

Using intelligent backtracking to improve  
Branch-and-bound methods:  
an application to Open-Shop problems

Christelle GUÉRET, Narendra JUSSIEN

and Christian PRINS

École des Mines de Nantes

La Chantrerie – 4 Rue Alfred Kastler

BP20722

F-44307 Nantes Cedex 03, FRANCE

email : `Christelle.Gueret@emn.fr`, `Narendra.Jussien@emn.fr`,  
`Christian.Prins@emn.fr`

**Abstract**

Only two branch-and-bound methods have been published so far for the Open-Shop problem. The best one has been proposed by Brucker *et al.* But some square problems from size 7 are still unsolved by it.

We present an improving technique for branch-and-bound methods applied to Brucker *et al.* algorithm for Open-Shop problems. Our technique is based on intelligent backtracking. Intelligent backtracking techniques involve the computation of explanations when encountering a contradiction in the search. We present here an adaptation of a generic explanation system we have initially developed in the constraint programming scheme.

We tested our approach on Open-Shop problems from the literature (benchmarks of Taillard). The search is definitely improved : on some square problems of size 10, the number of explored nodes is reduced by more than 90% and we solved an open problem of size 10.

We applied our technique on Open-Shop problems, but it can easily be adapted to solve any other shop problems with a Branch-and-Bound in which the branching scheme consists in fixing disjunctions.

**Keywords.** Scheduling theory, Open-Shop, Branch-and-bound, Intelli-

## 1 Introduction

In the Open-Shop problem, a set  $J$  of  $n$  jobs, consisting each of  $m$  operations (or tasks), must be processed on a set  $M$  of  $m$  machines. The processing times are given by a matrix  $P : m \times n$ , in which  $p_{ij}$  is the duration of operation  $O_{ij}$  of job  $J_j$ , to be done on machine  $M_i$ . The operations of a job can be processed in any order, but only one at a time. We consider the construction of non-preemptive schedules of minimal makespan, which is NP-Hard for  $m \geq 3$  [Gonzalez and Sahni, 1976].

Only two branch-and-bound methods for that problem have been published so far. The first one [Brucker *et al.*, 1996] is based on the resolution of a one-machine problem with release dates and due dates. The second, [Brucker *et al.*, 1994a], consists, in each node, in fixing disjunctions on the critical path of a heuristic solution. Although that last method is the best known until now, some problems from size 7 are still unsolved.

In this paper, we introduce an improving technique for branch-and-bound for shop problems in which the branching scheme consists in adding precedence constraints. We present the application to Brucker *et al.*'s algorithm for Open-Shop problems. Our technique is based on intelligent backtracking: when a node is eliminated, instead of systematically backtracking to the parent node (chronological backtracking), we backtrack higher in the search tree, without losing any optimal solution.

Our paper is organized as follows: section 2 presents the branch-and-bound method of Brucker *et al.* In section 3 we introduce our proposal to reduce the number of nodes developed. Then, section 4 describes how to apply the ideas of section 3 to the branch-and-bound algorithm of Brucker *et al.* Section 5 provides a computational evaluation of our new branch-and-bound method on benchmarks given by Taillard [1993].

## 2 The branch-and-bound algorithm of Brucker *et al.*

The branch-and-bound of Brucker *et al.* is a depth-first search algorithm. It combines two concepts:

- a generalization of a branching scheme first introduced by Grabowski *et al.* [1986] for one-machine problems with release dates and due dates;
- *immediate selections*, a method initially designed to fix disjunctions in Job-Shop problems by Carlier and Pinson [1989; 1994].

We briefly present this branch-and-bound algorithm. The details can be found in [Brucker *et al.*, 1994a] and [Brucker *et al.*, 1994b]. We first recall some basic concepts and vocabulary that are used throughout this paper.

## 2.1 Disjunctive graph, heads and tails

### Definition 1 (Disjunctive graph)

An instance of the Open-Shop problem can be represented by a disjunctive graph  $G = (X, U, D)$  where:

- $X$  is the set of vertices which correspond to the tasks of the problem. Two fictitious tasks 0 and  $n \times m + 1$ , respectively the starting task and the ending task of the schedule, are added to this set;
- $U$  is the set of conjunctive arcs. There exists a conjunctive arc  $(i, j)$  between two tasks  $i$  and  $j$  if  $i$  must be executed before  $j$ ; Note that initially the only conjunctive arcs are the ones originating from node 0 or ending at node  $n \times m + 1$ .
- $D$  is a set of disjunctions which express that two tasks cannot overlap (for example two tasks of a same job or of a same machine). A disjunction between two tasks  $i$  and  $j$  is represented by an edge  $[i, j]$ .

To obtain a schedule from graph  $G$ , one has to fix disjunctions, *i.e.* replace each disjunctive edge  $[i, j]$  by an arc  $(i, j)$  or  $(j, i)$  such that the resulting graph is acyclic. The makespan of the schedule is then the length of a longest path between 0 and  $n \times m + 1$  in this graph. Such a path is called a critical path.

In a graph in which some disjunctions have been fixed :

- $r_i$  denotes the *head* (or release date) of task  $i$ .  $r_i$  corresponds to the length of a longest path from 0 to  $i$ ;
- $f_i$  is the latest start time of  $i$ .  $f_i$  is the length of a longest path from  $i$  to task  $n \times m + 1$ ;
- $q_i$  is the *tail* of task  $i$ .  $q_i$  is the length of a longest path from the ending date of  $i$  to task  $n \times m + 1$ .

Each time a disjunction is fixed, heads and tails must be recomputed. A simple way to recalculate them is to use for example Bellman's algorithm [Bellman, 1958].

To obtain better heads, the following relation can be used for each task  $i$ , where  $\Gamma_M^{-1}(i)$  is the set of predecessors of  $i$  executed on the same machine as  $i$ , and  $\Gamma_J^{-1}(i)$  is the set of predecessors of  $i$  belonging to the same job as  $i$ :

$$r_i \leftarrow \max \left\{ \begin{array}{l} r_i, \\ \min_{j \in \Gamma_M^{-1}(i)} r_j + \sum_{j \in \Gamma_M^{-1}(i)} p_j, \max_{j \in \Gamma_M^{-1}(i)} (r_j + p_j), \\ \min_{j \in \Gamma_J^{-1}(i)} r_j + \sum_{j \in \Gamma_J^{-1}(i)} p_j, \max_{j \in \Gamma_J^{-1}(i)} (r_j + p_j) \end{array} \right\}$$

This relation comes from the fact that tasks belonging to the same job or executed on the same machine cannot overlap. Symmetrical equations can be derived for computing better tails.

## 2.2 Branching scheme

The branching scheme generalises an approach used by Grabowski *et al.* [1986] to solve one-machine problems with release dates and due dates. It is based on the computation of a heuristic solution in each node. Some disjunctions are fixed on a critical path of this solution.

The heuristic used has been developed by Bräsel *et al.* [1991]. This heuristic associates to the Open-Shop problem a bipartite-graph  $G = (X \cup Y, E, W)$  in which  $X$  is the set of machines,  $Y$  the set of jobs, and each edge  $[i, j]$  in  $E$  of weight  $w_{ij} = p_{ij}$  represents the task  $O_{ij}$ . The heuristic decomposes this bipartite-graph into a sequence of maximum cardinality matchings. As each matching is a set of tasks which can be executed at the same time, a heuristic solution can be obtained by concatenating the partial schedules defined by these sets of tasks in their order of construction.

Several matching algorithms can be used: min-max matching, max-min matching, min-weight matching ...

This heuristic is designed for problems without precedence constraints. It can easily be adapted for problems in which some disjunctions have been fixed [Brucker *et al.*, 1994a].

Once the heuristic solution is obtained, a critical path  $\mu$  is computed. Then the *blocks* of this critical path are determined:

**Definition 2 (Block [Brucker *et al.*, 1994a])**

A sequence  $u_1, u_2, \dots, u_l$  of successive nodes in  $\mu$  is called a *block* on  $\mu$  if the following properties hold:

1. The sequence represents a set of operations which either have to be processed on the same machine or belong to the same job.
2. Enlarging the sequence by one operation yields a sequence which does not fulfill the first property.

The branching scheme is based on the following theorem:

**Theorem 1 ([Brucker et al., 1994a])**

Let  $S$  be a heuristic solution and  $\mu$  a critical path of  $S$ . If there exists a better solution  $S'$ , then in  $S'$  at least one operation of one block of  $\mu$  has to be processed before the first or after the last operation of this block.

The idea is then to create two children nodes for each operation of each block by moving it before or after all the other operations of the corresponding block.

### 2.3 Immediate selections

*Immediate selections* [Carlier and Pinson, 1989] are elimination rules which fix additional disjunctions between tasks belonging to the same job or to be processed on the same machine.

Consider a task  $c$  and a set of tasks  $I$  all executed on the same machine (or which belong to the same job as  $c$ ). Let  $J$  be a subset of  $I$  and  $UB$  the makespan of the current solution. *Immediate selections* are based on the following proposition:

**Proposition 1 ([Carlier and Pinson, 1989])**

If

$$r_c + p_c + \sum_{j \in J} p_j + \min_{j \in J} q_j \geq UB$$

and

$$\min_{j \in J} r_j + \sum_{j \in J} p_j + p_c + \min_{j \in J} q_j \geq UB$$

then in all solutions better than  $UB$ ,  $c$  has to be processed after all operations of  $J$ .  $J$  is called *ascendant set* of  $c$ .

The first relation is a lower bound of an optimal schedule of the tasks of  $J \cup \{c\}$  in which  $c$  is executed before all the tasks of  $J$ . The second relation is a lower bound of an optimal schedule of the same set of tasks but in which  $c$  is not the first nor the last task of the schedule. If these two lower bounds are greater than  $UB$ , then  $c$  must be executed after all the tasks of  $J$  in all solution better than  $UB$ .

In this case, the head of  $c$  can be adjusted by:

$r_c \leftarrow \max(r_c, C^*)$  where

$$C^* = \max_{J' \subset J} \left\{ \min_{j \in J'} r_j + \sum_{j \in J'} p_j \right\}$$

In the case  $|J| = 1$ , Proposition 1 becomes:

**Proposition 2**

If  $r_c + p_c + p_i + q_i \geq UB$  then  $c$  has to be processed after  $i$ .

The disjunctive arc  $i \rightarrow c$  can then be fixed, and operation  $c$  cannot start before:

$$r_c \leftarrow \max(r_c, r_i + p_i).$$

These *immediate selections* are called *immediate selections on disjunctions*. The previous ones are named *immediate selections on sets*.

Remark : for efficiency reason, the ascendant set  $J$  is not explicitly determined by the algorithm used to compute immediate selections on sets. For this reason, it is not possible to fix new disjunctions between  $c$  and the tasks of  $J$ ; this algorithm only adjust the head of  $c$ . But these disjunctions will be fixed later by immediate selections on disjunctions.

In this version, called primal version, we adjust heads. Because of the symmetric role of heads and tails, a symmetric version (the dual version) can be used to adjust tails.

### 3 Improving the search

Depth-first search branch-and-bound are based upon the use of a chronological backtracking algorithm to explore the search tree. Such an exploration can lead to useless branch tests.

The aim of this section is to show that an *intelligent* behavior may be incorporated into the backtracking algorithm in order to avoid useless explorations. Such a technique will therefore improve the search by cutting down exploration times.

#### 3.1 Example

Consider a small example that will clearly show the interest of improved backtracking techniques.

Suppose that we want to solve a job-shop problem (Note that a job-shop is used here because many precedences are already defined. This paper is clearly dedicated to open-shop problems.) involving 3 machines and 2 jobs *i.e.* we want to determine which job is scheduled before the other on each machine. The binary search tree of

(cf. Figure 1) in which a disjunction is fixed in each node can be used to solve this problem. The optimum is 20 (the circled leaf in the figure).

Suppose that a heuristic solution has been found (with value 25 therefore giving an upper bound) but no lower bound has been computed.

Let us have a close look on the search. Our search will start from the node labeled 180 in Figure 1. This node exceeds the upper bound, it is therefore not a good solution. Since the upper bound was respected in its father node, the reason of the non optimality of the current node is the choice  $J_1 \rightarrow J_2$  on machine 3. The next explored node is then the one labeled 100.

The upper bound is once more exceeded. For the same reason as above, it is because of choice  $J_2 \rightarrow J_1$  on machine 3. There is no other possibility to be tested. A classical approach will therefore undo the parent node. But suppose here that only considering  $J_1 \rightarrow J_2$  on machine 1 together with either  $J_1 \rightarrow J_2$  or  $J_2 \rightarrow J_1$  on machine 3 exceeds the upper bound. Then, we can deduce that  $J_1 \rightarrow J_2$  on machine 1 is not a good choice. Therefore there is no need to test  $J_2 \rightarrow J_1$  on machine 2 as a standard backtracking would do : the search can directly proceed to the exploration of the branch where  $J_2 \rightarrow J_1$  on machine 1, thus saving the exploration of 3 nodes.

This situation is due to the fact that, in spite of their sophistication, branching schemes may take completely independant decisions along a branch, thus creating disconnected sub-problems.

The remainder of the paper will present a technique that allows an automatic detection of such situations without any *a priori* analysis.

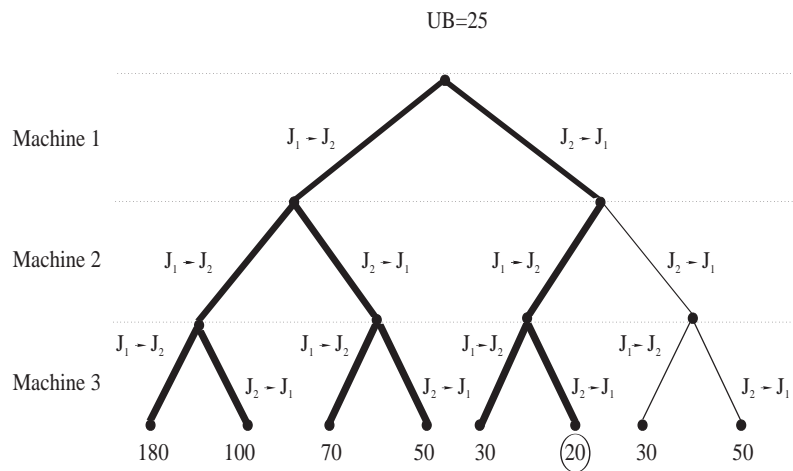


Figure 1: Binary search tree for a  $3 \times 2$  job-shop problem

### 3.2 Intelligent backtracking techniques

The idea of *intelligent backtracking* has been used in the Logic Programming community ([Bruynooghe and Pereira, 1984] or [Cox, 1984]) where the search is systematically done through backtracking.

Considering intelligent backtracking when using constraints has been introduced in the Constraint Programming community: either for *Constraint Satisfaction Problems* [Dechter, 1990] or for Constraint Logic Programming (Logic Programming with constraint propagation) [de Backer and Béringer, 1991]. In the latter, an intelligent backtracking mechanism is proposed to be used with an incremental simplex solving sets of linear constraints.

All *intelligent backtracking* techniques involve the computation of *explanations* (also called *nogoods*) when encountering a contradiction in the search (which leads to actually backtrack). An **explanation** for a contradiction is a subset of decisions (constraints) whose conjunction leads to a contradiction.

In previous work [Jussien and Boizumault, 1997a], we presented a generic explanation system to be used with constraint programming.

### 3.3 Our approach

Recall that constraint programming considers sets of variables taking their value in their respective domain (a set of allowed values) upon which is posted a set of constraints. Solving is based on domain filtering: removal of values (or tuples of values) that cannot appear in any solution of the problem. Since the underlying problem is NP-complete, only partial filtering is done *i.e.* the filtering algorithm cannot produce the set of solutions in the general case. In order to obtain a solution, an enumeration step is required which involves a tree-based search.

- **Recording *explanations***

The main idea of our explanation system is to maintain *explanations* not only for contradictions but also for all activities of the used solver *i.e.* for value removal from the domain of the variables (domain reduction).

In this approach, an **explanation** can be defined as a set of nodes such that the conjunctions of the decisions made in them leads to the associate conclusion (contradiction or value removal) *i.e.* this conclusion will be reached as soon as the considered decisions are made.

- **Using the explanations**

First of all, contradiction explanations can be computed from value removal explanations. As we said before, a contradiction occurs when the domain of a variable is emptied by the filtering algorithm. Since all the value removals have an explanation, the union of the value removal explanations for each value of the emptied domain is obviously a contradiction explanation.

There is another condition of contradiction. When all alternatives for a node have been unsuccessfully tested, it means that the set of constraints of that node is contradictory. The current node is clearly an explanation for the contradiction but a more precise information may be computed : when an alternative is tested, it is rejected if and only if its integration leads to a contradiction. This contradiction has an explanation. As for the first contradiction explanation defined above, the union of the explanations for each unsuccessful alternative is obviously an explanation for the unsuccessful node (actually, one considers this union minus the considered node.)

Let  $C_e$  be a contradiction explanation. Let  $r$  be the most recent node involved in  $C_e$ . Obviously, backtracking to any node between  $r$  and the current node is useless since all the decisions made in the nodes of  $C_e$  will remain active leading to a contradiction. Backtracking can thus be performed directly towards  $r$  saving more or less work. The set  $C_e \setminus \{r\}$  is then an explanation for the non selection of the alternative in node  $r$ .

Note that, as we informally saw in the example of section 3.1, no improvement will be made until all the possibilities have been tested for a given node  $r$ . Indeed the most recent reason for a contradiction in a child node will be the newly added constraints, and thus will induce backtracking to the parent node  $r$ . But as soon as all the possibilities have been tested for  $r$ , a true improvement may be reached by analysing the explanations of all its unsuccessful children.

## 4 Application

We explain now how to adapt our approach to the Branch-and-Bound of Brucker *et al.* As we just saw, the basic idea of our approach is to record information for each value removal from the domains of the variables.

In the Open-Shop problem, the variables are the starting dates  $t_i$  of each task  $i$ . The domain associated to each variable  $t_i$  is the interval  $[r_i, f_i]$ , where  $r_i$  is the

release date of task  $i$ , and  $f_i$  its latest start time. As a  $r_i$  (resp.  $f_i$ ) can only increase (resp. decrease) during the search, each time a  $r_i$  or a  $f_i$  is modified, some values are removed from the domain of task  $i$ . Then, each time a head  $r_i$  (resp. a latest start time  $f_i$ ) is modified, we will record all the nodes whose conjunction is responsible for this modification.

The explanations for  $r_i$  and  $f_i$  are the same than the one of  $r_i$  and  $q_i$ , since  $q_i = UB - f_i - p_i$ . In order to be consistent with *immediate selections*, modifications of  $r_i$  and  $q_i$  will be considered.

A modification of a head or a tail may be derived from the addition of a new precedence constraint. And a new precedence constraint may be created because of the decisions taken in some nodes. So, in order to record explanations for each modifications of heads and tails, the explanation of each fixed disjunction  $(i, j)$  must be recorded.

## 4.1 Recording explanations

We present now how to record explanations for fixed disjunctions and heads. Updating of explanations for tails is not explained since the method is symmetrical.

At root node, all explanations are initialized with  $\{root\}$  (root node). Then, each time an event occurs (a head or a tail is modified, a disjunction is fixed), the associated explanation is updated. We consider the different situations when such event may occur and explain how to update explanations :

- **Recording explanations for fixed disjunctions**

1. **Branching**

Suppose that a disjunction  $i \rightarrow j$  is fixed by the branching scheme in a new node  $r$  in the search tree.

If this precedence constraint does not exist, then its explanation is the current node ( $r$ ). So the explanation of the arc  $(i, j)$  is:

$$Expl(i, j) \leftarrow \{r\}$$

If  $i \rightarrow j$  already exists, its explanation does not change.

2. **Immediate selections on disjunctions**

Consider two tasks  $i$  and  $j$  executed on the same machine (in the case when these two tasks belong to the same job, the results can be easily deduced).

If  $r_j + p_j + p_i + q_i \geq UB$  then *immediate selections on disjunctions* fix the disjunction  $i \rightarrow j$ .

If this new arc does not exist, it is created because of the values of  $r_j$  and  $q_i$ . Its explanation becomes:

$$Expl(i, j) \leftarrow Expl(r_j) \cup Expl(q_i)$$

If this arc already exists, its explanation does not change.

- **Recording explanations for heads and tails**

1. **Immediate selections on disjunctions**

If two tasks  $i$  and  $j$  executed on the same machine or belonging to the same job are such that :  $r_j + p_j + p_i + q_i > UB$ , then immediate selections on disjunctions add the precedence constraint  $i \rightarrow j$ , and adjust the head of task  $j$  which cannot start before :  $\max(r_j, r_i + p_i)$ .

If  $r_j$  is modified, it is because of the new constraint  $i \rightarrow j$ , and its new value depends on the value of  $r_i$ . Then the explanation of the new value of  $r_j$  becomes :  $Expl(r_j) := Expl(r_j) \cup Expl(i, j) \cup Expl(r_i)$

If  $r_j$  is not modified, its explanation does not change.

2. **Immediate selections on sets**

Consider a machine  $M_k$  on which *immediate selections on sets* are calculated.

Let  $c$  be a task executed on  $M_k$ , and  $J$  be the set of all tasks executed on  $M_k$ , except  $c$ . If the head of a task  $c$  is modified, its new value is :  $r_c \leftarrow \max(r_c, C^*)$  where

$$C^* = \max_{J' \subset J} \left\{ \min_{j \in J'} r_j + \sum_{j \in J'} p_j \right\}$$

This modification is then due to the values  $r_i$  and  $q_i$  of all tasks belonging to  $J'$ . But we saw that the algorithm used does not compute explicitly this set  $J'$ . For this reason, we will suppose that the modification of  $r_c$  is due to all tasks executed on  $M_k$  (this is obviously a valid explanation).

If  $r_c$  is changed, the explanation of its new value is then:

$$Expl(r_c) \leftarrow \{Expl(r_i) \mid i \text{ executed on } M_k\} \cup \{Expl(q_i) \mid i \text{ executed on } M_k\}$$

3. **Recomputation of heads and tails**

To update a head  $r_i$ , the following relation is used:

$$r_i \leftarrow \max \left\{ \begin{array}{l} r_i, \\ \min_{j \in \Gamma_M^{-1}(i)} r_j + \sum_{j \in \Gamma_M^{-1}(i)} p_j, \max_{j \in \Gamma_M^{-1}(i)} (r_j + p_j), \\ \min_{j \in \Gamma_J^{-1}(i)} r_j + \sum_{j \in \Gamma_J^{-1}(i)} p_j, \max_{j \in \Gamma_J^{-1}(i)} (r_j + p_j) \end{array} \right\}$$

Five cases are to be considered:

- $r_i$  is not modified : its explanation does not change.
- $r_i \leftarrow \min_{j \in \Gamma_M^{-1}(i)} r_j + \sum_{j \in \Gamma_M^{-1}(i)} p_j$ : the modification of  $r_i$  is due to all the values  $r_j$ ,  $j \in \Gamma_M^{-1}(i)$ , and to all arcs  $(j, i)$ ,  $j \in \Gamma_M^{-1}(i)$ . Then we have:

$$Expl(r_i) \leftarrow Expl(r_i) \cup \{Expl(r_j), j \in \Gamma_M^{-1}(i)\} \cup \{Expl(j, i), j \in \Gamma_M^{-1}(i)\}$$

- $r_i \leftarrow \max_{j \in \Gamma_M^{-1}(i)} (r_j + p_j) = r_k + p_k$ :

The explanation of the new value of  $r_i$  becomes:

$$Expl(r_i) \leftarrow Expl(r_i) \cup Expl(r_k) \cup Expl(k, i)$$

- $r_i \leftarrow \min_{j \in \Gamma_J^{-1}(i)} r_j + \sum_{j \in \Gamma_J^{-1}(i)} p_j$ : The modification of  $r_i$  is due to all the values  $r_j$ ,  $j \in \Gamma_J^{-1}(i)$ , and to all arcs  $(j, i)$ ,  $j \in \Gamma_J^{-1}(i)$ . Then we have:

$$Expl(r_i) \leftarrow Expl(r_i) \cup \{Expl(r_j), j \in \Gamma_J^{-1}(i)\} \cup \{Expl(j, i), j \in \Gamma_J^{-1}(i)\}$$

- $r_i \leftarrow \max_{j \in \Gamma_J^{-1}(i)} (r_j + p_j) = r_k + p_k$ :

The explanation of the new value of  $r_i$  becomes:

$$Expl(r_i) \leftarrow Expl(r_i) \cup Expl(r_k) \cup Expl(k, i)$$

## • Propositions

### Proposition 3

In each node, for each task  $i$ ,  $Expl(r_i)$  (resp.  $Expl(q_i)$ ) contains a reference to all nodes responsible for a modification of  $r_i$  (resp.  $q_i$ ).

### Proposition 4

In each node, for each fixed disjunction  $(i, j)$ ,  $Expl(i, j)$  contains a reference to all nodes responsible for the creation of this arc  $(i, j)$ .

**Proof :** These propositions can be proved by induction: they are true at root node. By construction, it is easy to check that if they are true at level  $r$  of the search tree, they are still true a level  $r + 1$ .  $\square$

## 4.2 Using the explanations

During the search, there are two possibilities of the elimination of a node:

- There exists a task  $i$  for which  $r_i + p_i + q_i \geq UB$  (the domain of the variable  $t_i$  becomes empty). In this case, the contradiction is due to the last modification of  $r_i$  or  $q_i$ . Thus, we backtrack to the most recent node responsible for the modification of either  $r_i$  or  $q_i$ . This node is the most recent node in  $Expl(r_i) \cup Expl(q_i)$ .
- All children of a node  $r$  have been eliminated. Then node  $r$  is an unsuccessful node. The node of backtrack is the most recent node in the union of the explanations of children's eliminations, minus  $r$ .

## 4.3 Complexity

The spatial complexity of our approach is related to the number of disjunctions of the problem. Since each task  $c$  is in disjunction with all the tasks belonging to the same job and with the tasks executed on the same machine, this number is in  $O(m \times n(m + n))$ . The number of disjunctions gives the maximum depth of the search tree in the worst case, since at least one disjunction is fixed in each node. Thus this number is the maximum size of each explanation. The space needed to record all explanations (for heads, tails and disjunctions) is then  $O(m^2 \times n^2(m + n)^2)$  in the worst case.

The time complexity in each node depends on the number of union of explanations made during the computation of immediate selections. As immediate selections are executed until no improvement is found, this number cannot be determined. But for example, if immediate selections on sets are computed on a machine, the number of unions of explanations is in  $O(n)$ . As each union takes  $O(m \times n(m + n))$ , the overcost of our approach in the computation of immediate selections on sets is  $O(m \times n^2(m + n))$ .

## 5 Computational results

We tested our new approach on benchmarks by Taillard [1993]. These benchmarks are composed of 10 square instances of size 4, 5, 7 and 10. Among the problems of size 10, 4 are still unsolved.

## 5.1 Implementation

We show in [Guéret and Prins, 1996] that min-max matchings (instead of max-min, or max weight, ... matchings) gives better results at root node. So this version has been used in each node in our implementation of the branch-and-bound of Brucker *et al.*

In the same paper a new heuristic is presented, denoted *H1*. This heuristic is a list algorithm with two priorities for each task  $O_{ij}$ : the residual work duration of job  $J_j$  and of machine  $M_i$ . At each iteration, the priority list  $L$  is constructed and the tasks are scheduled according to this list without updating it after each task placement. Then the schedule obtained is scanned in order to find the set  $S$  of tasks which are in progress while one machine at least is idle or which end after a lower bound  $LB$ . Such tasks are moved by one position towards the beginning of  $L$ . At the next step, a new schedule is constructed according this new priority list, and so on. The best schedule obtained after 50 iterations is kept. As shown in [Guéret and Prins, 1996], this heuristic is more efficient than the matching heuristic used in Brucker *et al.* algorithm. So this heuristic is used to find an initial solution in our implementation.

## 5.2 Results

Table 1 presents the results obtained. For each problem, a lower bound  $LB$ , an upper bound  $UB$  (initial solution) and the optimal solution  $OPT$  (unknown for four problems of size 10) are given. This table also contains the number of backtracks. We stopped the search to 250000 backtracks (about three hours of CPU time on a Pentium PC clocked at 133 MHz). If the optimal solution is not achieved at this limit, the best solution found is indicated in brackets. The optimal solution of the open problem we solved is outlined.

Although it is difficult to generalize the results obtained on only 40 problems, it seems that the bigger the problem, the more efficient the technique is : indeed, this technique has no effect on the 4x4 problems of Taillard, the number of backtracks is reduced for 9 problems of size 5, and for all problems of size 7x7.

For some problems, the improvement is very important, for example for the tenth problem of size 5 for which the reduction is greater than 90%, but also for 10x10 problems : three problems are solved by our technique, among which an open problem (the one with outlined optimal value) in only 4843 backtracks, whereas without intelligent backtracking, no problem of this size is solved in less than 250000

Size	<i>OPT</i>	<i>LB</i>	<i>UB</i>	without intelligent backtracking number of backtracks	with intelligent backtracking number of backtracks
4	193	186	197	18	18
4	236	229	253	37	37
4	271	262	272	29	29
4	250	245	260	26	26
4	295	287	305	55	55
4	189	185	193	19	19
4	201	197	203	22	22
4	217	212	217	14	14
4	261	258	268	32	32
4	217	213	224	31	31
5	300	295	303	273	<b>270</b>
5	262	255	266	252	<b>238</b>
5	323	321	347	943	<b>940</b>
5	310	306	319	678	678
5	326	321	330	840	<b>836</b>
5	312	307	325	756	<b>737</b>
5	303	298	308	420	<b>411</b>
5	300	292	304	853	<b>851</b>
5	353	349	368	1000	<b>987</b>
5	326	321	337	2377	<b>2377</b>
7	435	435	452	2840	<b>1860</b>
7	443	443	465	18196	<b>16862</b>
7	468	468	495	98903	<b>96502</b>
7	463	463	482	1494	<b>1410</b>
7	416	416	425	317	<b>256</b>
7	451	451	471	21492	<b>20364</b>
7	422	422	449	56944	<b>53773</b>
7	424	424	433	1939	<b>1552</b>
7	458	458	476	560	<b>535</b>
7	398	398	411	731	<b>710</b>
10		637	654	> 250000 (644)	> 250000 (644)
10	588	588	600	> 250000 (594)	> 250000 ( <b>591</b> )
10		598	611	> 250000 (610)	> 250000 ( <b>604</b> )
10	577	577	588	> 250000 (584)	<b>26777</b>
10	640	640	661	> 250000 (658)	> 250000 (658)
10	538	538	544	> 250000 (544)	> 250000 (544)
10	616	616	633	> 250000 (633)	<b>4843</b>
10	595	595	604	> 250000 (604)	> 250000 (604)
10	595	595	612	> 250000 (597)	<b>245 100</b>
10		596	612	> 250000 (611)	> 250000 ( <b>606</b> )

Table 1: Results on Taillard's problems

backtracks. And for the other problems of size 10, our technique generally provides better solutions at the limit backtrack numbers.

As regards execution times, the version with intelligent backtracking is about two times slower than the initial version in each node. But this waste of time is widely compensated by the decrease of the number of explored nodes. Thus for the open problem we solved, the execution time is reduced by at least (since the problem is not solve after 250000 backtracks in the initial version) 90% .

## 6 Conclusion and further work

In this article, we presented an intelligent backtracking technique that we applied on a branch-and-bound for the Open-Shop problem. This technique provides interesting results : a problem with 10 jobs and 10 machines has been solved for the first time.

Our technique can easily be adapted to all branch-and-bound for other shops problems like Job-Shops and Flow-Shops in which branching scheme consists in fixing disjunctions.

This technique is an adaptation to a Branch-and-Bound for Open-Shop problems of a part of our previous works [Jussien, 1997; Jussien and Boizumault, 1997b] on DECORUM system in which intelligent backtracking is completely replaced with a *dynamic backtracking* [Ginsberg, 1993] which, instead of backtracking, consists in *jumping* from solution to other solutions. Currently we are working on the implementation of that whole system to solve Open-Shop problems.

## References

- [Bellman, 1958] R. E. Bellman. On a routing problem. *Quat. Appl. Math.*, 16:87–90, 1958.
- [Brasel *et al.*, 1991] H. Brasel, D. Kluge, and F. Werner. A polynomial time algorithm for an open-shop problem with unit processing times and tree constraints. Technical report, Technische Univeristaet Otto van Guericke, Magdeburg, 1991.
- [Brucker *et al.*, 1994a] P. Brucker, B. Jurish, and B. Sievers. A branch and bound algorithm for the open-shop problem. *Discrete Applied Mathematics*, 1994. to appear.

- [Brucker *et al.*, 1994b] P. Brucker, B. Jurish, and B. Sievers. A fast branch and bound algorithm for the job-shop scheduling problem. *Discrete Applied Mathematics*, 49:107–127, 1994.
- [Brucker *et al.*, 1996] P. Brucker, T. Hilbig, and J. Hurink. A branch and bound algorithm for scheduling problems with positive and negative time-lags. Technical report, Osnabrueck University, may 1996.
- [Bruynooghe and Pereira, 1984] M. Bruynooghe and L. M. Pereira. *Implementations of Prolog*, chapter Deduction revision by intelligent backtracking, pages 194–215. Ellis Horwood, 1984.
- [Carlier and Pinson, 1989] J. Carlier and É. Pinson. An algorithm for solving the job-shop problem. *Management Science*, 35:164–176, 1989.
- [Carlier and Pinson, 1994] J. Carlier and É. Pinson. Adjusting heads and tails for the job-shop problem. *European Journal of Operations Research*, 78:146–161, 1994.
- [Cox, 1984] P. T. Cox. *Implementation of Prolog*, chapter Finding backtrack points for intelligent backtracking, pages 216–233. Ellis Horwood, 1984.
- [de Backer and Béringer, 1991] Bruno de Backer and Henri Béringer. Intelligent backtracking for CLP languages: An application to CLP( $\mathcal{R}$ ). In Vijay Saraswat and Kazunori Ueda, editors, *ILPS'91: Proceedings International Logic Programming Symposium*, pages 405–419, San Diego, CA, October 1991. MIT Press.
- [Dechter, 1990] Rina Dechter. Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition. *Artificial Intelligence*, 41(3):273–312, January 1990.
- [Ginsberg, 1993] Matthew L. Ginsberg. Dynamic backtracking. *Journal of Artificial Intelligence Research*, 1:25–46, 1993.
- [Gonzalez and Sahni, 1976] T. Gonzalez and S. Sahni. Open-shop scheduling to minimize finish time. *Journal of the Association for Computing Machinery*, 23(4):665–679, 1976.
- [Grabowski *et al.*, 1986] J. Grabowski, E. Nowicki, and S. Zdrzalka. A block approach for single-machine scheduling with release dates and due dates. *European Journal of Operations Research*, 26:278–285, 1986.

- [Guéret and Prins, 1996] C. Guéret and C. Prins. Classical and new heuristics for the open-shop problem. Research Report 96-3-AUTO, École des Mines de Nantes, 1996. to appear in *European Journal of Operations Research*.
- [Jussien and Boizumault, 1997a] Narendra Jussien and Patrice Boizumault. Best-first search for property maintenance in reactive constraints systems. In *International Logic Programming Symposium*, Port Jefferson, N.Y., oct 1997. MIT Press.
- [Jussien and Boizumault, 1997b] Narendra Jussien and Patrice Boizumault. Dynamic backtracking with constraint propagation – application to static and dynamic csps. In *CP97 Workshop on The Theory and Practice of Dynamic Constraint Satisfaction*, Schloss Hagenberg, Austria, 1 November 1997.
- [Jussien, 1997] Narendra Jussien. *Relaxation de Contraintes pour les problèmes dynamiques*. PhD thesis, Université de Rennes I, 24 October 1997.
- [Taillard, 1993] É. Taillard. Benchmarks for basic scheduling problems. *European Journal of Operations Research*, 64:278–285, 1993.