



ELSEVIER

European Journal of Operational Research 127 (2000) 344–354

EUROPEAN
JOURNAL
OF OPERATIONAL
RESEARCH

www.elsevier.com/locate/dsw

Using intelligent backtracking to improve branch-and-bound methods: An application to Open-Shop problems

Christelle Guéret^{*}, Narendra Jussien, Christian Prins¹

École des Mines de Nantes, La Chantrerie, 4 Rue Alfred Kastler, BP20722, F-44307 Nantes Cedex 03, France

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 remain still unsolved with it.

We present an improving technique for branch-and-bound methods applied to Brucker et al.'s algorithm for Open-Shop problems. Our technique is based on intelligent backtracking. An adaptation of a generic explanation system we have initially developed in the constraint programming scheme is used to develop that technique.

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 backtracks is reduced by more than 90% and we even solved an open problem of that size. © 2000 Elsevier Science B.V. All rights reserved.

Keywords: Scheduling theory; Open-shop; Branch-and-bound; Intelligent backtracking

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 $m \geq 3$ [13].

Only two branch-and-bound methods for that problem have been published so far. The first one [3] is based on the resolution of a one-machine problem with positive and negative time-lags. The second one [4] 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 remain unsolved.

In this paper, we introduce an improving technique for branch-and-bound methods for shop problems in which the branching scheme consists in adding precedence constraints. We present its

^{*} Corresponding author.

E-mail addresses: christelle.gueret@emn.fr (C. Guéret), narendra.jussien@emn.fr (N. Jussien), christian.prins@univ-troyes.fr (C. Prins).

¹ Present address: Université de Technologie de Troyes – BP 2060 – 10010 Troyes Cedex, France.

application to the algorithm of Brucker et al. 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 towards a more relevant choice point, 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 [19].

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

The branch-and-bound method 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. [14] for one-machine problems with release dates and due dates;
- *immediate selections* [7,8], a method initially designed to fix disjunctions in Job-Shop problems.

We briefly present this branch-and-bound algorithm. The details can be found in [4,5]. We first recall some vocabulary and basic concepts 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 that 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 C_{\max} 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 .
- 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 $n \times m + 1$.
- f_i is the latest start time of i . $f_i = C_{\max} - q_i - p_i$.

Each time a disjunction is fixed, heads and tails must be recomputed. A simple way to achieve that is to use, for example, Bellman's algorithm [1].

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 :

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

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 generalizes an approach used by Grabowski et al. [14] 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 that solution.

The heuristic used has been developed by Bräsel et al. [2]. It 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 that 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, etc. This heuristic is designed for problems without precedence constraints. It can easily be adapted for problems in which some disjunctions have been fixed [4].

Once the heuristic solution is obtained, a critical path μ is computed. Then, the *blocks* of this critical path are determined.

Definition 2 (Block [4]). A sequence u_1, u_2, \dots, u_l of successive nodes in μ is called a block on μ 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 (Ref. [4]). Let S be a heuristic solution and μ a critical path of S . If there exists a better solution S' , then in S' at least one operation of one block of μ has to be processed before the first or after the last operation of that 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 [7] 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 $I, c \notin I$, of tasks 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 (Ref. [7]). *If*

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

and

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

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 those two lower bounds are greater than UB, then c must be executed after all the tasks of J in all solutions better than UB.

In this case, the head of c can be adjusted by $r_c \leftarrow \max(r_c, C^*)$, where

$$C^* = \max_{J' \subset J} \left\{ \min_{j \in J'} r_j + \sum_{j \in J'} p_j \right\}. \quad (2)$$

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

Proposition 2. *If $r_c + p_c + p_i + q_i \geq \text{UB}$ then c has to be processed after i .*

The disjunctive arc $i \rightarrow c$ can then be fixed, and operation c cannot start before $r_c \leftarrow \max(r_c, r_i + p_i)$.

Those *immediate selections* are called *immediate selections on disjunctions*. Those of Proposition 1 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 that reason, it is not possible to fix new disjunctions between c and the tasks of J ; this algorithm only adjusts the head of c . But those disjunctions will be automatically 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 defined to adjust tails.

3. Improving the search

Depth-first search branch-and-bound methods 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 three machines and two jobs, i.e., we want to determine which job is scheduled before the other on each machine. The binary search tree (cf. Fig. 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 Fig. 1. This node exceeds the upper bound, it is therefore not an acceptable solution. Since the upper bound was respected in its father node, one can infer that choice $J_1 \rightarrow J_2$ was inadequate on machine 3. The next explored node is thus the one labeled 100.

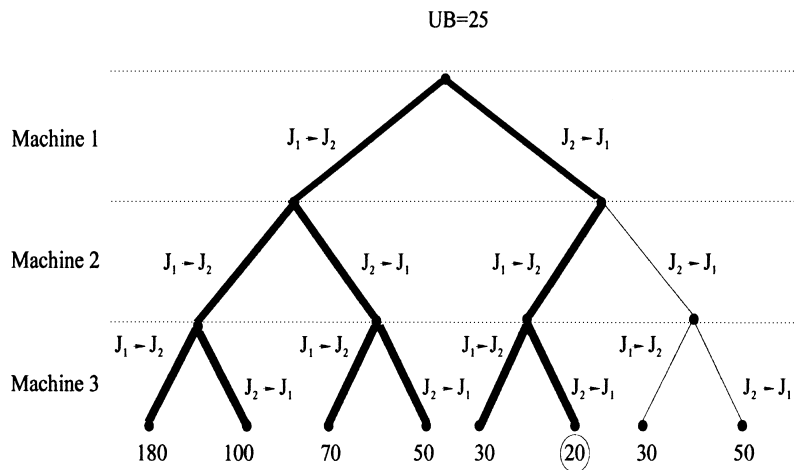


Fig. 1. Binary search tree for a 3×2 Job-Shop problem.

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 $J_1 \rightarrow J_2$ or together with $J_2 \rightarrow J_1$ on machine 3 exceeds the upper bound. We can therefore consider that an *explanation* or justification of removing choice $J_1 \rightarrow J_2$ on machine 3 is only the constraint: $J_1 \rightarrow J_2$ on machine 1. Moreover, an explanation for removing choice $J_2 \rightarrow J_1$ on machine 3 is only: $J_1 \rightarrow J_2$ on machine 1. From those two explanations it is therefore easy to deduce that the only reason why no possible disjunction can be fixed on machine 3 is because of the choice of $J_1 \rightarrow J_2$ on machine 1. Thus, 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 three nodes.

That situation is due to the fact that, in spite of their sophistication, branching schemes may take completely independent 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. We will more formally introduce *explanations* and their use in the search.

3.2. Intelligent backtracking techniques

The idea of *intelligent backtracking* has been defined in the Logic Programming community ([6] or [9]) 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* [11] or for *Constraint Logic Programming* (Logic Programming with constraint propagation) [10]. In the latter, an intelligent backtracking mechanism is proposed to be used with an incremental simplex algorithm 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 works [17], we presented a generic explanation system to be used with constraint programming.

3.3. An explanation system for constraint programming

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 of the tree such that the conjunction 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 whatever be the surrounding context.

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 in that node is contradictory. The path to the current node in the search tree is an explanation² 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_c be a contradiction explanation. Let r be the most recent node involved in C_c . Backtracking to any node between r and the current node is useless since all the decisions made in the nodes of C_c will remain active, thus leading to a contradiction. Backtracking can directly be performed towards r saving more or less work. The set $C_c \setminus \{r\}$ 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 at least 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 analyzing the explanations of all its unsuccessful children.

4. Application

We explain now how to adapt our approach to the branch-and-bound method 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 an r_i (resp. f_i) can only increase (resp. decrease) during the search, each time an r_i or an 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 that modification.

The explanations for r_i and f_i are the same than the one of r_i and q_i , since $q_i = \text{UB} - f_i - p_i$. In order to be consistent with *immediate selections*, modifications of r_i and q_i (and not on f_i) will be considered.

A modification of a head or a tail may be derived from the addition of a new precedence constraint. Such a new precedence constraint may be created because of the decisions taken in some nodes. So, in order to record explanations for each modification of heads and tails, an 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 for heads. Updating explanations for tails is not explained since the method is symmetrical to the one for the heads.

An explanation is a set of nodes such that the conjunction of the decisions made in them leads to the associate conclusion (contradiction or value removal from the domain of a variable). 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

² As an example, one can consider the search performed in Section 3.1.

(r). So the explanation of the arc (i, j) is

$$\text{Expl}(i, j) \leftarrow \{r\}. \tag{3}$$

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 those two tasks belong to the same job, the symmetrical results can easily be deduced).

If $r_j + p_j + p_i + q_i \geq \text{UB}$ 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 . Thus, its explanation becomes

$$\text{Expl}(i, j) \leftarrow \text{Expl}(r_j) \cup \text{Expl}(q_i). \tag{4}$$

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

• *Recording explanations for heads and tails*

3. *Immediate selections on disjunctions*

If two tasks i and j executed on the same machine or belonging to the same job are such that $r_j + p_j + p_i + q_i \geq \text{UB}$, 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 $\text{Expl}(r_j) := \text{Expl}(r_j) \cup \text{Expl}(i, j) \cup \text{Expl}(r_i)$. If r_j is not modified, its explanation does not change.

4. *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

$$C^* = \max_{J' \subset J} \left\{ \min_{j \in J'} r_j + \sum_{j \in J'} p_j \right\}. \tag{5}$$

That modification is due to the values r_i and q_i of all tasks belonging to J' . But we saw in Section 2.3 that the used algorithm does not explicitly compute this set J' . For this reason, we will suppose that the modification of r_c is due to all tasks executed on M_k (that is obviously a valid (rough) explanation).

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

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

5. *Recomputation of heads and tails*

As we saw in Section 1, in order to update a head r_i , the following relation is used:

$$r_i \leftarrow \max \left\{ \begin{array}{l} r_i, \\ \min_{j \in \Gamma_M^{-1}(i)} r_j + \sum_{j \in \Gamma_M^{-1}(i)} p_j, \\ \max_{j \in \Gamma_M^{-1}(i)} (r_j + p_j), \\ \min_{j \in \Gamma_J^{-1}(i)} r_j + \sum_{j \in \Gamma_J^{-1}(i)} p_j, \\ \max_{j \in \Gamma_J^{-1}(i)} (r_j + p_j). \end{array} \right\} \tag{6}$$

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

$$\text{Expl}(r_i) \leftarrow \text{Expl}(r_i) \cup \{ \text{Expl}(r_j) \cup \text{Expl}(j, i), j \in \Gamma_M^{-1}(i) \}. \tag{7}$$

○ $r_i \leftarrow \max_{j \in \Gamma_M^{-1}(i)} (r_j + p_j) = r_k + p_k$:
The explanation of the new value of r_i becomes

$$\text{Expl}(r_i) \leftarrow \text{Expl}(r_i) \cup \text{Expl}(r_k) \cup \text{Expl}(k, i). \tag{8}$$

○ $r_i \leftarrow \min_{j \in \Gamma_J^{-1}(i)} r_j + \sum_{j \in \Gamma_J^{-1}(i)} p_j$:
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

$$\text{Expl}(r_i) \leftarrow \text{Expl}(r_i) \cup \{\text{Expl}(r_j) \cup \text{Expl}(j, i), \\ j \in \Gamma_J^{-1}(i)\}. \quad (9)$$

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

The explanation of the new value of r_i becomes

$$\text{Expl}(r_i) \leftarrow \text{Expl}(r_i) \cup \text{Expl}(r_k) \cup \text{Expl}(k, i). \quad (10)$$

Proposition 3. *In each node, for each task i , $\text{Expl}(r_i)$ (resp. $\text{Expl}(q_i)$) 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) , $\text{Expl}(i, j)$ contains a reference to all nodes responsible for the creation of this arc (i, j) .*

Proof. Those two 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 leading to the elimination of a given node r :

- There exists a task i for which $r_i + p_i + q_i \geq \text{UB}$ (i.e., 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 . That node is the most recent node in $\text{Expl}(r_i) \cup \text{Expl}(q_i)$.
- All children of node r have been eliminated. Then, node r is an unsuccessful node. The backtrack node is then the most recent node in the union of the explanations of its children's eliminations, minus r .

4.3. Complexity

The spatial complexity of our approach is strongly related to the depth of the search tree since explanations are composed of a subset of nodes in

the path leading from root node to the current node. Let d_s denote the depth of the search tree. The maximum size of any explanation is then in $O(d_s)$.

Explanations are recorded for heads, tails and disjunctions. In a given Open-Shop problem of size $n \times m$, there are $n \times m$ heads and tails. 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, the number of disjunctions in a given problem is then: $m \times n(m + n - 2)$. Therefore, the number of explanations is in $O(m \times n(m + n))$ and the spatial complexity of our explanation system is, in the worst case, in $O(d_s \times m \times n(m + n))$.

The time complexity in each node depends on the number of unions 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 $O(d_s)$ (size of an explanation), the overhead of our approach in the computation of immediate selections on sets is $O(n \times d_s)$. Similar results can be given for the other head and tail adjusting techniques presented in this paper.

Note that the very unlikely worst case depth of the search is given when only one disjunction is fixed in each node leading to an $O(n \times m(m + n))$ depth. The related worst case spatial and time complexities thus become, respectively: $O(m^2 \times n^2 \times (m + n)^2)$ and $O(m \times n^2 \times (m + n))$.

5. Computational results

We tested our new approach on benchmarks by Taillard [19]. Those benchmarks are composed of 10 square instances of sizes 4, 5, 7 and 10. Among the problems of size 10, four are still unsolved.

5.1. Implementation

We showed in [15] that min–max matchings (instead of max–min, or max–weight, etc. matchings) give better results at root node. So that

Table 1
Results on Taillard's problems

Size	OPT	LB	UB	Number of backtracks	
				Without intelligent backtracking	With intelligent backtracking
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	270
5	262	255	266	252	238
5	323	321	347	943	940
5	310	306	319	678	678
5	326	321	330	840	836
5	312	307	325	756	737
5	303	298	308	420	411
5	300	292	304	853	851
5	353	349	368	1000	987
5	326	321	337	2377	2377
7	435	435	452	2840	1860
7	443	443	465	18 196	16 862
7	468	468	495	98 903	96 502
7	463	463	482	1494	1410
7	416	416	425	317	256
7	451	451	471	21 492	20 364
7	422	422	449	56 944	53 773
7	424	424	433	1939	1552
7	458	458	476	560	535
7	398	398	411	731	710
10		637	654	>250 000 (644)	>250 000 (644)
10	588	588	600	>250 000 (594)	>250 000 (591)
10		598	611	>250 000 (610)	>250 000 (604)
10	577	577	588	>250 000 (584)	26 777
10	640	640	661	>250 000 (658)	>250 000 (658)
10	538	538	544	>250 000 (544)	>250 000 (544)
10	<u>616</u>	616	633	>250 000 (633)	4843
10	595	595	604	>250 000 (604)	>250 000 (604)
10	595	595	612	>250 000 (597)	245 100
10		596	612	>250 000 (611)	>250 000 (606)

version has been used in each node in our implementation of the branch-and-bound method of Brucker et al.

In the same paper a new heuristic is presented, named H1. That 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 following that list without being updated after each task placement. Then, the schedule obtained is scanned in order to find the set S of tasks which are in progress while at least

one machine 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 that new priority list, and so on. The best schedule ever obtained after 50 iterations is kept. As shown in [15], that heuristic is more efficient than the matching heuristic used in Brucker et al. algorithm. So we used it 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 250 000 backtracks (about 3 hours of CPU time on a Pentium PC clocked at 133 MHz). If the optimal solution is not reached at that limit, the best solution found is indicated in brackets. The optimal solution of the open problem we solved is squared.

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 4×4 problems of Taillard, the number of backtracks is reduced for nine problems of size 5, and for all problems of size 7×7 .

For 10×10 problems, the improvement is very important: 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 that size is solved in less than 250 000 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 decreased number of explored nodes. Thus, for the open problem we solved, the execution time is reduced

by at least 90% (since the problem is not solved after 250 000 backtracks in the initial version).

6. Conclusion and further work

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

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

This technique is an adaptation to a branch-and-bound method for Open-Shop problems of a part of our previous works [16,18] on the DECORUM system in which intelligent backtracking is completely replaced with a *dynamic backtracking* [12] 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

- [1] R.E. Bellman, On a routing problem, *Quarterly of Applied Mathematics* 16 (1958) 87–90.
- [2] H. Bräsel, D. Kluge, F. Werner, A polynomial time algorithm for an open-shop problem with unit processing times and tree constraints, Technical Report, Technische Univeristat Otto van Guericke, Magdeburg, 1991.
- [3] P. Brucker, T. Hilbig, J. Hurink, A branch and bound algorithm for scheduling problems with positive and negative time-lags, Technical Report, Osnabrueck University, May 1996.
- [4] P. Brucker, J. Hurink, B. Jurish, B. Wöstman, A branch and bound algorithm for the open-shop problem, *Discrete Applied Mathematics* 76 (1997) 43–59.
- [5] P. Brucker, B. Jurish, B. Sievers, A fast branch and bound algorithm for the job-shop scheduling problem, *Discrete Applied Mathematics* 49 (1994) 107–127.
- [6] M. Bruynooghe, L.M. Pereira, Deduction revision by intelligent backtracking, in: *Implementations of Prolog*, Ellis Horwood, Chichester, 1984, pp. 194–215.
- [7] J. Carlier, É. Pinson, An algorithm for solving the job-shop problem, *Management Science* 35 (1989) 164–176.

- [8] J. Carlier, É. Pinson, Adjusting heads and tails for the job-shop problem, *European Journal of Operational Research* 78 (1994) 146–161.
- [9] P.T. Cox, Finding backtrack points for intelligent backtracking, in: *Implementation of Prolog*, Ellis Horwood, Chichester, 1984, pp. 216–233.
- [10] B. de Backer, H. Béringer, Intelligent backtracking for CLP languages: An application to CLP(\mathcal{R}), in: V. Saraswat, K. Ueda, (Eds.), *ILPS'91: Proceedings of the International Logic Programming Symposium*, San Diego, CA, October 1991, MIT Press, Cambridge, MA, pp. 405–419.
- [11] R. Dechter, Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition, *Artificial Intelligence* 41 (3) (1990) 273–312.
- [12] M.L. Ginsberg, Dynamic backtracking, *Journal of Artificial Intelligence Research* 1 (1993) 25–46.
- [13] T. Gonzalez, S. Sahni, Open-shop scheduling to minimize finish time, *Journal of the Association for Computing Machinery* 23 (4) (1976) 665–679.
- [14] J. Grabowski, E. Nowicki, S. Zdrzalka, A block approach for single-machine scheduling with release dates and due dates, *European Journal of Operational Research* 26 (1986) 278–285.
- [15] C. Guéret, C. Prins, Classical and new heuristics for the open-shop problem, *European Journal of Operational Research* 107 (1998) 306–314.
- [16] N. Jussien, *Relaxation de contraintes pour les problèmes dynamiques*, Ph.D. Thesis, Université de Rennes I, 24 October 1997.
- [17] N. Jussien, P. Boizumault, Best-first search for property maintenance in reactive constraints systems, in: *Proceedings of the International Logic Programming Symposium*, Port Jefferson, NY, October 1997, MIT Press, Cambridge, MA.
- [18] N. Jussien, P. 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.
- [19] É. Taillard, Benchmarks for basic scheduling problems, *European Journal of Operations Research* 64 (1993) 278–285.