Applying Tabu Search to the Job-Shop Scheduling Problem

Article *in* Annals of Operations Research · September 1993

DOI: 10.1007/BF02023076

CITATIONS
562

READS
2,237

2 authors:

Mauro Dell'Amico
Università degli Studi di Modena e Reggio Emilia
103 PUBLICATIONS **3,634** CITATIONS

SEE PROFILE

Marco Trubian
University of Milan
62 PUBLICATIONS **2,555** CITATIONS

SEE PROFILE

Annals of Operations Research 41(1993)231–252

Applying tabu search to the job-shop scheduling problem*

Mauro Dell'Amico and Marco Trubian
Politecnico di Milano, I-20133 Milano, Italy

Abstract

In this paper, we apply the tabu-search technique to the job-shop scheduling problem, a notoriously difficult problem in combinatorial optimization. We show that our implementation of this method dominates both a previous approach with tabu search and the other heuristics based on iterative improvements.

1. Introduction

The *job-shop* scheduling problem which can be described as $J||C_{max}$ using the three fields classification introduced in Graham et al. [9], is the following. A set M of m *machines* and a set J of n *jobs* are given. The ith job consists of a chain of m_i *operations* from set $O = \{1, \ldots, N\}$, with $N = \sum_{k=1}^{n} m_k$. Each operation $i \in O$ belongs to job J_i and has to be processed on machine μ_i for d_i consecutive time instants. The problem is to assign the operations to time intervals in such a way that:

- no one job is pre-empted,
- the precedences given by the chain relations are respected,
- no two jobs are processed at the same time on the same machine,
- the maximum of the *completion times* (C_i) of all operations (*makespan*) is minimized.

This problem, which has been studied for a long time, is known to be *NP-hard* [6] and has the well-earned reputation of being one of the more difficult combinatorial problems considered to date. An indication of its difficulty is given by the fact that the famous 10×10 instance formulated for the first time by Muth and Thompson in 1963 was exactly solved only in 1989 by Carlier and Pinson with a

*Partially supported by research contracts MPI 40% and 60% of the Italian Ministry of University and Scientific Research.

branch and bound algorithm which required about 5 hours of computing time on a PRIME 2655. More recently, other branch and bound algorithms have been proposed by Applegate and Cook [2], and Brucker et al. [3], which drastically improve the computational performance of the previous one on the 10×10 instance (372 seconds on a SUN Sparcstation 1). However, these algorithms are quite sensitive to the particular instance considered. In addition to exact methods, many heuristics have been developed; the most popular are *List Schedule* algorithms which assign one operation at a time from a list ordered by some priority rule (see, e.g. [15] for a comprehensive survey). A more sophisticated algorithm, called *Shifting Bottleneck*, was given by Adams et al. [1]. The algorithm builds up and improves a schedule by iterative solutions of a single *bottleneck* machine problem. Better solutions than the ones given by deterministic algorithms were found using simulated annealing [12, 18] but at the cost of longer computations. Tabu search was first applied to job shop by Taillard [17], who proposed a sequential and a parallel algorithm. The first solves the 10×10 instance exactly in more than 9 hours of computing time on a VAX 785. Taillard observed that this algorithm has a worse computational performance than the branch and bound method for squared problems ($n = m$), but has a higher efficiency for rectangular instances ($n > m$). More recently, the problem has been approached by a *conventional genetic algorithm* [14]. In this paper, we propose a randomized procedure, based on a priority rule, for generating feasible starting solutions and a randomized local search algorithm, based on the tabu-search technique, for solving the problem. We study the performance of our method by means of a well-known set of benchmark instances. In section 2, we introduce tabu search and a useful formulation of the problem. In section 3, we describe a new procedure for generating feasible starting solutions, and in the following section, we describe the neighborhood structures we have adopted. Section 5 presents the tabu-search approach as we used it, and in the final section we give the computational results and some final remarks.

In the following we will assume, as is usually done, that all input data of the problem are non-negative integers.

2. A tabu-search framework for job shop

Any instance of a combinatorial optimization problem is associated with a finite set of feasible solutions; each of which is characterized by a cost. The goal is to find a solution of minimum (or maximum) cost.

Given a problem P, let S denote the set of feasible solutions to P and $c: S \rightarrow \mathbb{R}$ its cost function. In order to derive a local search based algorithm for P, it is necessary to define a *neighborhood structure*, that is, a function $N: S \rightarrow 2^S$ which associates a set of solutions $N(s)$ with each solution $s \in S$ obtainable by a predefined partial modification of s, usually called *move*. Starting from an initial solution generated independently, a local search algorithm repeatedly replaces the current solution by a neighboring one until a superimposed stopping criterion becomes true. The algorithm returns the best solution found, with respect to the cost function.

Tabu search is a local search based optimization method: the search moves from one solution to another, choosing the best not *forbidden* element in the neighborhood. This method forbids solutions with certain attributes with the goals of preventing cycling and guiding the search towards unexplored regions. Without using the technique of forbidden solutions, starting from a local optimum s (i.e. a solution such that all elements in $N(s)$ have a worse cost than s), the method chooses the best solution in $N(s)$ and then conceivably at the next step falls back into the local optimum again. However, storing complete solutions in a *forbidden list* and testing if a candidate solution belongs to the list is generally too expensive, both for memory and for computational time requirements. Usually, a *tabu list* is defined which stores only the opposite of the move applied during the search to transform a solution into a new one (i.e. the move which leads from the new solution to the old one). A solution s' is considered forbidden if the current solution s can be transformed into s' by applying one of the moves in the tabu list. In addition to a tabu status, a so-called *aspiration criterion* is associated with each move. If a current tabu move satisfies the associated aspiration criterion, it is considered an admissible move.

For a precise and complete description of this method, the interested reader can refer to the papers of Glover [7,8]. Here, we present the general framework of our tabu-search algorithm, while the details will be discussed in section 5.

PROCEDURE *TS*

begin
⟨find an initial feasible solution s⟩;
$best := c(s)$;
$s^* := s$;
$Tabu\_list := \emptyset$;
repeat
$Cand(s) := \{s' \in N(s)$: the move from s to s' does not belong to *Tabu_list* or it satisfies an *aspiration criterion*$\}$;
⟨choose $\bar{s} \in Cand(s) : \bar{s}$ has the minimum *estimation* of the cost function⟩;
⟨put a move which leads from $\bar{s}$ to s in *Tabu_list*⟩;
$s := \bar{s}$;
if $c(s) < best$ **then**
begin
$s^* := s$;
$best := c(s)$
end
until *stopping_criteria* = TRUE;
return s^*
end

In order to describe our procedure for generating feasible starting solutions and the neighborhood structures adopted, we introduce the disjunctive graph theory model for the problems which is due to Roy and Sussmann [16].

Given an instance of $J||C_{max}$, we can associate with it a disjunctive graph $G = (V, A, E)$, with V = set of nodes, A = set of conjunctive directed arcs, E = set of disjunctive undirected arcs (edges), defined as follows:

$V = O \cup \{0\} \cup \{N+1\}$, ($\{0\}$ and $\{N+1\}$ are special nodes which identify the start and completion of the overall job shop);

$A = \{(i, j) : \text{operation } i \text{ is an immediate predecessor of operation } j \text{ in the chain of job } J_i(=J_j)\} \cup \{(0, j) : j \in O\} \cup \{(i, N+1) : i \in O\}$;

$E = \{(i, j) : \mu_i = \mu_j,\ i, j \in O\}$.

With each vertex $i \in O$, a weight d_i is associated, vertices 0 and $N+1$ have weight zero. The starting time and the completion time of vertices 0 and $N+1$ represent, respectively, the starting and finishing times of the overall job shop. Directed arcs between vertices associated with operations represent the precedence relation; the edges represent the machine capacity constraints. One can see that any orientation of the edges which does not create cycles corresponds to a feasible sequencing of the operations on the machines.

Once the length of a path is defined as the sum of the weights of the vertices in the path, solving the job shop corresponds to finding an acyclic orientation of G so that the length of the longest path between 0 and $N+1$ (*critical path*) is minimized.

3. A starting solution

Our procedure for finding feasible starting solutions is a new List Schedule algorithm. This class of algorithms first defines a rule of assigning priorities to the operations, then schedules the operation each time with maximum priority in subsequent stages. Usually, the operations are scheduled by increasing or decreasing time. In the first case, the operations which can be feasibly assigned are those for which all predecessors are already scheduled; in the second case, an operation can be assigned if all its successors have been scheduled. Our algorithm mixes the two approaches and schedules one from the beginning (time increasing) and one from the end (time decreasing). This *bi-directional* method finds a justification in the following argument. On average, when a single-direction algorithm is applied it is able to find "good" partial schedules when many more operations have to be scheduled, but the scheduling of the last operations strongly depends upon the previous assignments, and in the last stages the solutions generally worsen considerably. Our bi-directional algorithm operates as two single-direction procedures which construct two "half-schedules", one from the beginning and the second from the end. These two partial schedules are "good" and generally the complete one obtained combining them is still good

enough. The computational results (see section 6) confirm the effectiveness of our approach.

Let us describe the algorithm more precisely. An operation is said to be *schedulable* if either all its predecessors or its successors are scheduled. Let L denote the set of scheduled operations in order of increasing time and S the set of schedulable operations with predecessors in L. Let R and T denote the corresponding sets for the reverse order. Let $PJ[i]$ and $SJ[i]$ denote the immediate predecessor and the immediate successor of operation i of the job J_i. Let $PM[i]$ and $SM[i]$ denote the immediate predecessor and the immediate successor of operation i on the machine μ_i if they are defined in the current partial schedule, $PM[i] = 0$, $SM[i] = 0$ otherwise. Let r_i denote the earliest starting time of operation i (i.e. r_i is the length of a longest path from 0 to i minus d_i) and let t_i denote the queue of operation i (i.e. the length of a longest path from i to $N+1$ minus d_i).

PROCEDURE *Bidir*

begin
0. **initialization**
   $L := \{0\}$, $R := \{N+1\}$;
   $S := \{i : i$ is the first operation of job $J_i\}$, $T := \{i : i$ is the last operation of job $J_i\}$;
   $r_i := 0\ \forall i \in S$, $t_i := 0\ \forall i \in T$;
1. **repeat**
2. ⟨choose an operation $i \in S$ using a priority rule⟩;
3. ⟨put i on machine μ_i in the first position free from the beginning i.e. orientate all edges $[i, k]$ with $k \notin L$ from i to k⟩;
4. $S := S - \{i\}$; $L := L + \{i\}$;
5. **if** $i \in T$ **then** $T := T - \{i\}$;
6. **if** $(SJ[i] \notin R)$ **then** $S := S + \{SJ[i]\}$;
7. ⟨update $r_i\ \forall i \in S$⟩;
8. **if** $|L \cup R| \neq N+2$ **then**
   begin
9. ⟨choose an operation $i \in T$ using a priority rule⟩;
10. ⟨put i on machine μ_i in the first position free from the end i.e. orientate all edges $[i, k]$ with $k \notin R$ from k to i⟩;
11. $T := T - \{i\}$; $R := R + \{i\}$;
12. **if** $i \in S$ **then** $S := S - \{i\}$;
13. **if** $(PJ[i] \notin L)$ **then** $T := T + \{PJ[i]\}$;
14. ⟨update $t_i\ \forall i \in T$⟩;
   end
15. **until** $|L \cup R| = N+2$;
end

THEOREM 3.1

Procedure *Bidir* always produces feasible solutions.

Proof

Let $U = (W, V\backslash W)$, $W \subset V$ be a $(0, N+1)$ cut, i.e. a cut so that $0 \in W$, $N+1 \in V\backslash W$. From steps 3, 4 and 6 we can see that at each stage of the procedure any arc $(i, j) \in U$, with $W \subseteq L$, is oriented from W to $V\backslash W$. From steps 10, 11 and 13, all arcs $(i, j) \in U$ with $(V\backslash W) \subseteq R$ are oriented from W to $V\backslash W$. Since L and R are disjoint by construction, when the algorithm terminates each $(0, N+1)$ cut contains only arcs directed from W to $V\backslash W$ and no cycle can exist. □

The crucial points in this procedure are steps 2 and 9. We are going to describe in some detail only step 2, since step 9 is basically the same if we interchange vectors r, t, sets S, T and sets L, R. Our priority rule consists in ordering the operations according to nondecreasing values of a lower bound $est(i)$ of the length of a longest path in G from 0 to $N+1$ going through i, on the hypothesis that i will be the next operation sequenced on μ_i, i.e. i will be the next operation selected from S and put in L.

When the schedule is completed and values r, t computed consequently, a longest path passing through node i in the acyclic graph G has a length of $r_i + d_i + t_i$. At each iteration of our procedure, we know the exact values r_i for all $i \in L$ and for each $i \in S$, if it is scheduled in the current iteration. So we need to estimate t_i for each $i \in S$. Since each node i in G has at most two successors on the longest path from i to $N+1$, the first on the chain of operations belonging to the same job and the second on the same machine, the following results:

$$t_i = \max \{d_{SJ[i]} + t_{SJ[i]},\ d_{SM[i]} + t_{SM[i]}\}. \tag{1}$$

We know the exact value of t_i for all $i \in R$, while we can use a lower bound $\hat{t}_i$ for $i \in V\backslash R$. In the first part of the max in (1), we can substitute $\hat{t}_{SJ[i]}$ for $t_{SJ[i]}$; unfortunately, the same is not possible for the second part since $SM[i]$ is not defined for $i \in S$. However, we can estimate $d_{SM[i]} + t_{SM[i]}$ as $\max_{j \in \overline{V}} \{d_j + \hat{t}_j\}$, where $\overline{V} = \{j \in V\backslash(L \cup \{i\}) : \mu_j = \mu_i\}$ is the set of operations to be performed on machine μ_i after operation i if this one is scheduled in the current iteration. The lower bound $\hat{t}_i$ is defined as the value of the longest path from i to $N+1$ in the directed acyclic subgraph of G determined by the arc set A and by the already oriented edges of set E. The estimation of the longest path from 0 to $N+1$ through $i \in S$ is then:

$$est(i) = r_i + d_i + \max \{d_{SJ[i]} + \hat{t}_{SJ[i]},\ \max_{j \in \overline{V}} \{d_j + \hat{t}_j\}\}.$$

Given the bound $est(i)$, we choose the next operation to schedule using a *cardinality-based semi-greedy heuristic with parameter c* [10]. With this approach, a decision from a set of candidates is randomly selected among the c decisions with lower estimation. If $c = 1$, the procedure has the same behavior as a greedy heuristic based

on a priority rule. We decided to choose the randomized approach since we have experimentally observed that it gives, on average, better solutions than the corresponding deterministic version. A final observation about the algorithm is that, as pointed out in [5], this kind of randomized procedure does not guarantee to produce a local optimum with respect to even simple neighborhood structures.

4. Neighborhood structures

Given a set of feasible solutions, in order to apply local search one has to define a neighborhood structure $N : S \to 2^S$, where S denotes the set of solutions of the problem. Up to now, two different kinds of functions N have been proposed, denoted by $N1$ and $N2$ later on. Both of these neighborhoods are based on the following properties [18]:

(1) if $s \in S$ is a feasible solution, then reversing one of the oriented edges on a critical path of s can never lead to an infeasible solution;

(2) if the reversal of an oriented edge of a feasible solution s that does not belong to a critical path leads to a feasible solution s', then the critical path in s' cannot be shorter than the critical path in s.

With $N1$ [18], a neighboring solution s' of the current solution s is obtained by permuting two successive operations v and w assigned to the same machine and for which the arc (v, w) is on a critical path in s.

Neighborhood $N2$ [12] selects two operations with the same rules as before among those couples (v, w) for which at least one of the arcs $(PM[v], v)$ and $(w, SM[w])$ is not on a critical path. This restriction is motivated by the fact that the swap of (v, w), when both $(PM[v], v)$ and $(w, SM[w])$ are on a longest path, cannot improve the makespan directly since the minimum starting time of $SM[w]$ does not change. Moreover, for each candidate arc (v, w), the arcs $(SJ[v], SM[SJ[v]])$ and $(PM[PJ[w]], PJ[w])$ are also checked to see if reversing one of them and (v, w) at the same time can lead to a shorter critical path.

For the first neighborhood, the following *connectivity* property holds:

THEOREM 4.1 [18]

For each feasible solution s it is possible to construct a finite sequence of moves, using $N1$, which will lead from s to a globally minimal solution.

The following counterexample shows that the same property does not hold for $N2$.

EXAMPLE 4.1

Consider an instance of a job shop with two machines and four jobs, each of which consists of a chain of two operations to be performed on machines one

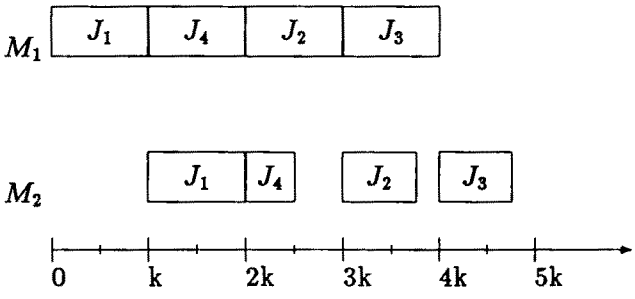

Fig. 1.

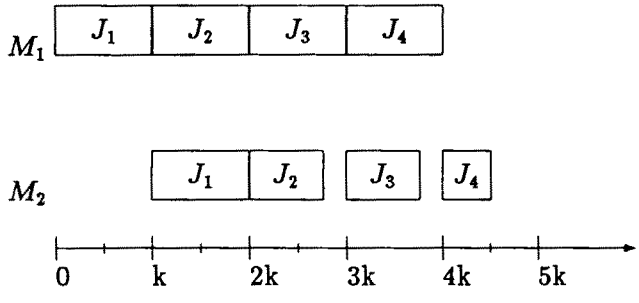

Fig. 2.

and two, respectively. The processing times of the operations on machine one are all equal to a positive integer k, while for the second machine the processing time of J_1 is k, that of J_2 and J_3 is $3k/4$, and that of J_4 is $k/2$. Given the feasible solution of fig. 1 with makespan $19k/4$ and the optimal solution with value $9k/2$ (fig. 2), it is easy to see that it is impossible to schedule J_4 as the last job on the first machine starting from the first solution and using only swaps of neighborhood $N2$. (Note that in this example we have a *flow shop* with 2 machines which can be solved in polynomial time with the algorithm proposed by Johnson [19].)

We have developed two new neighborhood structures NA and NB, and for the first one a restricted version named RNA. Neighborhood NA is an extension of $N1$ which considers the possible inversion of more than one arc at the same time. In particular, given two operations i, j assigned to the same machine and such that arc (i, j) belongs to a critical path, we consider all possible permutations of $\{PM[i], i, j\}$ and $\{i, j, SM[j]\}$ in which arc (i, j) is inverted. To estimate the value of the new solutions, we use an approach similar to that used in Taillard [17]. To evaluate the effect of inverting a single oriented edge (i, j), Taillard computes the exact value of the longest path which contains at least one of the vertices i, j in the graph associated with the new solution. The length of this path is a valid lower bound on

the value of the new solution. Similarly, for each possible new solution s' generated by *NA*, starting from solution s we compute the exact value of the longest path in s' which contains at least one of the nodes involved in the inversion of the arcs. Given an operation k, let r_k, t_k and r'_k, t'_k be the minimum starting time and the longest path from k to $N+1$ in schedule s and s', respectively, and let $PM'[k]$ and $SM'[k]$ be the predecessor and successor of operation k in s'. Moreover, let $Q = \{Q_1, \ldots, Q_q\}$ be a set of operations to be permuted in order to obtain solution s' from s. Observing that $r_k = r'_k$ for all predecessors of Q_1 and $t_k = t'_k$ for all successors of Q_q, we can compute the longest path going through one of the nodes in Q with the following procedure.

PROCEDURE *lpath*(q, Q)

begin
  $a := Q_1$;
  $r'_a := \max\{r_{PJ[a]} + d_{PJ[a]}, r_{PM'[a]} + d_{PM'[a]}\}$;
  for $i = 2$ **to** q **do**
  begin
    $b := Q_i$;
    $r'_b := \max\{r_{PJ[b]} + d_{PJ[b]}, r'_a + d_a\}$;
    $a := b$
  end;
  $b := Q_q$;
  $t'_b := \max\{t_{SJ[b]} + d_{SJ[b]}, t_{SM'[b]} + d_{SM'[b]}\}$;
  for $i = q - 1$ **to** 1 **do**
  begin
    $a := Q_i$;
    $t'_a := \max\{t_{SJ[a]} + d_{SJ[a]}, t'_b + d_b\}$;
    $b := a$
  end
  return $(\max_{i=1,\ldots,q}\{r'_{Q_i} + d_{Q_i} + t'_{Q_i}\})$
end

For each arc (u, v) belonging to a critical path such that u and v have to be processed on the same machine, the estimation of the value of the new solution obtained by applying *NA* is given by:

PROCEDURE *estim*(u, v)

begin
  $e_1 := lpath(2, \{v, u\})$;
  if exists $PM[u]$ **then**
  begin $e_2 := lpath(3, \{v, PM[u], u\})$;
    $e_3 := lpath(3, \{v, u, PM[u]\})$
  end;

```
    if exists SM[v] then
    begin e4 := lpath(3, {v, SM[v], u});
          e5 := lpath(3, {SM[v], v, u})
    end;
    ⟨output the sequence i with the lowest e_i, in the case of tie
     output the sequence with lowest index i⟩
end
```

The properties of connectivity and feasibility of neighborhood *NA* are given by the following two theorems.

THEOREM 4.2

For each feasible solution s, it is possible to construct a finite sequence of moves, using *NA*, which lead from s to a globally minimal solution.

Proof

$N1 \subset NA$. □

THEOREM 4.3

For each candidate arc, procedure *estim* outputs a sequence corresponding to a feasible solution.

Proof

Given a candidate arc (u, v) of the critical path, it is sufficient to prove that the value returned by procedure *lpath*, when only arc (u, v) is inverted (which always produces a feasible solution (theorem 4.1)), is not greater than the value returned by *lpath* for any infeasible solution obtained with the inversion of more than one arc. In particular, we will completely describe the proof that $lpath(2, \{v, u\}) \leq lpath(3, \{v, PM[u], u\})$. The other cases considered in the procedure *estim* can be proved with similar arguments. Let $x = PM[u]$ and suppose the permutation v, x, u produces an infeasible solution. Since the inversion of the only arc (u, v) cannot produce infeasibility (fig. 3), a path from $SJ[x]$ to $PJ[v]$ must exist (fig. 4). Let r_i', t_i' and r_i'', t_i'' be the new values of r_i and t_i as computed by *lpath* when the inversion of the only arc (u, v) of the permutation v, x, u is considered, respectively. Looking at fig. 3 and fig. 4, it is easy to see that $t_u' = t_u''$. Moreover, we have $r_v' = r_v''$ since $r_{PJ[v]} > r_x + d_x$. From these observations it follows that:

$$r_u' = \max\{r_v' + d_v, r_{PJ[u]} + d_{PJ[u]}\},$$

$$r_u'' = \max\{r_x'' + d_x, r_{PJ[u]} + d_{PJ[u]}\}$$

and

$$r_x'' \geq r_v'' + d_v = r_v' + d_v.$$

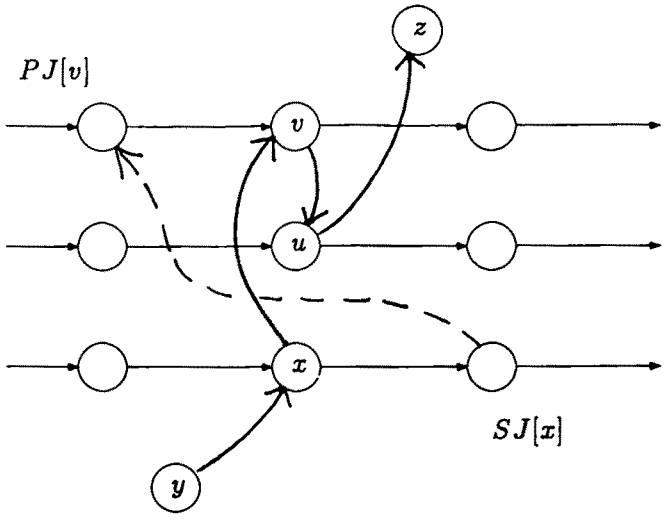

Fig. 3. Inversion of the only arc (u, v).

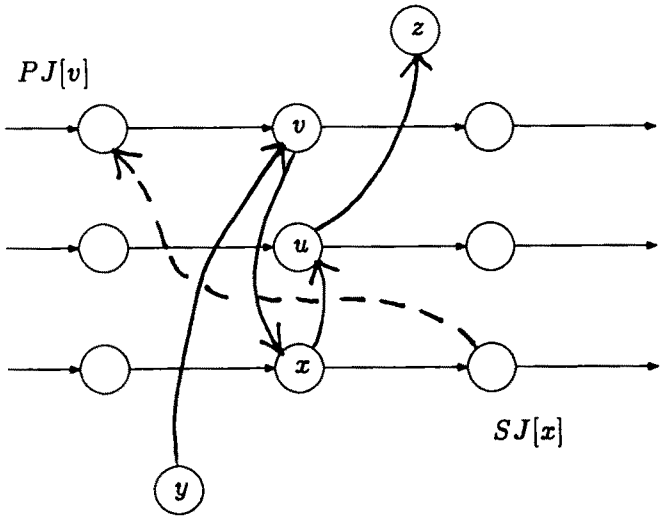

Fig. 4. Inversion of arcs (u, v) and (x, v).

It follows that $r'_u \leq r''_u$. Similarly,

$$t'_v = \max\{t'_u + d_u, t_{SJ[v]} + d_{SJ[v]}\},$$

$$t''_v = \max\{t''_x + d_x, t_{SJ[v]} + d_{SJ[v]}\},$$

$$t''_x \geq t''_u + d_u = t'_u + d_u$$

and so $t'_v \leq t''_v$. From the above observations, it follows that:

$$\begin{aligned} lpath(2, \{v, u\}) &= \max\{(r'_v + d_v + t'_v), (r'_u + d_u + t'_u)\} \\ &\leq \max\{(r''_v + d_v + t''_v), (r''_u + d_u + t''_u)\} \\ &\leq \max\{(r''_v + d_v + t''_v), (r''_u + d_u + t''_u), (r''_x + d_x + t''_x)\} \\ &= lpath(3, \{v, x, u\}). \end{aligned}$$ □

The analysis of neighborhood *NA* can be done in time $O(N)$ since each call to procedure *lpath* requires a constant time for $q \in \{2, 3\}$ and no more than N arcs can belong to the critical path.

The restricted version *RNA* follows from the same considerations reported in [12]. In *RNA*, arc (v, w) is not considered as *candidate* when both $(PM[v], v)$ and $(w, SM[w])$ are on a longest path in the current solution. In this way, one of the two tests in our procedure *estim* is always false and we have to compute for each candidate arc three estimations at most.

With a slight modification to the counterexample used for neighborhood *N2*, we can prove that the connectivity property does not hold even for *RNA*.

The second neighborhood we propose is called *NB*. Once two operations are defined *adjacent* so that the completion time of the first is equal to the starting time of the second, we call *block* a maximal sequence of adjacent operations to be processed on the same machine and belonging to a critical path of the current solution s. For all operations x in a block, we consider as neighboring s those solutions that can be generated by moving x in the first or in the last position of the block to which it belongs, if the corresponding solution is feasible, otherwise moving x in the position inside the block closest to the first or last one and such that the feasiblity is preserved. The estimation of the new sequences can be computed directly using the procedure *lpath*, while an exact feasibility test can be done in $O(N)$ with the labeling algorithm described in [1]. For computational reasons, we preferred to adopt a simpler but not exact test. Given an operation x in block $b = (b', x, b'')$, we show how to determine the possible infeasibility of the sequence (x, b', b''); a similar method can be adopted for the other sequences. The set of arcs reverted is $\{(x, i) : i \in b'\}$ and a cycle in the new solution can exist if and only if there is a path from $SJ[k]$, $k \in b'$ to $PJ[x]$. If such a path exists, then operation $SJ[k]$ must be scheduled before operation $PJ[x]$ and the inequality $C_{SJ[k]} \leq C_{PJ[x]} - d_{PJ[x]}$ must be satisfied. Testing the above condition for all $k \in b'$ prevents the generation of infeasible solutions, but it forbids legal moves too. However, these moves are, at best, not improving ones and the probability they were selected is very small.

For the neighborhood *NB*, we can prove that the connectivity property holds.

THEOREM 4.4

For each feasible solution s, it is possible to construct a finite sequence of moves, using NB, which will lead from s to a globally minimal solution.

Proof

Let s be the current feasible solution, $cp = \{b_1, b_2, \ldots, b_q\}$ a critical path in s, of q blocks (where a block is defined as above) and let f_i and l_i be the first and the last operation in b_i $(i = 1, \ldots q)$. Since path cp is critical, it is $(l_i, f_{i+1}) \in A (i = 1, \ldots, q-1)$, while $(k, h) \in E (k, h \in b_i)$. Given the globally optimal solution s_{opt}, if the operations $\{b_i \backslash \{f_i, l_i\}\}$ are scheduled between f_i and l_i in s_{opt} (for $i = 1, \ldots, q$), then the current solution is optimal too, since the length of the critical path in s_{opt} can not be less than the length of cp. Otherwise, at least a block b_i and an operation $k \in b_i$ exist such that the arc (f_i, k) or (k, l_i) is reversed in s_{opt}. We consider the case in which (f_i, k) is reversed; a similar argument can be used for the other case. We claim that at least a move in NB exists which strictly decreases the number of arcs with different orientation (*distance*) in s and s_{opt}. Let $k \in b_i$ be the first operation such that arc (f_i, k) is reversed in s_{opt}. If the solution obtained, scheduling operation k in the first block b_i, is feasible we can produce such a solution with a move in NB, and the distance between s and s_{opt} decreases. Otherwise, let $j \in b_i$ be the last operation preceding k such that the reversal of arc (j, k) and of all arcs (h, k), for each $h \in b_i$ scheduled between j and k, produces an infeasible solution. Let us call $\hat{b}$ the operations of block b_i between j and k. Observe that $\hat{b}$ contains at least one operation since a single swap inside b_i can never lead to an infeasible solution (property (1) of neighborhood $N1$). Since arc $(k, f_i) \in s_{opt}$ and the arcs $(h, f_i) \notin s_{opt}$, $h \in \hat{b}$ (by definition of k and $\hat{b}$), then k precedes any $h \in \hat{b}$ in s_{opt}. Therefore, the feasible sequence $(j, k, \hat{b})$ decreases the distance between s and s_{opt}, but this is one of the sequences we can produce with a move in NB. □

Finally, we propose a neighborhood structure, denoted NC later on, which combines the ideas of neighborhoods RNA and NB. Given a feasible solution s, the set $NC(s)$ is equal to $RNA(s) \cup NB(s)$. For this neighborhood, the connectivity property trivially holds.

5. Tabu list and strategies

The main components of a tabu-search algorithm are *memory structures*, in order to have a trace of the evolution of the search, and *strategies*, in order to use the memorized information in the best possible way. In the following, we are going to explain how we applied the tabu-search ideas to the job-shop scheduling problem.

The fundamental memory structure in tabu search is the so-called *tabu list*. We associate with each move leading from the current solution s to a solution $s' \in N(s)$ a set of *attributes*. We memorize in the tabu list the attributes of the

applied moves and, at each iteration, we select the best move among the set of candidates whose attributes do not belong to the tabu list. The tabu list has a finite dimension and we handle it with an *FIFO* strategy: at each iteration, the algorithm memorizes a new set of attributes and it *forgets* the oldest one. The basic idea is that we can avoid cycles in the evolution of the search by preventing the algorithm from repeating more recently made moves. With the neighborhood *NA*, a move consists in swapping one, two or three arcs involving two or three operations. Given a candidate arc (i, j) and the associated move, we memorize the attributes in the following way:

- if neither $SM[j]$ nor $PM[i]$ belong to a current critical path, then a move consists in the reversal of the only arc (i, j) and we memorize as forbidden the reversal of arc (j, i);
- if one of the nodes $SM[j]$ and $PM[i]$ belongs to a current critical path, then the move consists in the reversal of one, two or three arcs, but the estimation always considers the reversal of three arcs, so we memorize in the tabu list the reversal of all arcs considered.

In order to establish a candidate move tabu, we divide it into swaps of single arcs and we forbid it if at least one swap is currently tabu.

With the neighborhood *NB*, we consider each move as a sequence of elementary swaps which put operation x in the first (or last) position of its block. For each move, we memorize in the tabu list the sequence of single arcs swapped, forbidding their inversion. A candidate move is considered forbidden if its last swap is forbidden.

The data structure we used is conceptually the same as in [17], that is, a square matrix *TM* with dimensions equal to the maximum number of operations. The element $TM_{i,j}$ contains the count of the iteration in which the arc (i, j) has been reversed last time. We forbid a swap of arc (i, j) if the value of $TM_{j,i}$ plus the length of the tabu list is greater than the count of current iteration. Note that matrix *TM* is very sparse and it is convenient to memorize its information using specialized data structures in order to save memory.

In our implementation the length of the tabu list, i.e. the number of iterations a move maintains a tabu status, varies according to the following rules:

- if the current objective function value is less than the best value found before, then set the list length to 1;
- if we are in an improving phase of the search (i.e. the value of the objective function of the current solution is less than the value at the previous iteration) and the length of the list is greater than a threshold *min*, then decrease the list length by one unit;
- if we are not in an improving phase of the search (i.e. the value of the objective function of the current solution is greater than or equal to the value at the previous iteration) and the length of the list is less than a given *max*, then increase the list length by one unit.

If the value of the threshold *min* is too large, it can lead to forbid too many moves at each iteration; on the other hand, if the value of *min* is too small, the search of the algorithm can be trapped into cycles. In a cyclic situation, a solution $s(i)$ after i iterations is equal to a solution $s(i+k)$ after $i+k$ iterations, for some positive k, and the algorithm selects the same move in order to go from $s(i)$ to $s(i+1)$ or from $s(i+k)$ to $s(i+k+1)$. In order to detect this cycle, we select a *witness* arc for each move and associate with it the value of the current objective function. Before making a move, we determine the witness of the move and test if the current value of the objective function is equal to the value associated with the witness (i.e. the value of the last solution which was modified with a move having the same witness). If the test is true for the same arc for more than *Tcycle* consecutive iterations, we suppose we are in a cycle.

Another usual strategy in tabu search is the definition of an *aspiration criterion*, that is, a rule which cancels the effect of a tabu status on a move in particular situations. In our implementation, we adopt the following basic aspiration criterion:

- make a forbidden move candidate if its estimation is less than the value of the best solution found before the current iteration.

Besides the fact that tabu search in its *canonical* form [7] does not make randomized choices, we introduced randomization in our algorithm motivated by computational experiences. One randomized step was introduced to solve the "critical" situations occurring when, at a given iteration, all possible moves belong to the tabu list and not one of them satisfies the aspiration criterion. In this case, we randomly select a move among the possible ones. A second kind of critical decision concerns the length of the tabu list and, in our algorithm, the values *min* and *max* used to determine this length. Every Λ iterations, we randomly choose the value *min* between two positive integers a and b, and the value *max* between the positive integers A and B.

In order to improve the performance of our algorithm, we adopted one more strategy: *restarting*. Given a positive integer Δ, restarting is the following:

- if in the last Δ iterations the algorithm did not improve the best solution found up to the current iteration, then set the current solution to the best one.

This is quite a simple way to memorize in a long-term fashion the attributes associated with a good solution. Note that we can apply this strategy since our method does not have a completely deterministic behavior.

Our stopping rule is based on a limit to the maximum number of iterations and on the restarting strategy just explained. The algorithm continues to execute a series of Δ iterations and terminates if the more recent improvement to the best solution found did not occur during the last Δ iterations and the current iteration count is greater than or equal to a global parameter *Maxiter*.

6. Computational results and final remarks

6.1. RESULTS

We coded algorithms *Bidir* and *TS* in PASCAL and ran them on a PC 386 with a clock of 33 MHz. Algorithm *TS* was implemented with neighborhood *NC*, according to sections 2, 4, 5.

For the parameters of the procedures, we used the following values.

The parameter c of procedure *Bidir* was set to 3 (with a smaller value, the procedure generates too narrow a range of feasible solutions, and with values larger than 3, the quality of the solutions worsens, on average). For the tabu search, we set the restarting parameter Δ to 800 and the global number of iterations *Maxiter* to 1200 ($=\Delta \times 15$).

The parameter *Tcycle* was set to 3 (with this value, the algorithm can detect cyclic behavior quite early).

The parameter Λ was set to 60, while for the values a, b, A and B, we applied the following rule: $a := 2$, $b := a + \lfloor \frac{n+m}{3} \rfloor$; $A := min + 6$; $B := A + \lfloor \frac{n+m}{3} \rfloor$.

We tested our algorithms on a set of 53 problem instances: three from Muth and Thompson [13] (MT6, MT10, MT20), five from Adams et al. [1] (ABZ5–9), forty from Lawrence [11] (LA01–40), and five from Applegate and Cook [2] (ORB1–5). The optimal solution value is known for 46 of the 53 instances, while problems ABZ7, ABZ8, ABZ9, LA21, LA27, LA29 and LA38 are still open.

The non-deterministic nature of our algorithms makes is necessry to carry out multiple runs on the same problem instance in order to obtain meaningful results. We ran algorithm *TS* five times from different starting solutions, each of which was determined by a run of procedure *Bidir*.

The computational results are given in tables 1–4; all times are in seconds of a PC 386. Tables 1 and 2 compare the behavior of procedure *Bidir* with five list schedule algorithms using the following priority rules:

- *SI* : select the job with the *shortest imminent operation time*;
- *LI* : select the job with the *longest imminent operation time*;
- *SR* : select the job with the *shortest remaining processing time*;
- *LR* : select the job with the *longest remaining processing time*;
- *LRM* : select the job with the *longest remaining processing time excluding the operation under consideration.*

Tables 1 and 2 contain the following information:

Opt(*LB*, *UB*) = optimum value if known, otherwise, in brackets, the best lower and upper bound found up to now;

n = number of jobs;

Table 1

Problem	n	m	Opt (LB, UB)	Z_B	T_B	Z_{LS}					T_{LS}
						SI	LI	SR	LR	LRM	
ABZ5	10	10	1234	1359	0.18	2073	1877	2093	1890	1890	0.01
ABZ6	10	10	943	1025	0.24	1500	1377	1397	1306	1250	0.02
ABZ7	20	15	(651, 681)	785	1.93	1117	985	1080	1019	1070	0.06
ABZ8	20	15	(627, 670)	804	1.96	1039	998	1014	1021	1029	0.08
ABZ9	20	15	(650, 706)	821	1.93	1094	1029	1159	1077	973	0.06
MT6	6	6	55	56	0.01	87	73	94	67	67	0.01
MT10	10	10	930	1076	0.16	1399	1534	1530	1272	1420	0.01
MT 20	20	5	1165	1310	0.27	1521	1610	1513	1544	1534	0.01
ORB1	10	10	1059	1281	0.17	1434	1583	1489	1391	1314	0.02
ORB2	10	10	888	1035	0.22	1303	1376	1370	1320	1241	0.02
ORB3	10	10	1005	1214	0.16	1405	1466	1463	1360	1327	0.03
ORB4	10	10	1005	1161	0.18	1760	1376	1556	1551	1464	0.02
ORB5	10	10	887	1049	0.20	1199	1251	1296	1180	1212	0.02

m = number of machines;

Z_B = value of the best solution found by *Bidir* out of five runs;

T_B = computing time for the five runs of procedure *Bidir*;

Z_{LS} = value of the solution found by each list schedule algorithm;

T_{LS} = computing time for the five list schedule algorithms.

Tables 1 and 2 show that algorithm *Bidir* always produces better results than the simple list schedule algorithms, but at the cost of greater computational times.

Tables 3 and 4 contain the following information:

$Opt(LB, UB), n, m$ = as in table 1;

Z_{best} = value of the best solution found by *TS* out of 5 runs;

Z_{av} = average solution value over 5 runs or nothing if all runs gave the optimum value:

$\Delta Z\%$ = $((Z_{best} - Opt)/Opt) \times 100$ if the optimum value is known, $((Z_{best} - LB)/LB) \times 100$ otherwise;

T_{av} = average computing time;

T_{max} = maximum computing time for one run.

Table 2

Problem	n	m	Opt (LB, UB)	Z_B	T_B	Z_{LS}					T_{LS}
						SI	LI	SR	LR	LRM	
LA01	10	5	666	724	0.01	859	889	1210	948	907	0.01
LA02	10	5	655	739	0.01	1080	858	966	882	882	0.01
LA03	10	5	597	710	0.01	770	748	897	843	927	0.01
LA04	10	5	590	664	0.05	895	848	976	930	815	0.01
LA05	10	5	593	593	0.06	827	787	905	689	712	0.01
LA06	15	5	926	940	0.22	1369	1105	1498	1120	1042	0.02
LA07	15	5	890	946	0.17	1128	1079	1282	1098	1062	0.01
LA08	15	5	863	984	0.16	1168	1102	1348	1003	1085	0.02
LA09	15	5	951	1017	0.11	1332	1111	1384	1280	1280	0.01
LA10	15	5	958	958	0.11	1312	1136	1509	1151	1087	0.01
LA11	20	5	1222	1259	0.33	1654	1472	1653	1387	1409	0.02
LA12	20	5	1039	1044	0.33	1328	1222	1622	1153	1239	0.01
LA13	20	5	1150	1160	0.27	1747	1246	1517	1344	1401	0.01
LA14	20	5	1292	1294	0.33	1757	1477	1679	1388	1357	0.03
LA15	20	5	1207	1304	0.33	1538	1516	1900	1508	1504	0.01
LA16	10	10	945	1045	0.16	1480	1244	1371	1262	1212	0.01
LA17	10	10	784	838	0.17	1141	1157	1416	1167	958	0.02
LA18	10	10	848	973	0.11	1259	1264	1353	1117	1125	0.01
LA19	10	10	842	941	0.22	1352	1326	1302	1146	1370	0.03
LA20	10	10	902	1050	0.17	1331	1293	1447	1230	1230	0.01
LA21	15	10	(1040, 1050)	1210	0.44	1690	1519	1806	1452	1503	0.02
LA22	15	10	927	1087	0.44	1392	1409	1736	1312	1507	0.01
LA23	15	10	1032	1093	0.55	1681	1330	1526	1414	1662	0.01
LA24	15	10	935	1132	0.55	1521	1472	1623	1606	1435	0.02
LA25	15	10	977	1175	0.49	1691	1546	1849	1465	1270	0.05
LA26	20	10	1218	1355	0.93	1867	1746	1989	1918	1792	0.02
LA27	20	10	(1235, 1269)	1470	1.04	1964	1773	2161	1844	1860	0.05
LA28	20	10	1216	1415	1.03	1991	1668	2145	1867	1967	0.05
LA29	20	10	(1120, 1195)	1401	0.99	1969	1833	2154	1618	1633	0.02
LA30	20	10	1355	1493	0.88	2084	1809	2444	1823	1805	0.03
LA31	30	10	1784	1840	2.80	2352	2410	2783	2422	2403	0.01
LA32	30	10	1850	1928	2.74	2749	2544	2899	2464	2518	0.04
LA33	30	10	1719	1802	2.80	2329	2302	2514	2306	2346	0.06
LA34	30	10	1721	1837	2.86	2581	2480	2743	2464	2416	0.02
LA35	30	10	1888	1983	2.69	2406	2468	2842	2439	2397	0.09
LA36	15	15	1268	1383	0.76	2070	1789	2218	1920	1871	0.01
LA37	15	15	1397	1657	0.82	2086	1940	2176	1731	1940	0.01
LA38	15	15	(1171, 1184)	1362	0.83	1746	1841	1928	1752	1731	0.01
LA39	15	15	1233	1502	0.82	1790	2064	2102	2065	2080	0.01
LA40	15	15	1233	1389	0.88	2021	1829	2231	1726	1756	0.01

Table 3

Problem	n	m	Opt (LB, UB)	Z_{best}	$\Delta Z\%$	Z_{av}	T_{av}	T_{max}
ABZ5	10	10	1234	1236	0.16	1237.6	139.5	140.1
ABZ6	10	10	943	(3) 943	0	943.8	86.8	128.3
ABZ7	20	15	(651, 681)	667	2.45	675.6	320.1	328.8
ABZ8	20	15	(627, 670)	678	8.13	684.2	336.1	351.6
ABZ9	20	15	(650, 706)	692	6.46	700.2	320.8	325.5
MT6	6	6	55	(5) 55	0	–	2.4	6.6
MT10	10	10	930	935	0.53	948.4	155.8	156.6
MT20	20	5	1165	(4) 1165	0	1166.8	160.1	260.2
ORB1	10	10	1059	1064	0.47	1080.0	157.6	166.1
ORB2	10	10	888	889	0.11	893.4	136.4	137.7
ORB3	10	10	1005	1016	1.09	1032.2	157.3	160.5
ORB4	10	10	1005	1013	0.79	1018.2	156.8	158.8
ORB5	10	10	887	889	0.22	895.8	140.1	141.1

Tables 3 and 4 show that our tabu-search algorithm finds an optimal solution for 33 of 46 problems for which the optimal solution is known. For the 13 remaining problems, the distance from the optimum ($\Delta Z\%$) is smaller than one percent, with only one exception: problem ORB3, having $\Delta Z\% = 1.09$. Concerning the 7 *open* problems, algorithm *TS* improves the best known upper bound for 5 of them. The larger value of $\Delta Z\%$ associated with those instances is probably due to the lower bound value, which is generally not so close to the optimum. Comparing the value Z_{best} with value Z_{av}, we can see that algorithm *TS* is quite *robust* (i.e. the differences between the solution values of the runs are small) and the quality of the solutions is weakly dependent upon the random choices. The computing times are generally small (no run required more that 6 minutes) and increase almost linearly with the number of operations for "difficult" problems. Moreover, the behavior of algorithm *TS* is not different for square ($n = m$) or rectangular instances, or for different instances of the same shape. With regard to the neighborhood $N1$, we compared our algorithm with the serial version of the tabu search proposed by Taillard [17]. Our computational experience with that approach, with a time limit equal to T_{av} as a stopping criterion, showed that the neighborhood $N1$ leads to good solutions for squared instances, but definitely gives poor results for the "difficult" rectangular ones (i.e. MT20 and LA26–30). This is because with neighborhood $N1$ a local search algorithm spends a lot of time making unfruitful moves, i.e. changing the order of operations internal to critical sequences. On the other hand, neighborhood $N2$ has a weaker theoretical

Table 4

Problem	n	m	Opt (LB, UB)	Z_{best}	$\Delta Z\%$	Z_{av}	T_{av}	T_{max}
LA01	10	5	666	(5) 666	0	–	0.1	0.5
LA02	10	5	655	(5) 655	0	–	18.8	54.4
LA03	10	5	597	(4) 597	0	598.2	21.6	38.3
LA04	10	5	590	(5) 590	0	–	32.2	35.1
LA05	10	5	593	(5) 593	0	–	0.3	0.6
LA06	15	5	926	(5) 926	0	–	0.3	0.6
LA07	15	5	890	(5) 890	0	–	0.6	1.1
LA08	15	5	863	(5) 863	0	–	0.3	0.5
LA09	15	5	951	(5) 951	0	–	0.2	0.5
LA10	15	5	958	(5) 958	0	–	0.2	0.4
LA11	20	5	1222	(5) 1222	0	–	0.4	0.5
LA12	20	5	1039	(5) 1039	0	–	0.2	0.6
LA13	20	5	1150	(5) 1150	0	–	0.4	0.5
LA14	20	5	1292	(5) 1292	0	–	0.4	0.6
LA15	20	5	1207	(5) 1207	0	–	1.2	4.4
LA16	10	10	945	(2) 945	0	946.0	97.4	157.2
LA17	10	10	784	(5) 784	0	–	21.7	32.7
LA18	10	10	848	(5) 848	0	–	63.1	90.5
LA19	10	10	842	(1) 842	0	846.6	103.8	128.8
LA20	10	10	902	(4) 902	0	903.0	71.7	132.7
LA21	15	10	(1040, 1050)	1048	0.76	1057.0	198.8	204.3
LA22	15	10	927	933	0.64	936.6	191.4	202.2
LA23	15	10	1032	(5) 1032	0	–	1.8	4.4
LA24	15	10	935	941	0.64	943.8	181.8	183.3
LA25	15	10	977	979	0.20	980.4	191.7	193.3
LA26	20	10	1218	(5) 1218	0	–	22.1	34.4
LA27	20	10	(1235, 1269)	1242	0.56	1252.4	254.2	256.6
LA28	20	10	1216	(3) 1216	0	1216.8	186.4	260.5
LA29	20	10	(1120, 1195)	1182	5.53	1194.6	281.3	311.6
LA30	20	10	1355	(5) 1355	0	–	10.4	18.8
LA31	30	10	1784	(5) 1784	0	–	2.1	2.3
LA32	30	10	1850	(5) 1850	0	–	2.2	2.9
LA33	30	10	1719	(5) 1719	0	–	1.8	2.4
LA34	30	10	1721	(5) 1721	0	–	5.1	7.8
LA35	30	10	1888	(5) 1888	0	–	1.3	2.4
LA36	15	15	1268	1278	0.78	1289.4	238.4	240.1
LA37	15	15	1397	1409	0.85	1423.0	242.2	250.5
LA38	15	15	(1171, 1184)	1203	2.73	1210.0	256.6	291.6
LA39	15	15	1233	1242	0.72	1254.8	237.8	266.6
LA40	15	15	1233	(1) 1233	0	1235.4	236.6	250.5

support (the connectivity property does not hold) and furthermore, a tabu-search algorithm, both with the complete version of $N2$ as proposed in [12] and with the restricted one (only one arc reversed at each iteration), leads to good solutions for rectangular and square instances of small size and it becomes increasingly worse than our method for instances of larger size (i.e. LA26–30 and LA36–40).

The simulated annealing approach of Van Laarhoven et al. [18] produces solutions with values comparable to the one of *TS* only with very large computation times (greater than 5200 seconds of a VAX-785 for the instances LA36–40). The simulated annealing algorithms of Matsuo et al. [12], which is a spurious implementation of this approach mixed with a local search technique, is always dominated by *TS* with the only exception of instance LA22.

With regard to computation times, the best branch and bound algorithms presented in the literature show a strong dependence on the instance, for problems of the same size, and on the shape (all but one of the seven open problems have $n > m$). Moreover, the computation times become enormous for some of the instances LA21–30 and LA36–40 (more than 240,000 seconds on a Sun 4/20 workstation).

6.2. FINAL REMARKS

In this paper, we have presented a tabu-search based algorithm for solving the job-shop scheduling problem and a new heuristic procedure for generating feasible starting solutions. Our approach uses two new powerful neighboring structures for which we proved the connectivity property, which is that, given any feasible solution, there always exists a sequence of transitions inside the neighborhood which leads to a global optimum.

We ran our algorithm on a set of 53 benchmark problem instances. In many cases we found the optimal solution, and in others the distance from the optimum was very small, exhibiting a robustness not common to previously presented local search based algorithms or exact branch and bound methods. The computing times required by our algorithm are small, never exceeding 6 minutes for a single run.

We believe that it is possible to tune the algorithm in order to increase the number of optimal solutions found maintaining a low threshold to the maximum computation time allowed. There are two research directions: decreasing the level of randomization and increasing the use of memory structures of the long-term type.

Acknowledgements

We are indebted to Professor Francesco Maffioli and Professor Jan Karel Lenstra for useful suggestions, and to Professor William Cook who provided us with data sets of job-shop test problems.

References

[1] J. Adams, E. Balas and D. Zawack, The shifting bottleneck procedure for job-shop scheduling, Manag. Sci. 34(1988)391–401.

[2] D. Applegate and W. Cook, A computational study of the job-shop scheduling problem, ORSA J. Comput. 3(1990)149–156.
[3] P. Brucker, B. Jurisch and B. Sievers, A fast branch and bound algorithm for the job-shop scheduling problem, Internal Report 136, Fachbereich Mathematik/Informatik, Universität Osnabrück (1991).
[4] J. Carlier and E. Pinson, An algorithm for solving the job-shop problem, Manag. Sci. 35(1989) 164–176.
[5] T. Feo, K. Venkatraman and J.F. Bard, A GRASP for single machine scheduling with due dates and earliness penalties, Internal Report, OR Group, Department of Mechanical Engineering, University of Texas, Austin, TX (1989).
[6] M.R. Garey, D.S. Johnson and R. Sethi, The complexity of flowshop and jobshop scheduling, Math. Oper. Res. 1(1976)117–129.
[7] F. Glover, Tabu search, Part I, ORSA J. Comput. 1(1989)190–206.
[8] F. Glover, Tabu search, Part II, ORSA J. Comput. 2(1990)4–32.
[9] R.L. Graham, E.L. Lawler, J.K. Lenstra and A.H.G. Rinnooy Kan, Optimization and approximation in deterministic sequencing and scheduling: a survey, Ann. Discr. Math. 5(1979)287–326.
[10] J.P. Hart and A.W. Shogan, Semi-greedy heuristics: an empirical study, Oper. Res. Lett . 6(1987) 107–114.
[11] S. Lawrence, Resource constrained project scheduling: an experimental investigation of heuristic scheduling techniques, GSIA, Carnegie-Mellon University (1984).
[12] H. Matsuo, C.J. Suh and R.S. Sullivan, A controlled search simulated annealing method for the general jobshop scheduling problem, Working Paper 03-44-88, Graduate School of Business, University of Texas, Austin, TX (1988).
[13] J.F. Muth and G.L. Thompson, *Industrial Scheduling* (Prentice–Hall, Englewood Cliffs, 1963).
[14] R. Nakano and T. Yamada, Conventional genetic algorithm for job shop problems, *Proc. 4th Int. Conf. on Geneting Algorithms*, San Diego, CA (1991) pp. 474–479.
[15] S.S. Panwalker and W. Iskander, A survey of scheduling rules, Oper. Res. 25(1977)45–61.
[16] B. Roy and B. Sussmann, Les Problèmes d'ordonnancement avec contraintes disjonctives, Note DS n.9 bis, SEMA, Montrouge (1964).
[17] E. Taillard, Parallel taboo search technique for the jobshop scheduling problem, Internal Report ORWP 89/11, Département de Mathématiques, Ecole Polytechnique Fédérale de Lausanne, Lausanne (1989).
[18] P.J.M. van Laarhoven, E.H.L. Aarts and J.K. Lenstra, Job shop scheduling by simulated annealing, Report OS-R8809, Centre for Mathematics and Computer Science, Amsterdam (1988).
[19] S.M. Johnson, Optimal two- and three-stage production schedules with setup times included, Naval Res. Logist. Quart. 1(1954)61–68.