arc of length 0 to the initial operation of every job; the final operation of every job is connected to operation $N + 1$. The length of arc (i,j) is the duration of operation i. These arcs represent a set of conjunctive constraints.

All operations that have to be processed by the same machine are fully connected together by edges (disjunctive constraints). Figure 1a illustrates the case of a 3-machine job shop with 3 jobs composed of 3, 2, and 2 operations.

Subject classifications: Scheduling, taboo search technique, complexity, parallel algorithms.

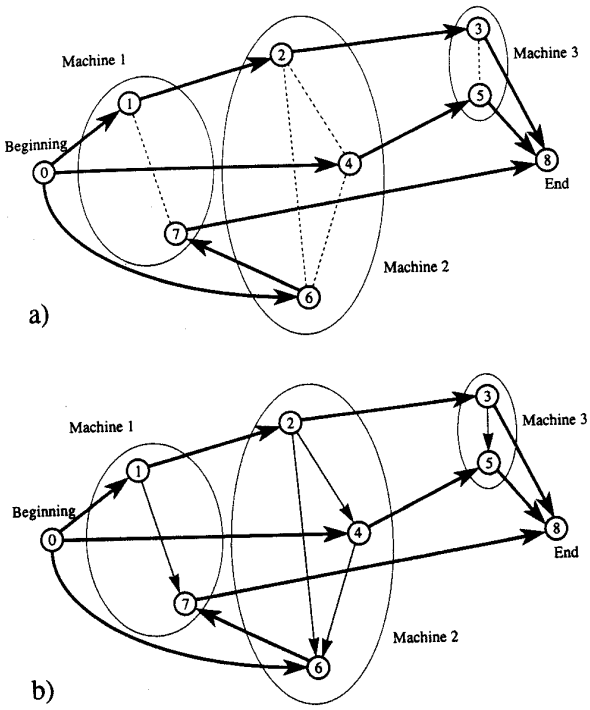

Figure 1. a) Example of a 3-machine, 3-job problem; b) feasible schedule of this problem.

The problem consists in orienting these edges and giving them a length corresponding to the duration of the operation corresponding to their origin. This orientation has to be chosen in such a way that the length of a longest path from 0 to $N + 1$ is minimized. A feasible solution corresponds to an orientation of the edges of the graph without oriented cycle. Figure 1b illustrates a feasible schedule of the example of Figure 1a. The length of a longest path from 0 to $N + 1$ corresponds to the make span; an operation is said to be *critical* if it belongs to a longest path.

The problem consists now in finding a longest path in a graph. If one removes the redundant disjunctive arcs, every operation (except the fictitious ones) has exactly 2 successors and 2 predecessors. We introduce the following additional notations. For every operation i ($1 \leqslant i \leqslant N$):

- J_i Job to which i belongs.
- M_i Machine which processes i.
- d_i Processing time of i.
- PJ_i (If it exists) operation belonging to job J_i that precedes i.
- SJ_i (If it exists) operation belonging to job J_i that follows i.
- PM_i (If it exists) operation processed on machine M_i just before i.
- SM_i (If it exists) operation processed on machine M_i just after i.
- r_i (Release date) earliest beginning date of operation i.
- q_i (Length of the tail), length of a longest path from i to $N + 1$.

An operation is critical if and only if:

$$r_i + q_i = C_{max}$$

For every operation i we have the following identities (where we assume that d_i, r_i, and $q_i = 0$ for the undefined indices i):

$$r_i = \max(r_{PM_i} + d_{PM_i}, r_{PJ_i} + d_{PJ_i})$$

$$q_i = \max(q_{SM_i}, q_{SJ_i}) + d_i$$

We now present an algorithm that computes in time $O(N)$ all values of r_i, using a variation of the labeling algorithm of Bellman, adapted to our special graph. (The computation of the q_i values is similar.)

Algorithm to Calculate r_s's

0) $\forall i$ such that PM_i and PJ_i are undefined, introduce i into Q, a set of operations for which r_i is calculable.
1) Repeat, until $Q = \varnothing$:
   2) Label $i \in Q$.
   $Q \leftarrow Q \setminus \{i\}$.
   Calculate r_i.
   if PM_{SJ_i} is labeled or not defined then $Q \leftarrow Q \cup \{SJ_i\}$.
   if PJ_{SM_i} is labeled or not defined then $Q \leftarrow Q \cup \{SM_i\}$.

2. Implementation of Taboo Search

Briefly stated, TS is a global iterative optimization method: the search moves from one solution to another, in order to improve the quality of the solutions visited. This supposes a *neighborhood structure*. When the search arrives at a local optimum, it does not terminate but moves beyond the local optimum by choosing the best possible (allowed) neighbor. In order to avoid cycling, the move that leads back to the local optimum just left is forbidden. This is accomplished in a *short-term memory* framework by keeping the forbidden (taboo) moves in a structure called *taboo list*. For a taboo list of a given size, when an element is added to the taboo list, another (the oldest one) is removed. The size of the taboo list must be large enough to avoid cycling, but small enough not to forbid too many moves. However, it may happen that an interesting move (such as a move that improves the best solution already found) is taboo. In order to nevertheless perform such moves, an *aspiration level* is defined (depending, for example, on the current solution and the best solution found). Further, in order to diversify the search, a *long-term memory* mechanism is implemented.

Taboo search has been applied to many scheduling problems, as related in Barnes and Laguna.[3] For additional features of the method, as applied in a variety of combinatorial optimization settings, the reader may refer to the papers of Glover.[6,7]

We now describe the structures we have used in our implementation of TS:

NEIGHBORHOOD:
A neighboring solution in our approach is a solution obtained by permuting two successive and critical operations that use the same machine. Van Laarhoven et al.[8] have shown the following properties of this neighborhood:

- Starting with any feasible solution, the new one is feasible too.
- Starting with any feasible solution, it is possible to reach an optimal solution.

The permutation of non-critical operations, by contrast, cannot improve the objective function and may create a directed cycle in the graph, i.e., an infeasible solution. The evaluation of the make span of neighboring solutions may be done very quickly (in time $O(N)$). If critical operations a and b ($b = SM_a$) are permuted, the new values of changed parameters may be calculated as follows:

$$r'_b = \max(r_{PM_a} + d_{PM_a}, r_{PJ_b} + d_{PJ_b}).$$
$$r'_a = \max(r'_b + d_b, r_{PJ_a} + d_{PJ_a}).$$
$$q'_a = \max(q_{SM_b}, q_{SJ_a}) + d_a.$$
$$q'_b = \max(q'_a, q_{SJ_b}) + d_b.$$

Then the value $C'_{max} = \max(r'_b + q'_b, r'_a + q'_a)$ gives the value of the new longest path if it passes through a or b, or a lower bound on the new value of the make span. Each step of TS consists of examining the entire neighborhood (complexity: $O(N)$) and of choosing the best allowed neighbor (i.e., non-taboo or accepted with regard to the aspiration level).

For creating an initial solution, we schedule one job after the other, placing the successive operations of a job at the first possible place in the partial schedule. So, the complexity of creating an initial solution is $O(N^2)$. As the jobs are chosen in index order (essentially random) for creating the initial solution, the resulting initial solution is generally very bad. However, a taboo search that is properly implemented is characteristically able to find good solutions, whatever the initial solution is. Starting with a better initial solution normally only affects the search in the short term, but not over long term.

TABOO LIST:
Let k be the number of the iteration (i.e., the number of moves already executed) at the point where operations a and $b = SM_a$ are permuted. Changing the current successor of b (on a machine) is now forbidden if the number of the current iteration is lower than $k + L$; L is a value called *length of the taboo list*, which strongly depends on the number of jobs and machines. For (hard) problems with about as many jobs as machines, we have found in preliminary experiments that the taboo list length has to be set to about $(n + m)/2$ for getting good results; conversely, for (simple) problems with $n \gg m$, a taboo list length of $N/2$ provides good solutions. The transition between hard and simple problems occurs (for random generated problems, see next section) when the number of jobs becomes greater than 4 to 5 times the number of machines. So we propose setting:

$$L = (n + m/2) \cdot e^{-n/5m} + N/2 \cdot e^{-5m/n}.$$

Such a taboo list length seems to be convenient if $n \geqslant m$, but we have not performed extensive tests for problems with $m > n$.

The taboo list is implemented as an integer vector containing N iteration numbers (each set to $-\infty$ initially) which creates a memory for each operation of the last time it was swapped to become the new predecessor of another operation. One also could use a list that forbids a to have b as successor, for every possible a and b. However, such a list may use a huge memory space if N is large.

In order to avoid cycling phenomena more effectively, the size of the list is randomly and uniformly chosen between two extreme values L_{min} and L_{max} and changed each time a number of iterations slightly greater (for example 5%) than its maximal length has been performed. $L_{min} = \lfloor 0.8L \rfloor$ and $L_{max} = \lceil 1.2L \rceil$ are convenient values for the considered problems. Dynamic taboo lists have been suggested by Glover,[7] and we have used a list similar to the preceding in an adaptation to flow shop sequencing,[13] but this is the first time to our knowledge that the foregoing random policy has been used.

ASPIRATION LEVEL:
The type of taboo list we have chosen may forbid interesting moves. Thus, a move will be performed despite being taboo if the length of the new longest path from 0 to $N + 1$ passing through the permuted operations is shorter than the value of the best solution found up to the current iteration.

LONG-TERM MEMORY:
For problems that need a great number of iterations, a mechanism that prevents the repetition of the same (small) exchanges over long term is useful. The mechanism we choose is the following: we store for each operation the number of times it was pushed earlier in the schedule in order to compute the frequency at which each operation is pushed earlier.

The more frequently an operation is pushed back, the more we will penalize a move that pushes the operation back again in the future: in the evaluation of the neighborhood, we add the quantity $P \cdot f(b)$ to the value of a move that swaps operations a and $b = SM_a$, where P is a new parameter of our method and $f(b)$ is the frequency at which b has been pushed back. The value given to P depends strongly on the problem size and on the problem instance. If we call Δ_k^{max} the maximal increase of the objective function between two successive solutions met by the search till iteration k, it turns out that $P = 0.5 \cdot \Delta_k^{max} \cdot \sqrt{N}$ provides good solutions for every problem tested. Intuitively, the formula that gives the value of P is "justified" as follows: the penalty should be proportional to the value

of the moves (this explains the factor Δ_k^{max}); as the number of moves increases with the size of the problem, it can be observed that the frequency at which each move is performed decreases as the size of the problem grows; the factor $\sqrt{N}$ normalizes the decrease of the frequencies. We have observed that the factor 0.5 in the computation of P provides generally good results, whatever the problem type and size are. Nevertheless, such a formula, as well as the formula that gives the taboo list size, is not based on theory.

Using a frequency-based memory, we have succeeded in improving every best known solution of 50-job and 20-machine problems proposed by Taillard[15] (see Table 3) without increasing the computation times.

3. Efficiency of Taboo Search

In this section, computational results are given first for well known instances of problems and then for some types of random problems.

A. Fisher-Thompson's[5] 10-Job-10-Machine Problem:

Although this famous problem might not be a good test of the general performance of a job shop scheduling method, we applied our approach to see whether it could find an optimal solution. The optimal solution of this problem was ultimately proved to be 930 in 1988 by Carlier and Pinson.[4] We have run our TS algorithm 15 times (first without frequency based memory), and we have found an optimal solution of the problem every time. The length of the taboo list was randomly changed every 15 iterations between 8 and 14.

The number of iterations necessary to find the optimal solution using only the short-term memory is huge: between $2 \cdot 10^6$ and $35 \cdot 10^6$ iterations. Adding a longer term frequency-based memory sometimes makes it possible to find an optimal solution much faster: performing 10 runs with a cut-off limit of 10^7 iterations (i.e., some hours on a personal work station) we found an optimal solution 8 times; in 2 cases, an optimal solution was found in less than 5 minutes ($2 \cdot 10^5$ iterations). But more intelligence should be added to TS if one wants to get an efficient method that provides sub-optimal solutions to small and hard problems. Indeed, exact branch and bound algorithms run faster at the present time than this implementation of TS for such small problems. However, the mean make span obtained after 10^4 iterations (i.e., some seconds on a personal workstation) is less than 3% above the optimal one. The time needed to solve optimally another 10-job-10-machine problem will be discussed also in section 4c.

B. Other Classical Problems:

We have considered problems from 3 different sources:

1) 40 problems of 8 different sizes due to Lawrence[10] and that were communicated to us by E. Pinson: 10 jobs × 10 machines, 15 × 10, 20 × 10, 30 × 10, 10 × 5, 15 × 5, 20 × 5, and 15 × 15. Five problems of each size have been considered.
2) 3 problems of size 20 × 15 due to Adams et al.[1] that were communicated to us by D. Applegate.
3) 80 problems of 8 different sizes (15 × 15, 20 × 15, 20 × 20, 30 × 15, 30 × 20, 50 × 15, 50 × 20, and 100 × 20) due to Taillard.[15]

We have repeated the experiments done by van Laarhoven et al.[8] on Lawrence's[10] problems, but by using sequential TS instead of SA; every problem was solved 5 times with independent trajectories (starting with the same initial solution but another seed for the random number generator used for the choice of the taboo list lengths). The computers used to do CPU time comparisons were identical (VAX 785).

The mean and the best make span obtained by TS after a fixed amount of time were uniformly better than those of SA. Figure 2 shows the comparison of the make spans found for the five 10 × 10 problems. The full line is the mean make span, and the dotted line is the best make span found by TS; the triangles represent the mean make span, and the circles the best make span found by SA. We can observe that TS is capable of getting a solution with a given make span about 10 times faster than SA; this is true for the other classical problems considered.

In the same figure, we have also plotted the mean solution value that the (generalized) shifting bottleneck procedures (*bottle*, *bottle-4*, and *bottle-5*) provide. We see that TS runs faster than these procedures in these instances. For other instances (with 20 and 30 jobs) *bottle* procedure sometimes runs slightly faster than TS. However, CPU time comparisons are hard to perform because the code of Applegate and Cook[2] runs more and more slowly with increases in the constants that fix the sizes of arrays (maximal number of jobs and machines...).

In Table 1, solutions obtained with TS are compared with the best solutions provided by some authors on the problems due to Lawrence[10] and Adams et al.,[1] for which the optimal solution is not yet established. (The names given to these problems are those given by Applegate and Cook.[2]) It turns out that our TS implementation has succeeded in improving every best-known solution.

Table 2 gives the percent above best-known solution values obtained by the *bottle* procedures for the problems of size up to 30 × 20 due to Taillard.[15] It was not possible

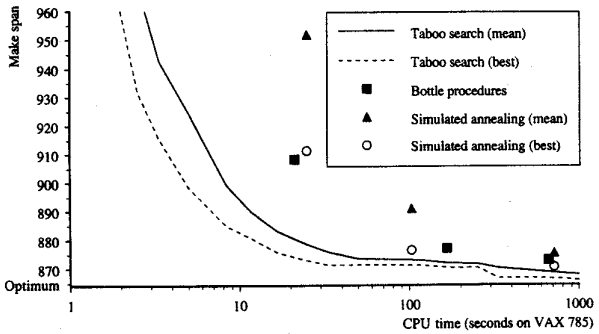

Figure 2. Performance of TS, SA, and bottle procedures on 10 × 10 problems.

Table I. Best Solution Values Found by Some Authors

Problem	Size (Jobs, Machines)	Adams, Balas and Zawack[1] Upper Bound	Applegate and Cook[2] Upper Bound	Taboo Search Upper Bound
ABZ7	20, 15	730	668	665
ABZ8	20, 15	774	687	676
ABZ9	20, 15	751	707	691
LA21	15, 10	1084	1053	1047
LA27	20, 10	1291	1269	1240
LA29	20, 10	1239	1195	1170
LA38	15, 15	1255	1209	1202

Table II. Percent Above Best-Known Solutions Provided by Some Methods (Run Time of TS Restricted)

	15 Jobs, 15 Machines	20 Jobs, 15 Machines	20 Jobs, 20 Machines	30 Jobs, 15 Machines	30 Jobs, 20 Machines
ABZ	9.0	10.1	10.0	8.3	13.1
TS	(3790) 5.2	(5544) 7.3	(8941) 6.6	(8174) 11.9	(14169) 13.6
Bottle-4	4.5	5.6	6.2	4.8	8.4
TS	(42423) 1.7	(64844) 2.6	(93176) 1.6	(105695) 2.1	(201542) 3.5
Bottle-5	3.4	4.4	4.8	4.3	7.3
TS	(142989) 0.9	(215935) 1.5	(343927) 0.9	(426762) 1.1	(762446) 1.4
Shuffle	3.1	4.0	4.7	4.2	7.2

to treat bigger problems because of run-time errors generated by *bottle* procedures, probably due to the explosion of the size of the search tree that these procedures use. The performance of TS without long-term memory is shown beside each *bottle* procedure, where the TS procedure is restricted to the same time as the *bottle* procedure. The numbers in parenthesis are the number of iterations performed by TS. We see that TS produces much better solutions than *bottle-4* and *bottle-5* procedures, even when restricted to terminate before finding its best solution. However, in some cases under this restriction, it may be favorable to use the shifting bottleneck procedure as proposed by Adams et al.[1] This suggests that one can create a TS method utilizing this procedure as well as the simple swap moves we incorporate.

In this table, we give also the performance of the *shuffle* post-optimization procedure proposed by Applegate and Cook.[2] As these authors suggest, we use the starting solution provided by the *bottle-5* procedure. Then we run *shuffle* with a set of parameters that allows the procedure to end after a "reasonable" computation time (between some seconds and some hours, depending heavily on the instance of problem). The improvements provided by this post-optimization procedure are very low in mean, especially for the biggest problems.

In sum neither the *bottle* procedures nor the *shuffle* procedure (applied to the solution produced by *bottle-5* procedure) succeeds in improving the best-known solutions to any of Taillard's[15] problems, and, hence, the best-known solutions continue to be those found by TS.

Finally, we summarize in Table 3 the best-known value for the problems proposed by Taillard.[15] Bold characters indicate the problems for which the frequency-based memory TS version has provided better solutions. The problems are given in the same order as they are published. Proven optimal values are indicated with an asterisk in the table, and we see that most of the 50×15 and 100×20 problems have been solved to optimality. (Optimality is established by attaining a lower bound, computed by a procedure contained in the Applegate and Cook[2] codes.)

C. Random Problems:

For additional numerical experiments, we have chosen the following type of problems (the same type as Fisher-Thompson's[5]):

- There are exactly m operations per job, one per machine.
- The processing times are randomly generated, uniformly distributed between 1 and 99 (integers).
- The sequence of operations of a job on the machines is independent from the other jobs.

Slight changes of our implementation suffice in order to treat many other types of problems (any number of operations per jobs, release date for each operation, set-up times depending on the previous operation on the same machine, open shop problems...).

Table III. Best-Known Solutions Values of Taillard's[15] Problems

15 Jobs, 15 Machines	20 Jobs, 15 Machines	20 Jobs, 20 Machines	30 Jobs, 15 Machines	30 Jobs, 20 Machines	50 Jobs, 15 Machines	50 Jobs, 20 Machines	100 Jobs 20 Machines
1231*	1376	1663	1770	2064	**2760***	**2921**	**5464***
1253	1381	1626	1853	1983	**2756***	**3002**	**5181***
1224	**1367**	1574	**1855**	**1896**	**2717***	**1835**	**5568***
1181	**1355**	**1660**	**1851**	2031	**2839***	**2775**	**5339***
1235	**1366**	1598	**2007**	**2032**	**2689**	**2800**	**5392***
1243	**1371**	1679	1844	2057	**2781***	**2914**	**5342***
1228	**1480**	1704	**1822**	**1947**	**2943***	**2895**	**5436***
1221	1432	**1626**	1714	**2005**	**2885***	**2835**	**5394***
1289	**1361**	1635	1824	2013	**2655***	**3097**	**5358***
1262	**1373**	**1614**	1723	1973	**2723***	**3075**	**5213**

For these problems, when $n \geqslant 6m$ $(m = 2 \ldots 10)$, for many thousand problem instances of various different sizes, we have always found a schedule which saturates a machine. So we conjecture that the optimal make span, when $m/n \to 0$, is surely given by:

$$C_{max} = \max_{j=1 \ldots m} \left(\sum_{i:M_i=j} d_i \right)$$

The independence of the sequences is important, because it is clear that, in the case of a general flow shop (which is a special case of a job shop problem for which the operations of every job uses the machines in the same order), the optimal make span is not given by this formula.

We were first interested in the mean number of iterations needed by TS to find an optimal schedule that saturates a machine for such problems. Figure 3 shows the dependence of the number of iterations, as a function of:

a) the number of jobs, for 5 machines;
b) the number of machines, for 200 jobs; and
c) the number of machines, for $n/m = 10$.

Graphically, we see that this dependence is polynomial (logarithmic scales!), and we see that 20×5 problems are harder than those of size 30×5. Fisher and Thompson[5] previously mentioned that squared problems ($n \cong m$) were harder than rectangular ones ($m \ll n$), but it is surprising that these may be solved in polynomial mean time. As an iteration may be done in time $O(nm)$, the mean complexity of our TS implementation is $O(n^{2.26}\ m^{3.88})$ for these problems. This complexity can be deduced by linear regression over the logarithm of the number of iterations needed to solve problems with $n \gg m$.

In Figure 4 we have plotted the mean CPU time needed by TS and *bottle* procedure, as a function of n, to solve simple problems with $m = 5$. It turns out that the mean complexity of the *bottle* procedure (as implemented by Applegate and Cook[2]) seems to be exponential; this fact is not surprising if one knows that this procedure solves potentially difficult sub-problems of size n by a branch and bound technique. (However, it is not clear whether the sub-problems created are more difficult than the original problem or whether the branch and bound implementation is poorly adapted for the sub-problems. Perhaps the number of sub-problems also grows exponentially.)

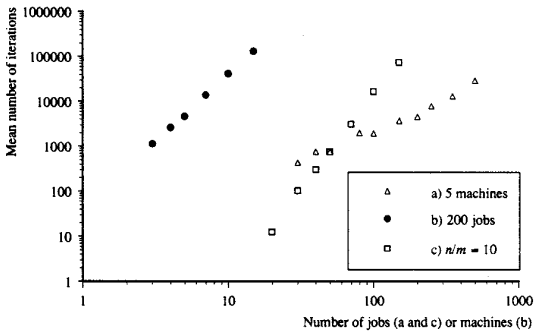

Figure 3. Mean number of iterations to solve optimally random problems.

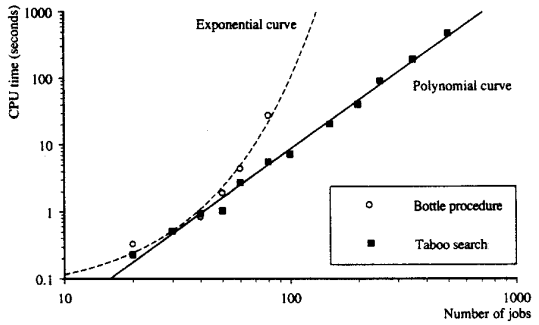

Figure 4. CPU time needed by TS and bottle procedures to solve optimally random problems with $m = 5$.

D. Big Problems:

Since finding optimal solutions for big problems with $n \gg m$ is very time consuming, it is of interest to find good solutions rapidly. The solutions of the greedy algorithm we used for generating the initial solution for TS are more than 8% above the optimal solution (or more precisely a lower

bound to the optimal solution) for problems with 5 or 10 machines, with 1,000 to 100,000 jobs, and with processing times randomly generated between 80 and 100 (see Figure 5). Generating such a solution takes a time proportional to $O(n^2m)$, and this complexity is relatively high. For such problems, it is possible to generate better solutions in $O(n \cdot g(m))$, where $g(m)$ is a function depending on the procedure used in step 2 of the following "divide and conquer" algorithm:

Algorithm Generating Good Solutions to Big Problems:

1) Divide up the original problem into p sub-problems, each including about n/p jobs, p proportional to n and depending on m (each job of the original problem appears once in the set of sub-problems).
2) Find a good schedule for every sub-problem, treating each as an independent problem.
3) Construct the global schedule by putting one sub-problem after the other and by scheduling in one time every operation of a sub-problem.

The larger the number of sub-problems into which the initial problem is divided in step 1, the faster the sub-problems will be solved in step 2 (assuming that the sub-problems are big enough to remain simple); but step 3 will become harder and the final schedule will be worse. The way of creating sub-problems may be arbitrary or more elaborate. (For example, if each job has a due date as subsidiary data, it seems reasonable to create sub-problems with jobs having approximately the same due dates.)

Since solving every sub-problem optimally by TS may take a great amount of time, solving them in a summary way may accelerate the computation without degrading the final solution too much. For example, one could use a TS that runs, at most, half the mean number of iterations needed to find an optimal schedule of the operations of each sub-problem.

An optimal sequence of sub-problems for step 3 will result by solving optimally an asymmetric traveling salesman problem with $p + 1$ cities: the distance from city i to city j $(i,j = 1,\ldots,n)$ is given by the minimum amount of time between the beginning of sub-problem i and the beginning of sub-problem j (as sequenced in step 2); the distance from city i to city $p + 1$ is the make span of sub-problem i and the distance from city $p + 1$ to city i is 0 $(i = 1,\ldots,n)$. The make span of the whole problem is clearly given by the length of the traveling salesman's tour if the sub-problems are sequenced as the corresponding cities in the tour, the first sub-problem treated being the one that corresponds to the city that follows city $p + 1$. In Figure 5, we give the performance (percent above a lower bound to the optimal solution as a function of the number of jobs) of this (parallel) algorithm if the sub-problems are sequenced in an arbitrary order (sizes of sub-problems: 25×5 and 50×10). This provides motivation for using (fast) heuristic methods for solving this asymmetric traveling salesman problem.

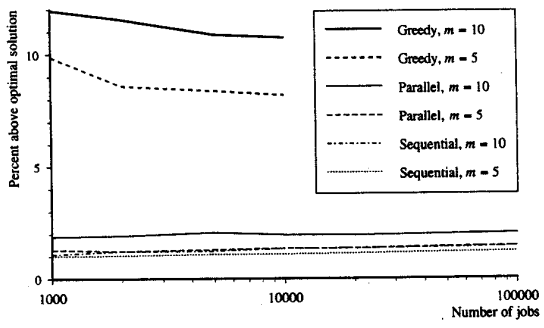

Figure 5. Performances of some algorithms for big problems.

A parallelization of the most time-consuming part of the algorithm, step 2, is straightforward, and it is easy to derive a parallel algorithm that generates good solutions in $O(\log n \cdot g(m))$ time using $O(n/\log n)$ processors: each processor solves $O(\log n)$ sub-problems in time $O(g(m))$, and the computation of the make span of the whole problem may be done in time $O(m \log n)$ if the sub-problems are sequenced in an arbitrary order in step 3 and if the connections between the processors create a tree (where $g(m)$ is larger than m).

If the sub-problems are solved sequentially in step 2, one could suppress step 3 by adding release dates to each operation: these release dates correspond to the ending date of the last operation on each machine of the previous sub-problem scheduled (if any). We see in Figure 5 that this sequential algorithm is slightly better than the parallel one that sequences the sub-problems in an arbitrary order. (The plot of the parallel algorithm with $m = 5$ is almost covered by the plot of the sequential one with $m = 10$.)

4. Parallelization of TS

As TS can be time consuming, it is of interest to use parallel computers to reduce the computing times. Consequently we propose some adaptations of our algorithm to parallel computers in this section.

A. Computation of the Longest Path:

Using a profiler program, we have observed that the most consuming part of our TS algorithm is the computation of the longest paths. If we consider the algorithm of section 2 to calculate the r_i's, we see that step 2 may be computed in parallel for every element in Q. So, this algorithm may be expressed as follows: let A_i be the maximal number of operations on a path from 0 to i $(i = 0 \ldots N + 1,\ A_0 = 0)$. So $A_i = \max(A_{PM_i},\ A_{PJ_i}) + 1$. Given the values of A_i, we might use the following algorithm to calculate the r_i's:

a) For $k = 1$ to $\max_i A_i$ do
   b) Compute r_i for the operations such that $A_i = k$

Operation b, which corresponds to operation 2 of the algorithm of section 2 if Q is handled as a queue (first in,

first out), may be computed in parallel, so we have a theoretic speed-up S of:

$$S = \frac{N}{\max_i A_i}$$

This speed-up depends on the problem and on the schedule; the bounds on S are 1 in the worst case and m in the best case (for example, in a general flow shop problem).

In practice, the values of A_i need to be calculated, and the operations for which this value is identical need to be grouped; this requires additional work. Thus, we propose a different parallel approach for computing the longest path in which we allocate one process to each of the m machines. The principle used may be expressed as follows:

Process *machine* j:

0) Send to process *machine* M_{SJ_i} the provisional value $r_i + d_i$ and label i, for every operation i that may be processed initially by machine j (that is, considering the first operations of jobs).
1) Repeat, until every r_i has been computed:
   2) Wait for a provisional value of the earliest beginning time of an operation i such that $M_i = j$; this value is communicated by another process.
   3) Store this value
   4) For every operation i for which the provisional value is known and r_i is computable (i.e., r_{PM_i} is already computed), label i, calculate r_i, send to process *machine* M_{SJ_i} (if it exists) the provisional value $r_i + d_i$.

An implementation on a distributed machine (MIMD, Transputer for example) works well, since only the values for the operations on machine j must be known by the process *machine* j. Here, the speed-up of the algorithm is bounded by the previously defined value S.

We have implemented this parallel method on Transputers, involving one process per machine, where every process runs on a single processor. Figure 6 gives the speed-up obtained using m processors connected in ring.

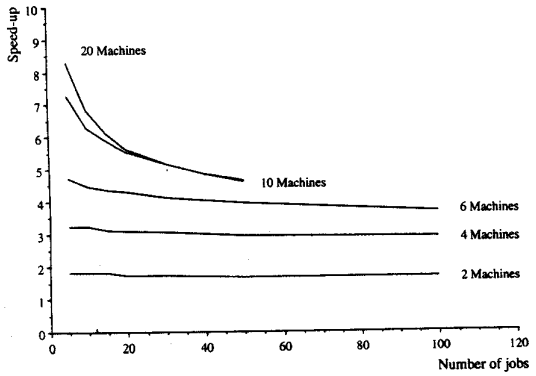

Figure 6. Speed-up of the Transputer's implementation.

The reason for the relatively bad performance of this algorithm for a high number of processors is due to two factors. First, the parallel formulation has a speed-up limited by the value of S. Second, the interconnecting topology is not appropriate for a complete interconnecting. Hence, the transmission times are large compared to the computation times.

In the case of a general flow shop sequencing problem, the configuration is ideal for our implementation since *machine* j has to transmit information only to *machine* j + 1. In this case, the speed-ups are much better and do not depend on the problem and on the current solution. We have observed speed-ups of 14 with 15 processors. The communication times remain significant, but, if the computations become harder (for example, if there are set-up times or if the processing times are not fixed, ...), then this implementation may become interesting.

B. Determining Critical Operations:

In order to determine the critical operations, our TS needs first to calculate the q_i's. This may be done in parallel with the computation of the r_i's. Determining a longest path from 0 to $N + 1$ may be done as follows:

a) Calculate $r_i \; \forall i$
b) Calculate $q_i \; \forall i$
c) Calculate $C_{max} = \max_i (r_i + q_i)$
d) Determine the critical operations, i.e., $\{i \mid r_i + q_i = C_{max}\}$.

A second parallel level appears here, because it is possible to perform steps a and b independently; however, an efficient implementation supposes a common memory, since steps c and d need simultaneous accesses to variables r_i and q_i. So a PRAM (parallel random access machine) is well suited for this process. We have tried to implement such a parallelization on a Cray-2 with 2 processors. But such a computer is not designed to treat applications with frequent synchronizations, and we have observed that the time needed to synchronize two tasks executed on different processors can be greater than the computation time of the r_i's. So, such a big computer is not useful to treat the problems we are considering. However, other computers might speed up significantly the search with such a parallelization.

C. General Method of Parallelizing Random Algorithms:

The method presented here has been known for a long time and is currently used for a wide range of problems (see, for example, the works of Roussel-Ragot et al.,[12] Mohr,[11] and Taillard[13,14]). However, we have not seen any theoretical analysis of it.

Let $\mathcal{A}$ be an iterative algorithm that has a probability $1 - q(t)$ to meet a condition after time t. Let us suppose that the time needed to meet this condition depends only on the initial solution given to $\mathcal{A}$, or on a random parameter chosen by $\mathcal{A}$, depending only on the running occurrence of $\mathcal{A}$. In practice, $\mathcal{A}$ may be a TS, a SA, or any random

iterative algorithm, and the condition to meet may be: "optimal solution found."

If $\exists a > 1$, $\hat{t} > 0$ and $p \geqslant 2$ integer such that

1) $q(t) \leqslant a^{-t}$ $\quad (\hat{t} \leqslant t \leqslant p\hat{t})$
2) $q(t) \geqslant a^{-t}$ $\quad (p\hat{t} \leqslant t)$

then algorithm $\mathscr{B}$, which consists of executing $\mathscr{A}$ p times during time t, has a higher probability of meeting the condition for $t \in [\hat{t}, p\hat{t}]$ than algorithm $\mathscr{A}$ executed during time pt. We have:

$$\begin{aligned} & q^p(t) \leqslant (a^{-t})^p = a^{-pt} && (\hat{t} \leqslant t \leqslant p\hat{t}) \\ & q(pt) \geqslant a^{-pt} && (\hat{t} \leqslant t) \\ & \Rightarrow q^p(t) \leqslant q(pt) && (\hat{t} \leqslant t \leqslant p\hat{t}) \end{aligned}$$

Note that the conditions needed if algorithm $\mathscr{B}$ is to be better than $\mathscr{A}$ are rather strong. However, in many cases, the empirical function $q(t)$ for iterative algorithms is not very far from an exponential one. So, the execution of many independent trajectories is a very efficient form of parallelization, since the speed-up is nearly ideal.

Figure 7 shows an empirical curve $q(t)$ and the best exponential curve interpolating $q(t)$. This empirical curve was found by solving the 10×10 problem LA21 independently 500 times with our TS algorithm. In this case, the sequential algorithm $\mathscr{B}$ would not be as good as the initial one, but Figure 7 shows that a parallelization of it will lead to very good speed-ups. For example, if the optimal solution of problem LA21 is desired with probability 0.95, a speed-up of about 14 can be obtained with 20 processors.

5. CPU Times

Above, we have purposely suppressed references to CPU times. Indeed, these depend strongly on the implementation of the program; sometimes minor changes lead to major fluctuations in computing time. For example, the re-sizing of arrays in the *bottle* procedures may significantly alter the speed of these procedures.

In Table 4, we give the mean time by iteration and by operation needed by our sequential implementation on some different computers. More precisely, one iteration of our algorithm for a problem with N operations takes a time given by the appropriate constant of this table multiplied by N. For example, 10,000 iterations of our implementation of TS on a VAX 8600 takes about 1 minute (64 seconds) for a problem with 100 operations.

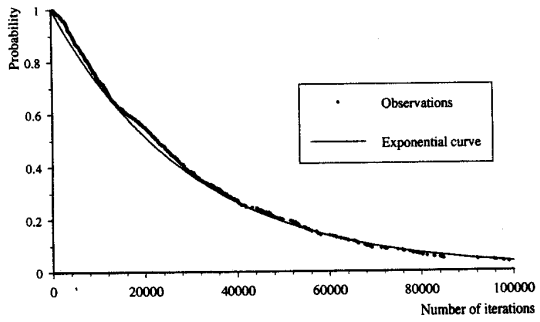

Figure 7. Empirical function $q(t)$.

Table IV. Speed of Several Computers

VAX 785	VAX 8600	Silicon Graphics (10 Mips)	Cray-2
167 μs	64 μs	17.2 μs	17.5 μs

The VAX and Silicon Graphics computers have been programmed in Pascal and compiled with optimization options, the Cray in Fortran (without vectorization).

6. Conclusion

We have presented a special method based on taboo search for minimizing the make span of a job shop scheduling problem, considering both sequential and parallel implementations. We have implemented a parallel version of our algorithm on a distributed computer. Although the parallel approach is not well suited for problems with fixed processing times and fully independent orders of operations on machines, we think that our implementation may be interesting for practical problems, especially if the processing times are given by some complicated function and if there exists a certain order on the machines for the operations of a job.

In the sequential case, we have shown that TS is a very competitive technique. First it is very simple to implement. The number of iterations done by a search may be fixed *a posteriori*, unlike SA where the cooling parameters determine the length of the execution. Further, our version of TS is much more efficient than the SA version proposed by other authors, and for many problems is more efficient than the shifting bottleneck procedure (which is much more difficult to implement than TS). The addition of a long-term memory that is very easy to implement is an efficient way to improve the quality of the solutions produced by long TS runs.

Although it was not possible to demonstrate that TS converges to an optimal solution, our implementation never exhibited cycling phenomena for reasonable parameters choices.

For small "square" problems ($n \cong m$), TS is slower than the best branch and bound methods. However, when the problem grows or becomes "rectangular," the efficiency of TS is higher than any other exact or heuristic methods published. In addition, our TS approach establishes new best known solutions for every problem in two sets of benchmark problems from the literature.

Surprisingly, we have observed that TS optimally solves large random problems with $m \ll n$ in a polynomial mean time. Thus, we have obtained optimal solutions of some 10,000 operation problems (2000 jobs and 5 machines). Problems of this size have never been addressed in previous studies.

Finally, we have presented a parallel algorithm that very rapidly generates good solutions to huge problems (including 50,000, 1,000,000 ... operations).

Possibilities for future research include the use of other features of TS and other neighborhood definitions as bases for archiving further enhancement of solution capabilities of job shop scheduling problems.

Acknowledgments

The author would like to thank the anonymous referees for their comments. Special thanks are addressed to Fred Glover, who greatly improved the presentation of this paper by his multiple and valuable suggestions. This research was supported by the Fond National suisse pour la Recherche Scientifique, grant number 20-27926.89.

References

[1] J. ADAMS, E. BALAS and D. ZAWACK, 1988. The Shifting Bottleneck Procedure for Job Shop Scheduling, *Management Science 34*, 391–401.

[2] D. APPLEGATE and W. COOK, 1991. A Computational Study of the Job-Shop Scheduling Problem, *ORSA Journal on Computing 3*, 149–156.

[3] J.W. BARNES and M. LAGUNA, 1993. A Tabu Search Experience in Production Scheduling, in *Tabu Search*, F. Glover, M. Laguna, É. Taillard and D. de Werra (eds.), *Annals of Operations Research 41*, 141–156.

[4] J. CARLIER and E. PINSON, 1989. An Algorithm for Solving the Job-Shop Problem, *Management Science 35*, 164–176.

[5] H. FISHER and G.L. THOMPSON, 1963. Probabilistic Learning Combinations of Local Job-Shop Scheduling Rules, in *Industrial Scheduling*, J.F. Muth and G.L. Thompson (eds.), Prentice-Hall, Englewood Cliffs, NJ, pp. 225–251.

[6] F. GLOVER, 1989. Tabu Search—Part I, *ORSA Journal on Computing 1*, 190–206.

[7] F. GLOVER, 1990. Tabu Search—Part II, *ORSA Journal on Computing 2*, 4–32.

[8] 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.

[9] E.L. LAWLER, J.K. LENSTRA, A.H.G. RINNOOY KAN and D.B. SHMOYS, 1989. *Sequencing and Scheduling: Algorithms and Complexity, Report BS-R89xx*, Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands.

[10] S. LAWRENCE, 1984. *Resource Constrained Project Scheduling: An Experimental Investigation of Heuristic Scheduling Techniques* (supplement), Graduate School of Industrial Administration, Carnegie Mellon University.

[11] T. MOHR, 1988. *Parallel Taboo Search Algorithms for the Graph Coloring Problem*, Report ORWP 88/11, DMA, École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland.

[12] P. ROUSSEL-RAGOT, P. SIARRY and G. DREYFUS, 1987. *La Méthode du «Recuit Simulé» en Électronique: Principe et Parallélisation, Internal Report*, École Supérieure de Physique et de Chimie Industrielles de la Ville de Paris, Paris, France.

[13] É. TAILLARD, 1990. Some Efficient Heuristic Methods for the Flow Shop Sequencing Problem, *European Journal of Operational Research 47*, 65–79.

[14] É. TAILLARD, 1991. Robust Taboo Search for the Quadratic Assignment Problem, *Parallel Computing 17*, 443–455.

[15] É. TAILLARD, 1993. Benchmarks for Basic Scheduling Problems, *European Journal of Operational Research 64*, 278–285.