# A Position-Based Propagator for the Open-Shop Problem

Jean-Noël Monette, Yves Deville, and Pierre Dupont

Department of Computing Sciences and Engineering
Université catholique de Louvain
{jmonette,yde,pdupont}@info.ucl.ac.be

**Abstract.** The Open-Shop Problem is a hard problem that can be solved using Constraint Programming or Operation Research methods. Existing techniques are efficient at reducing the search tree but they usually do not consider the absolute ordering of the tasks. In this work, we develop a new propagator for the One-Machine Non-Preemptive Problem, the basic constraint for the Open-Shop Problem. This propagator takes this additional information into account allowing, in most cases, a reduction of the search tree. The underlying principle is to use shaving on the positions. Our propagator applies on one machine or one job and its time complexity is in $\mathcal{O}(N^2 \log N)$, where $N$ is either the number of jobs or machines. Experiments on the Open-Shop Problem show that the propagator adds pruning to state-of-the-art constraint satisfaction techniques to solve this problem.

## 1 Introduction

Open-Shop Problems (OSP) are disjunctive scheduling problems known to be really hard to solve. Up to now, some problems with less than 50 tasks remain unsolved, although several powerful techniques and algorithms mentioned below have been designed to reduce efficiently the search.

The Open-Shop Problem aims at finding the order in which a set of tasks is executed such as to minimize the makespan, *i.e.* the ending time of the latest tasks. Each task must be executed on a particular machine for a given duration and without interruption. A machine cannot process two tasks at the same time. Additionally, tasks are part of jobs and two tasks of the same job cannot be processed at the same time. There is no predefined ordering between tasks.

This problem fits very well in the framework of Constraints Programming (CP). Propagators have been developed to remove inconsistent values from the domains as early as possible in order to reduce the size of the search tree. The more prominent techniques are Edge-Finding (EF) and Not-First-Not-Last (NFNL). Edge-Finding [1–4] consists in testing whether a particular task must start before or after a set of tasks. It can be implemented with a time complexity of $\mathcal{O}(N \log N)$ where $N$ is the number of tasks on one machine or one job. Not-First-Not-Last [5–7] checks if a task can be the first or the last among a set of tasks. Its best time complexity is $\mathcal{O}(N \log N)$.

Shaving [5, 6, 8] is an orthogonal technique that performs well in practice for solving the OSP. It consists in iteratively assigning to a variable its possible values and checking if this assignment leads to inconsistency. In that case, the value is removed from the domain of the variable. Every constraint can be propagated to check consistency until the fixpoint is observed. Since it can be costly to reach this fixpoint, simpler propagators are used. For instance, in [6], only Edge-Finding is used to look for inconsistencies. Even so, shaving is costly because the size of the domain of the starting time variables can be huge. For this reason, shaving in OSP usually considers only the bounds of the domain.

We propose here a new propagator for the One-Machine Non-preemptive Problem that exploits information given by the position of the tasks. This idea has already been used successfully in [9], [10] and [11]. The first work uses the positions as permutation variables in a sorting constraint. An extension of Edge-Finding is also applied. Second, [10] proposes a possible way to decide if a task can start at some position looking at the number of other tasks that can come before and after this task. Finally, [11] extends this idea proposing tighter bounds with an algorithm running in $\mathcal{O}(N^3)$

In this paper, we present an alternative way to use the position of the tasks based on the idea of shaving. For each possible position of a task, lower and upper bounds on the possible starting time of the task are computed using the duration and the domain of the variables of the tasks in the same job or machine. The resulting propagator is applied on the tasks that are part of the same job or machine with a time complexity of $\mathcal{O}(N^2 \log N)$, where $N$ is the number of tasks that are part of the job or that must be processed on the machine. This propagator permits additional pruning that is not performed by NFNL and EF and permits to detect about 14 % extra inconsistent nodes of the search tree on a standard benchmark [12].

The next section explains the problem under interest and its mapping in CP. Section 3 presents the new propagator and Section 4 describes experimental results assessing the pruning efficiency of the technique. In the last section, conclusions are drawn as well as directions for future work.

## 2 The One Machine Non-preemptive Problem

The OSP is an optimization problem that can be solved using branch-and-bound techniques. The goal of the optimization is to minimize the makespan, *i.e.* the ending time of the latest task. Branch-and-bound consists in solving successive feasibility versions of our problem. The feasibility version consists in determining if there exists a solution with a makespan smaller than a fixed value. Each time a solution with makespan $m$ is found, another solution is searched with the constraint that its makespan is at most $m - 1$. When no more solution exists, the last solution found is an optimal one.

The feasibility version of the OSP can be stated as the conjunction of smaller problems called One Machine Non-preemptive Problem (1NP). This problem aims at scheduling a set of tasks on a machine such that there is only one not

interruptible task processed at a time. Each task is given a duration, an earliest and a latest starting times. To model the OSP, it is sufficient to define a 1NP for every machine and every job (and to link them with the makespan). Indeed, jobs and machines in the OSP have the same behavior: No two tasks associated with a same job or a same machine can be processed simultaneously. The 1NP is also the basis for other disjunctive problems such as the Job-Shop Problem for instance.

Formally, the 1NP is defined as follows: $T$ is the set of tasks that must be processed and $N$ is its cardinality. For each task $t \in T$, $d(t)$, initial $est(t)$ and initial $lst(t)$ are given and denote respectively the duration, earliest and latest starting times of the task $t$. The problem is to find for each task $t$, the value $S(t)$ of its starting time such that $est(t) \leq S(t) \leq lst(t)$ and without task overlap: $\forall t_1, t_2 \in T, S(t_1) + d(t_1) \leq S(t_2) \vee S(t_2) + d(t_2) \leq S(t_1)$.

### 2.1 Problem Modelling in CP

To model the 1NP, several variables are defined for each task $t \in T$. An integer variable $S(t)$ represents the starting time of the task $t$. Its domain ranges from the earliest starting time to the latest starting time of the task ($dom(S(t)) = [est(t), lst(t)]$). To model the relative order between the tasks, a set variable $B(t)$ represents the set of tasks that come before the task $t$. Its initial domain is $dom(B(t)) = [\emptyset, \{u | u \in T, u \neq t\}]$. Indeed, initially no task is known to come before $t$ and all the tasks might come before $t$. The symbol $\overline{B}(t)$ (resp. $\underline{B}(t)$) represents the upper (resp. lower) bound of the variable $B(t)$. Furthermore, an additional variable $P(t)$ represents explicitly the absolute order (or the position) of the tasks in the machine. The domain of this variable ranges from 0 to $N-1$ with $N$ being the number of tasks to be processed. The link between the relative and absolute orders of the tasks is that $P(t)$ represents the size of $B(t)$.

The starting time and the relative ordering between tasks are commonly used in the modelling of disjunctive scheduling. The use of an absolute order comes from [9] where the author solves the Job-Shop Problem fixing the permutations of task orders. In their proposed formulation, a variable is defined for the starting time of each task, a variable for the starting time of the task in each position and a variable for the position of each task. Those three sets of variables are linked together with a sorting constraint and various reduction rules are then defined. As an initial approach, we chose here for simplicity not to use the variables for the starting time of the task in each position.

### 2.2 Constraints

With three complementary representations, the 1NP can be equivalently expressed using anyone of the three following sets of constraints stating that two tasks cannot be processed at the same time.

1. $\forall t_1, t_2 \in T, \quad (S(t_1) + d(t_1) \leq S(t_2)) \quad \vee \quad (S(t_2) + d(t_2) \leq S(t_1))$
2. $\forall t_1, t_2 \in T, \quad (t_1 \in B(t_2)) \quad \vee \quad (t_2 \in B(t_1))$

3. $\forall t_1, t_2 \in T, \quad (P(t_1) < P(t_2)) \quad \vee \quad (P(t_2) < P(t_1))$

Our model uses the three sets of constraints to speed-up propagation. Additionally, the following channeling constraints ensure the consistency between variables of each representation. The position of a task $t$ is the number of tasks that come before $t$ ($|B(t)| = P(t)$). Also, a task $t_1$ ends before another task $t_2$ starts if and only if the position of $t_1$ is less than the position of $t_2$ ($S(t_1) + d(t_1) \leq S(t_2) \Leftrightarrow t_1 \in B(t_2) \Leftrightarrow P(t_1) < P(t_2)$).

In addition to these basic constraints, other redundant constraints can be defined. First, if $t_1$ comes before $t_2$, every task that comes before $t_1$ comes also before $t_2$ ($t_1 \in B(t_2) \Leftrightarrow B(t_1) \subset B(t_2)$). An AllDifferent constraint is also defined on the position variables ($\mathit{alldiff}(\{P(t) : t \in T\})$), because two tasks cannot have the same order of execution.

This last constraint is a first example of global constraint. Global constraints take into account more than two tasks at a time. NFNL and EF are also such global propagators that allow a much better pruning than the basic constraints. However, NFNL and EF do not use the information given by the position of the tasks to derive their information. This work shows how to use this additional information.

## 3 The Propagator

The main idea of the new propagator is to apply shaving on the position variables. Commonly, shaving is applied on the starting time variables and only on their bounds because of the size of their domains. On the contrary, the domain of the position variables is rather small and could be shaved in a reasonable time. To test if the task can be scheduled in a particular position, we compute bounds on its earliest and latest starting time under this assumption. If the resulting range does not intersect the domain of $S(t)$, the task cannot be scheduled in that position. Furthermore, shaving $P(t)$ permits also to reduce the domain of $S(t)$ to the union of the ranges computed for every position. Following this scheme, two issues need to be addressed. Firstly, the way to use the bounds on the task starting time to reduce the domains of the variables (Section 3.1). Secondly, the approximations used to compute ranges as tight as possible (Section 3.2). Notice that our approach of shaving is local to this propagator.

Let us first introduce some additional notations. As $est(t)$ represents the earliest starting time of a task $t$, $ect(t)$ will denote its earliest completion time. Those values are linked by $ect(t) = est(t) + d(t)$. The same quantities can be defined for set of tasks. If $U$ is a non-empty subset of $T$, $d(U)$ is the sum of the durations of the tasks in $U$ and $est(U)$ is the earliest starting time of the set of tasks $U$. It is equal to the earliest starting time of any tasks in $U$ ($est(U) = \min_{t \in U} est(t)$). The dual quantity $ect(U)$ is the earliest completion time of the set $U$, the time when every task in $U$ is finished. This last quantity cannot be computed easily but several lower bounds are known. Especially, the maximum, among every subset $U'$ of $U$, of the sum of the earliest starting time of $U'$ and the duration of $U'$ will be used in this work to approximate $ect(U)$ (Equation

1). This is only a bound because it does not take into account the latest starting time of the tasks. We denote it $b\_ect(U)$.

$$b\_ect(U) = \max_{\emptyset \neq U' \subseteq U} (est(U') + d(U')) \tag{1}$$

### 3.1 Shaving on position variables

Shaving enumerates every possible value of $P(t)$. Under the assumption that the position $P(t)$ of a task $t$ takes a particular value $p$ of its domain, the possible starting time of $t$ belongs to an interval $[est(t,p), lst(t,p)]$ where $est(t,p)$ and $lst(t,p)$ denote respectively the earliest and latest possible starting times when $t$ is in position $p$.

The value $est(t,p)$ is related with $ect(B(t),p)$ that is the earliest time when $p$ tasks among those in $\overline{B}(t)$ have been processed and when all the tasks in $\underline{B}(t)$ have been processed. Indeed, in position $p$, the task $t$ cannot start before that $p$ tasks among those that can come before $t$ have been processed. Furthermore, $t$ cannot start before the tasks that must come before are completed. This leads to the relation

$$est(t,p) = \max\left(ect(B(t),p), est(t)\right).$$

In this formula, $ect(B(t),p)$ cannot be computed exactly with a reasonable complexity. We propose however to compute a lower bound as tight as possible. Section 3.2 details how to approximate the value of $ect(B(t),p)$. A very similar reasoning, not detailed here, can be made to compute $lst(t,p)$.

Once the ranges $[est(t,p), lst(t,p)]$ have been computed for each $p \in P(t)$, the domain of $P(t)$ and $S(t)$ can be reduced with two simple rules:

$$\forall p \in dom(P(t)) : ([est(t,p), lst(t,p)] \cap dom(S(t)) = \emptyset) \Rightarrow P(t) \neq p \tag{2}$$

$$dom(S(t)) := dom(S(t)) \cap (\cup_{p \in dom(P(t))} [est(t,p), lst(t,p)]) \tag{3}$$

The first rule removes from the domain of $P(t)$ the values $p$ for which there is no valid starting time, i.e. when the range $[est(t,p), lst(t,p)]$ is empty or when it does not intersect with the domain of $S(t)$. The second rule restricts the domain of $S(t)$ to be included in the union of the computed ranges. Alternatively rule (3) could only reduce the bounds of the domain of $S(t)$, while ensuring that $S(t)$ remains a single interval. The latter is standard in scheduling. $S(t)$ must then be greater than the least value among the $est(t,p)$ for valid $p$'s and less than the greatest value among the $lst(t,p)$ for valid $p$'s.

$$dom(S(t)) := [\min_{p \in dom(P(t))} (est(t,p)), \max_{p \in dom(P(t))} (lst(t,p))] \tag{4}$$

Experiments will consider the two versions of the reduction of $S(t)$. The reduction of $S(t)$ (using rule (4)) and $P(t)$ can be done with a time complexity of $\mathcal{O}(N)$ where $N$ is the number of tasks to be processed on the machine, thus an upper bound on the size of the domain of $P(t)$.

### 3.2 Bounding the earliest completion time of a task subset

This section presents the evaluation of $ect(B(t), p)$ that is useful to evaluate $est(t, p)$. The algorithm to compute $lst(t, p)$ is similar but is not exposed. In order to compute a lower bound of $ect(B(t), p)$, we compute the minimum of the earliest completion time over all the sets $U$ of cardinality $p$ that are superset of $\underline{B}(t)$ and subset of $\overline{B}(t)$. In the following, $b\_ect(B(t), p)$ will denote the lower bound of $ect(B(t), p)$. This is a lower bound because it makes use of $b\_ect(U)$ which is a lower bound itself.

$$b\_ect(B(t), p) = \min_{U} (b\_ect(U)) \tag{5}$$

$$\text{where } |U| \geq p \text{ and } \underline{B}(t) \subseteq U \subseteq \overline{B}(t)$$

Interestingly, this lower bound can be computed efficiently using rules similar to the ones in the Jackson Preemptive Schedule [13] for computing the earliest ending time of a set of task supposing preemption. Our algorithm also allows preemption for the tasks but does not take into account the latest starting time of the tasks. Instead, the duration of the tasks is considered to schedule a subset of tasks of fixed size as soon as possible in a preemptive way. It is done respecting the following precedence rules:

- Whenever a task $t$ is available and the machine is free, process $t$.
- When a task $t_1$ becomes available during the processing of another task $t_2$ and the remaining processing time of $t_1$ is less than the remaining processing time of $t_2$, stop $t_2$ and start processing $t_1$.
- When a task $t_1$ becomes available during the processing of another task $t_2$, such that $t_1 \in \underline{B}(t)$ and $t_2 \notin \underline{B}(t)$, stop $t_2$ and start $t_1$.

The value $b\_ect(B(t), p)$ is obtained when every tasks in $\underline{B}(t)$ have been processed and at least $p$ tasks in $\overline{B}(t)$ have been processed.

An important property is that, although the algorithm supposes the tasks to be interruptible, the resulting quantities correspond exactly to the ones given by equation (5) where no preemption is supposed. Indeed, it is possible to merge the different parts of the completed tasks following the order of their starting times. The result is a non-preemptive schedule of the set of tasks. Preemption is not used here as a relaxation but just as a way to ease the computation. The computed value of $b\_ect(B(t), p)$ is however a relaxation of the exact value because the latest possible starting times of the tasks are not considered.

Moreover a single run of the above algorithm gives the value $b\_ect(B(t), p)$ for every $p$. Indeed, it suffices to remember the successive times when a task ends to have the $b\_ect(B(t), p)$ value for the successive values of $p$.

A pseudo-code of the algorithm is presented in Algorithm 1. The algorithm uses two priority queues. The first (Q1) sorts the tasks in order of earliest starting time. It permits to put in the second priority queue (Q2) only the available tasks at a particular time (lines 9-14). Q2 sorts the tasks in ascending order of remaining duration. When a task is popped from Q2, two situations arise.

**Algorithm 1**: Simplified Algorithm to Compute $ect(B(t), p)$

**Input**: B : the set of tasks
D : vector of the duration of the tasks
EST : vector of the est of the tasks
**Output**: ECT : vector of the $b\_ect(B(t), p)$ for each position p

```
1  Q1 := new PriorityQueue()
2  Q2 := new PriorityQueue()
3  time := 0
4  p := 0
5  forall t ∈ B do
6  |  RD(t) := D(t) //RD is the remaining duration
7  |  Q1.put(t,EST(t))
8  while not Q1.empty() do
9  |  t := Q1.pop()
10 |  time := EST(t)
11 |  Q2.put(t,RD(t))
12 |  while not Q1.empty() and EST(Q1.top()) = time do
13 |  |  t := Q1.pop()
14 |  |  Q2.put(t,RD(t))
15 |  while not Q2.empty() and
16 |  (Q1.empty() or time + RD(Q2.top()) < EST(Q1.top()) ) do
17 |  |  t := Q2.pop()
18 |  |  time := time +RD(t)
19 |  |  RD(t) := 0
20 |  |  p := p+1
21 |  |  ECT(p) := time
22 |  if not Q2.empty() then
23 |  |  t := Q2.pop()
24 |  |  RD(t) := RD(t) + time - EST(Q1.top())
25 |  |  Q2.push(t,RD(t))
26 |  |  time := EST(Q1.top())
27 |
28 return ECT
```

Either it can be processed before a new task is available and the time when it ends is recorded (lines 15-21). Or the task must be interrupted to check if a newly available task could not end earlier (lines 23-26).

For simplicity, the outlined algorithm is a shortened version where the fact that some tasks are part of $\underline{B}(t)$ is not considered. Taking it into account can be done simply using a penalty in the second priority queue to ensure that those tasks are chosen first. Two parallel queues can also be used and the one containing the tasks in $\underline{B}(t)$ is emptied first. Additionally a counter must be used to record when all mandatory tasks have been processed.

The time complexity of the algorithm is $\mathcal{O}(n \log n)$ with $n = |\overline{B}(t)|$ which in the worst case is equal to $N - 1$ ($N$ is the number of tasks that must be processed). Indeed, the operation put() and pop() of the priority queues can be implemented in $\mathcal{O}(\log n)$. There are exactly $n$ tasks that are put in Q1 (lines 5-7) and at most $2n$ tasks that are put in Q2 because there are exactly $n$ tasks that can be extracted from Q1 (lines 9-14) and at most $n$ reinsertions of task due to interruption (lines 22-26).

*Example 1.* To show the computation of $b\_ect(B(t), p)$, let us suppose the following tasks:

- $t_0$ which is the task under consideration; $dom(B(t_0)) = [\{t_4\}, \{t_1, t_2, t_3, t_4\}]$ and $dom(P(t_0)) = [1, 4]$
- $t_1$ with $est(t_1) = 0$ and $d(t_1) = 5$.
- $t_2$ with $est(t_2) = 1$ and $d(t_2) = 3$.
- $t_3$ with $est(t_3) = 2$ and $d(t_3) = 1$.
- $t_4$ with $est(t_4) = 3$ and $d(t_4) = 3$.

Following a chronological order, $t_1$ is scheduled first, starting at the time 0. On time 1, $t_2$ is available and as its duration ($d(t_2) = 2$) is shorter than the remaining duration of $t_1$ ($5 - 1 = 4$), $t_1$ is stopped and $t_2$ is started. On time 2, $t_2$ is interrupted to let process $t_3$ whose duration is shorter than its remaining duration ($3 - 1 = 2 > 1$). On time 3, $t_3$ has been fully processed. Tasks $t_1$, $t_2$ and $t_4$ are available but $t_4$ is chosen as it is the only mandatory task among them. Indeed, by definition of $dom(B(t))$, $t_4$ is the only task which must be performed before $t_0$. This task is run for 3 units of time. When it is finished, $t_2$ is run before $t_1$ as its remaining duration is less than the remaining duration of $t_1$. After two more units of time, $t_2$ is fully processed and $t_1$ is processed until time 12. The next table gives the processing times of each task in a preemptive way.

| Time | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Task | $t_1$ | $t_2$ | $t_3$ | $t_4$ | | | $t_2$ | | $t_1$ | | | | |

Recording the values when tasks are fully processed, we obtain the following values:

- $b\_ect(B(t_0), 1) = b\_ect(B(t_0), 2) = 6$. Indeed, the mandatory task ($t_4$) was only finished in second position.
- $b\_ect(B(t_0), 3) = 8$

– $b\_ect(B(t_0), 4) = 12$

Although the computation interrupts several tasks, the obtained bounds correspond to non-preemptive schedules (as expected by equation (5)). The table below shows the reordering for each position.

| Time | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $p = 1$ | | | | | $t_4$ | | | | | | | | |
| $p = 2$ | | | $t_3$ | | $t_4$ | | | | | | | | |
| $p = 3$ | | $t_2$ | | | $t_3$ | | $t_4$ | | | | | | |
| $p = 4$ | $t_1$ | | | | | | $t_2$ | | $t_3$ | | $t_4$ | | |

For instance, with the 4 tasks being scheduled, it is possible to run $t_1$ from time 0 until time 5 where $t_2$ is run until time 8. At time 8, $t_3$ is started for 1 time unit and afterward $t_4$ is being run until the time 12 which corresponds to the computed value. □

*Example 2.* Figure 1 presents a small example where the new propagator permits to remove inconsistent values. In this example, there are five tasks to be processed. Their respective domains and duration are the following.

– $d(t_1) = 3$ and $dom(S(t_1)) = [8, 17]$
– $d(t_2) = 5$ and $dom(S(t_2)) = [0, 15]$
– $d(t_3) = 4$ and $dom(S(t_3)) = [5, 16]$
– $d(t_4) = 4$ and $dom(S(t_4)) = [1, 16]$
– $d(t_5) = 2$ and $dom(S(t_5)) = [7, 18]$

Applying NFNL or EF on this set of tasks does not reduce any domain of the starting time variables. However, our propagator allows to remove the value 8 from the domain of $S(t_1)$. Using the algorithm to compute the earliest and latest possible starting time of $t_1$ in each position, the obtained values are given next.

– $est(t_1, 0) = 8$ and $lst(t_1, 0) = 2$
– $est(t_1, 1) = 8$ and $lst(t_1, 1) = 7$
– $est(t_1, 2) = 9$ and $lst(t_1, 2) = 11$
– $est(t_1, 3) = 11$ and $lst(t_1, 3) = 15$
– $est(t_1, 4) = 15$ and $lst(t_1, 4) = 17$

From those values, it can be derived that $t_1$ cannot be processed in position 0 or 1. Thus the domain of its starting time can be reduced to the union of the ranges defined in position 2, 3 and 4, resulting in $dom(S(t_1)) = [9, 17]$. In comparison with the initial domain, the value 8 has been removed. □

The computing of $est(t, p)$ and $lst(t, p)$ for each $p \in dom(P(t))$ is done in $\mathcal{O}(N \log N)$ with $N$ the number of tasks and the reduction of the domains can be done in $\mathcal{O}(N)$. The time complexity of the whole reduction algorithm for a task $t$ is thus $\mathcal{O}(N \log N)$. This yields a total complexity of $\mathcal{O}(N^2 \log N)$ for one pass of our reduction algorithm, as there are $N$ tasks to consider. In comparison, the well-known techniques NFNL and EF can be both implemented to run with a time complexity of $\mathcal{O}(N \log N)$.

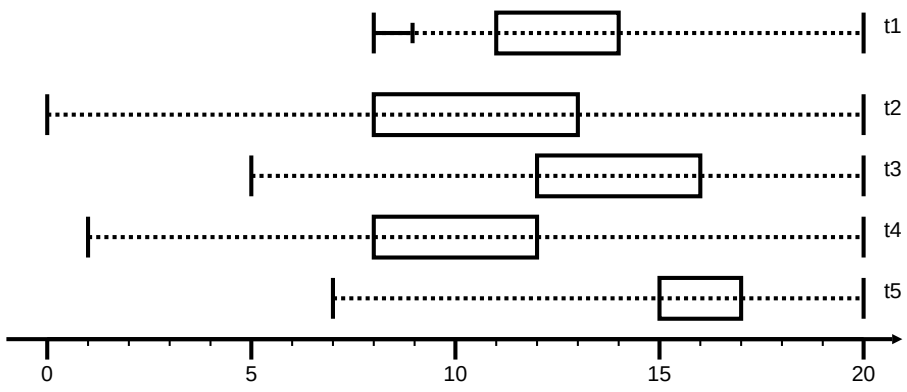


**Fig. 1.** Example of reduction, see Example 2 for details

## 4 Experiments

To assess the practical usefulness of the new propagator, we implemented it in the open constraint environment Gecode [14]. Two versions of the propagator have been written. The first that we will refer to as PS (standing for Position Shaving) may remove values inside the domains of the starting time variables, while the second, PSB (for Position Shaving with Bounds reduction), is limited to reduce the bounds of the starting time variables. We implemented also the NFNL and EF techniques following the algorithms described in [15]. Note that the implementations of EF and NFNL described in that book run in $\mathcal{O}(N^2)$ but they use much simpler data structures than the theoretically most efficient algorithm described respectively in [3] and [7]. Finally, we modeled the Open-Shop Problem as described in the first section with the NFNL, EF and PS or PSB propagators and the AllDifferent constraint. PS and PSB are never used together as they are two versions of the same propagator. Concerning the branching, we applied a simple heuristic that uses the position variables. It orders the tasks in the machine before ordering them in the jobs. Among the tasks whose position is not fixed, it chooses the task for which there is the smallest number of remaining possible positions. In case of tie, the shortest task is chosen. The value-heuristic chooses the smallest value in the position variable.

Our tests have been run using the instances of the Guéret and Prins benchmark [12]. It is composed of 80 square problems, i.e. the number of jobs and machines are equal. There are 10 instances for each size ranging from 3x3 tasks to 10x10 tasks. Every runs have been performed on an Intel Xeon 3 Ghz with 512 KB of cache.

The first experiment consists in observing the total runtime and the size of the search tree to solve each instance of the benchmark, using different combinations of propagators. The running time is limited to one hour for each instance. The results are presented in Tables 1 and 2. Table 1 gives the number of solved instances and the average number of nodes in the search tree. The mean is

computed over the instances commonly solved whenever the number of solved instances differs (only for problem size 7x7). In table 2, the same scheme is used but the mean running time is presented instead of the size of the search tree. The running time is given in seconds.

In the two tables, columns 2 and 3 present the results when only PS is used but nor EF neither NFNL. Columns 4 and 5 presents the results when only PSB is used. In the third setting (columns 6-7), NFNL and EF are activated but not PS, nor PSB. In the columns 8-9 and 10-11, NFNL and EF are used in conjunction respectively with PS and with PSB.

**Table 1.** Number of solved instances and mean size of the search tree

| | PS | | PSB | | NFNL+EF | | PS+NFNL+EF | | PSB+NFNL+EF | |
|---|---|---|---|---|---|---|---|---|---|---|
| Size | Solved | Nodes | Solved | Nodes | Solved | Nodes | Solved | Nodes | Solved | Nodes |
| 3x3 | 10 | 39 | 10 | 38 | 10 | 38 | 10 | 39 | 10 | 38 |
| 4x4 | 10 | 128 | 10 | 127 | 10 | 134 | 10 | 127 | 10 | 126 |
| 5x5 | 10 | 451 | 10 | 456 | 10 | 371 | 10 | 369 | 10 | 373 |
| 6x6 | 10 | 3483 | 10 | 3896 | 10 | 2612 | 10 | 3402 | 10 | 3816 |
| 7x7 | 3 | - | 3 | - | 7 | 280914 | 8 | 208571 | 8 | 208582 |
| 8x8 | 0 | - | 0 | - | 1 | 120156 | 1 | 12953 | 1 | 12929 |
| 9x9 | 0 | - | 0 | - | 1 | 747146 | 0 | - | 0 | - |
| 10x10 | 0 | - | 0 | - | 0 | - | 0 | - | 0 | - |
| Tot | 43 | | 43 | | 49 | | 49 | | 49 | |

**Table 2.** Number of solved instances and mean running time in seconds

| | PS | | PSB | | NFNL+EF | | PS+NFNL+EF | | PSB+NFNL+EF | |
|---|---|---|---|---|---|---|---|---|---|---|
| Size | Solved | Time | Solved | Time | Solved | Time | Solved | Time | Solved | Time |
| 3x3 | 10 | 0.008 | 10 | 0.008 | 10 | 0.006 | 10 | 0.01 | 10 | 0.008 |
| 4x4 | 10 | 0.075 | 10 | 0.047 | 10 | 0.054 | 10 | 0.069 | 10 | 0.068 |
| 5x5 | 10 | 0.38 | 10 | 0.32 | 10 | 0.19 | 10 | 0.36 | 10 | 0.32 |
| 6x6 | 10 | 3.9 | 10 | 3.5 | 10 | 1.9 | 10 | 4.3 | 10 | 3.9 |
| 7x7 | 3 | - | 3 | - | 7 | 338 | 8 | 496 | 8 | 432 |
| 8x8 | 0 | - | 0 | - | 1 | 106 | 1 | 43 | 1 | 37 |
| 9x9 | 0 | - | 0 | - | 1 | 1708 | 0 | - | 0 | - |
| 10x10 | 0 | - | 0 | - | 0 | - | 0 | - | 0 | - |
| Tot | 43 | | 43 | | 49 | | 49 | | 49 | |

Whenever NFNL and EF are used, the same total number of instances are solved with or without our new propagator. However, the solved instances are not always the same. From the two first settings, it can be concluded that the new

propagator is not able to solve hard problems alone. In conjunction with NFNL and EF, Table 1 shows that PS and PSB are able to reduce the size of search tree, sometimes substantially, as it is the case for the unique solved instance of size 8x8. Concerning the size 6x6, surprisingly the mean size is greater when using PS and PSB. Looking at the detail for each instance of this size, it appears that only the first instance(GP06-01) has a greater search tree when using the new propagators. For GP06-01, the search tree is ten times bigger when using PS or PSB while it is on average 30% smaller for the nine other instances of size 6x6.

When the running times are considered, Table 2 shows that it is always greater when using PS or PSB than without them, except for the solved instance of size 8 where the time is 2 to 3 times smaller, while the search tree size was almost 9 times smaller.

Note that the reported times are much longer than those presented in [15] because we did not use an environment dedicated to scheduling but a general purpose constraint engine. However, implementing our new propagator in a dedicated environment would be beneficial.

The next experiment (Table 3) compares the mean runtime to reach the fixpoint when NFNL, EF and PS(B) are activated with the mean runtime when PS(B) is not used. This comparison is performed on the search tree obtained when NFNL, EF and PS(B) are activated with a maximum number of backtracks of 300,000. For each instance, the runtimes to reach the fixpoints are summed along every states in the search tree.

At the same time, the pruning is also compared. As for the runtime, this pruning is computed along the search tree obtained when every propagators are activated. The number of failed states with and without PS(B) activated are counted. Additionally in each state the supplementary reduction performed after adding PS(B) is counted for each type of variables ($S(t)$, $B(t)$ and $P(t)$) and these quantities are summed upon the whole search tree. The reduction is computed as the difference between the size of the domains of the variables in the initial state in a node of the search tree and their size after performing propagation until the fixpoint in the same node. If a failure is detected, the node is not taken into account for the reduction counts.

Table 3 presents the results averaged by size. The three first pairs of columns presents the additional pruning of the variables $S(t)$, $B(t)$ and $P(t)$. The next two columns shows the additional failures detected and the last columns reports the additional time spent to reach those improvements. Two cells are empty because the running time was to short to compute them accurately.

It can be seen that the results are quite similar between PS and PSB, except in the columns of the starting time variables, since PS may prune inside the domain of $S(t)$ while PSB cannot. However, this difference does not influence the other variables nor the failures. Indeed, no other constraint considers forbidden values inside domains. Concerning the increase of the running time to reach a fixpoint, it is smaller with PSB because less values are removed by PSB. Taking into account the pruning potential and the used time, we can conclude that

**Table 3.** Additional Pruning and time spent with PS and PSB(in %)

| | Red. S(t) | | Red. B(t) | | Red. P(t) | | Fails | | Time | |
|---|---|---|---|---|---|---|---|---|---|---|
| Size | PS | PSB | PS | PSB | PS | PSB | PS | PSB | PS | PSB |
| 3 | 7.6 | 1.5 | 0.3 | 0.3 | 5.7 | 3.6 | 0 | 0 | - | - |
| 4 | 13.6 | 5.6 | 7.0 | 7.4 | 14.2 | 12.7 | 2.1 | 2.1 | 184.0 | 165.7 |
| 5 | 14.1 | 5.6 | 4.8 | 4.9 | 11.2 | 9.8 | 0.7 | 0.8 | 192.1 | 182.2 |
| 6 | 27.3 | 14.9 | 9.0 | 9.2 | 15.6 | 14.1 | 8.0 | 8.2 | 241.3 | 181.0 |
| 7 | 108.7 | 58.3 | 21.3 | 21.8 | 42.5 | 42.7 | 13.9 | 14.2 | 333.2 | 325.3 |
| 8 | 116.9 | 34.9 | 17.4 | 16.7 | 30.1 | 26.1 | 13.3 | 13.9 | 281.1 | 254.0 |
| 9 | 78.9 | 25.4 | 13.9 | 13.2 | 20.5 | 18.0 | 37.7 | 36.3 | 291.5 | 272.0 |
| 10 | 64.6 | 17.9 | 9.9 | 10.3 | 20.9 | 19.5 | 37.4 | 37.3 | 155.5 | 196.5 |
| Mean | 54.0 | 20.5 | 10.4 | 10.5 | 20.1 | 18.3 | 14.1 | 14.1 | 239.8 | 225.3 |

PSB is more efficient than PS. Furthermore, there are about 14% more failures detected with either version of our propagator. When no failure is detected, the domains of the variables are also substantially reduced.

Looking at the evolution of the results in function of the size of the problem, the amount of reduction of the domains increases until problems of size 7 and then decreases. The time spent follows the same scheme while the number of failures keeps increasing. Because from size 7 the search trees may be not full (because the search is cut) and the explored part is smaller for increasing size, we can suppose that our propagator detect more failures early in the search but reduces more domains at the end or in the middle of the search than in the first steps. Observing the failures for the smallest sizes, it can also be seen that PS and PSB do not reduce further the small search trees of these instances. When size grows ($\geq 6$) and complexity increases, PS and PSB prove their usefulness.

In conclusion, the experiments show that although the introduction of PS or PSB does not increase the number of solved instances, the addition of such a propagator substantially improves the pruning at the nodes of the search tree, as well as the number of detection of inconsistencies.

## 5 Conclusion

This work addresses the Open-Shop Problem by Constraint Programming. It presents a new propagator, in two versions, that uses the absolute position of the tasks to detect new inconsistencies not discovered by standard algorithms, known as Not-First-Not-Last or Edge-Finding. Based on the principle of shaving, this propagator prunes the variables for the starting time of tasks and for the position of tasks. In its first version, holes in the starting time variable are allowed while this is not the case in the second version that only reduces the bounds of the domains.

Experiments on a standard benchmark show that the new propagator helps efficiently in the reduction of the domains and may detect about 14% more

inconsistent states in the search tree but at a higher cost. Concerning the size of the search tree, the reduction of the domains is not always reflected by a reduction of the tree size. The search can be up to 10 time smaller but for the majority of the problems the reduction is not as important. In a few cases, the search tree is even increased with the new constraint.

Another observation comes from the comparison between the two versions of the propagators. Making holes in the domain of the starting time variables is not rewarding regarding the reduction of the other variables and of the search tree. The cause is that no other constraint makes use of this additional information.

In conclusion, we argue that our propagator would be especially useful in conjunction with other constraints that take into account the position of tasks or the holes in the domains. This may be a good way to improve resolution of hard disjunctive scheduling problems. The branching heuristic should also be adapted in order to avoid an augmentation of the size of the search tree when the filtering is strengthened.

Possible future work includes the definition of tighter bounds for the earliest completion time of a subset of tasks of fixed size. It also covers the definition of better branching heuristics and additional position-based constraints.

## References


1. J. Carlier and E. Pinson. An algorithm for solving the job-shop problem. Management Science, 35(2):164-176, 1989.
2. D. Applegate and B. Cook. A computational study of the job shop scheduling problem. ORSA J. Comput., 3(2):149-156, 1991.
3. J. Carlier and E. Pinson. Adjustment of heads and tails for the job-shop problem. European J. Oper. Res.,78:146-161, 1994
4. Y. Caseau and F. Laburthe. Improved CLP scheduling with task intervals. Proc. 11th Intl. Conf. on Logic Programming, 1994.
5. J. Carlier and E. Pinson. A practical use of Jackson's preemptive schedule for solving the job-shop problem. Ann. Oper. Res.,26:269-287, 1990.
6. U. Dorndorf, E. Pesch and T. Phan-Huy. Solving the open shop scheduling problem. J. Scheduling, 4:157-174, 2001.
7. P. Vilim. $\mathcal{O}(n \log n)$ filtering algorithms for unary resource constraint. Proc. CPAIOR 2004, LNCS 3011, 335-347, 2004.
8. P. Martin and D. Shmoys. A new approach to computing optimal schedules for the job-shop scheduling problem. Proc. 5th Conf. Integer Programming and Combinatorial Optimization, 1996.
9. Jianyang Zhou. A permutation-based approach for solving the job-shop problem. Constraints, 2:185-213, 1997.
10. W. Nuijten and C. Le Pape. Constraint-based job shop scheduling with ILOG Scheduler. J. Heuristics, 3:271-286, 1998.
11. A. Wolf. Better propagation for non-preemptive single-ressource constraint problems. Proc. of CSCLP 2004, LNAI 3419, 201-215, 2005.
12. C. Guéret and C. Prins. A new lower bound for the open-shop problem. Annals of Operation Research, 92:165-183, 1999.
13. J. R. Jackson. An extension of Johnson's results on job lot scheduling. Naval Research Logistics Quarterly, 3:201-203, 1956.

14. http://www.gecode.org
15. P. Baptiste, C. Le Pape and W. Nuijten. Constraint-based scheduling. Kluwer Academics Publisher, 2001.