$O(n \log n)$ Filtering Algorithms for Unary Resource Constraint

Petr Vilím

Charles University
Faculty of Mathematics and Physics
Malostranské náměstí 2/25, Praha 1, Czech Republic
vilim@kti.mff.cuni.cz

Abstract. So far, edge-finding is the only one major filtering algorithm for unary resource constraint with time complexity $O(n \log n)$. This paper proposes $O(n \log n)$ versions of another two filtering algorithms: not-first/not-last and propagation of detectable precedences. These two algorithms can be used together with edge-finding to further improve the filtering. This paper also propose new $O(n \log n)$ implementation of fail detection (overload checking).

1 Introduction

In scheduling, *unary resource* is an often used generalization of a machine or a job. Unary resource models a set of non-interruptible *activities* T which must not overlap in a schedule.

Each activity $i \in T$ has following requirements:

- earliest possible starting time est_i
- latest possible completion time lct_i
- processing time p_i

A (sub)problem is to find a schedule satisfying all these requirements. One of the most used technique to solve this problem is *constraint programming*.

In constraint programming, we associate an *unary resource constraint* with each unary resource. A purpose of such constraint is to reduce a search space by tightening time bounds est_i and lct_i. This process of elimination of infeasible values is called a *propagation*, actual propagation algorithm is often called a *filtering* algorithm.

Naturally, it is not efficient to remove all infeasible values. Rather we use fast but not complete algorithms which can find only some impossible assignments. These filtering algorithms are repeated in every node of a search tree, therefore their speed and filtering power are crucial.

Today, edge-finding and not-first/not-last are the mainly used filtering algorithms for the unary resource constraint. The edge-finding algorithm has time complexity $O(n \log n)$ [3], whereas time complexity of the not-first/not-last [1, 8]

algorithm is $O(n^2)$ (where $n = |T|$ is the number of activities). This paper introduce a new $O(n \log n)$ version of the not-first/not-last algorithm and also a third $O(n \log n)$ filtering algorithm. All these three algorithms filter out different inconsistent values and therefore they can be used together to join their filtering powers.

Let us establish a notation concerning a subset of activities. Let $\Theta \subseteq T$ be an arbitrary subset of activities. An earliest starting time est_Θ, a latest completion time lct_Θ and a processing time p_Θ of the set Θ are:

$$\begin{aligned} \mathrm{est}_\Theta &= \min \{\mathrm{est}_j,\ j \in \Theta\} \\ \mathrm{lct}_\Theta &= \max \{\mathrm{lct}_j,\ j \in \Theta\} \\ \mathrm{p}_\Theta &= \sum_{j \in \Theta} \mathrm{p}_j \end{aligned}$$

An earliest completion time of the set Θ is:

$$\mathrm{ECT}_\Theta = \max \{\mathrm{est}_{\Theta'} + \mathrm{p}_{\Theta'},\ \Theta' \subseteq \Theta\}$$

2 Θ-Tree

Algorithms in this paper are based on an idea of organizing a set $\Theta \subseteq T$ in a balanced binary tree. Because the set represented by the tree will be always named Θ, we will call the tree Θ-tree. The purpose of a Θ-tree is to quickly recompute ECT_Θ when an activity is inserted or removed from the set Θ.

A Θ-tree is a balanced binary search tree with respect to est_i. Each activity $i \in \Theta$ is represented by a single node. In the following we do not make a difference between an activity the tree node representing that activity.

Notice, that so far Θ-tree does not require any particular way of balancing. Any type of balanced binary tree (AVL-tree, black-red-tree *etc.*) is possible. The only requirement is a time complexity $O(n \log n)$ for inserting or deleting a node, and time complexity $O(1)$ for finding a root node.

It is also possible to start with a perfect balanced tree build from all activities T with "empty" nodes. Inserting a node means to fill such empty node, deleting a node makes it empty over again. This is the implementation choosed by the author.

Let $\mathrm{left}(i)$ be a left son of an activity i (if it has one), similarly let $\mathrm{right}(i)$ be a right son of the activity i. We will also need a notation for subtrees: let $\mathrm{Subtree}(i)$ be a set of all activities in the subtree rooted in i; $\mathrm{Left}(i) = \mathrm{Subtree}(\mathrm{left}(i))$, $\mathrm{Right}(i) = \mathrm{Subtree}(\mathrm{right}(i))$.

Because a Θ-tree is a balanced binary tree with respect to est_i, we have:

$$\begin{aligned} &\forall i \in \Theta\, \forall j \in \mathrm{Left}(i) : \quad \mathrm{est}_j \leq \mathrm{est}_i \\ &\forall i \in \Theta\, \forall j \in \mathrm{Right}(i) : \mathrm{est}_j \geq \mathrm{est}_i \end{aligned}$$

Besides the activity itself, each node i of a Θ-tree also holds following two values:

$$\Sigma \mathrm{P}_i = \sum_{j \in \mathrm{Subtree}(i)} \mathrm{p}_j$$
$$\mathrm{ECT}_i = \mathrm{ECT}_{\mathrm{Subtree}(i)} = \max\{\mathrm{est}_{\Theta'} + \mathrm{p}_{\Theta'},\ \Theta' \subseteq \mathrm{Subtree}(i)\}$$

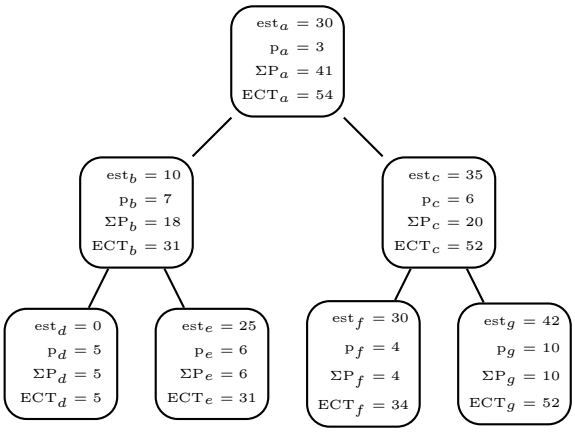

Fig. 1. An example of a Θ-tree

Values $\Sigma\mathrm{P}_i$ and ECT_i can be computed from direct descendants of the node i:

$$\begin{aligned}\Sigma\mathrm{P}_i &= \Sigma\mathrm{P}_{\mathrm{left}(i)} + \mathrm{p}_i + \Sigma\mathrm{P}_{\mathrm{right}(i)} \\ \mathrm{ECT}_i &= \max\big\{\, \mathrm{ECT}_{\mathrm{left}(i)} + \mathrm{p}_i + \Sigma\mathrm{P}_{\mathrm{right}(i)}, \\ &\qquad \mathrm{est}_i + \mathrm{p}_i + \Sigma\mathrm{P}_{\mathrm{right}(i)}, \\ &\qquad \mathrm{ECT}_{\mathrm{right}(i)} \,\big\}\end{aligned}$$

Thanks to this recursive nature, values ECT and ΣP can be computed within usual operations with balanced binary search trees without changing their time complexities. Following table summarizes time complexities of operations with Θ-tree:

Operation	**Time Complexity**
$\Theta := \emptyset$	$O(1)$ or $O(n \log n)$
$\Theta := \Theta \cup \{i\}$	$O(\log n)$
$\Theta := \Theta \setminus \{i\}$	$O(\log n)$
ECT_Θ	$O(1)$
$\mathrm{ECT}_{\Theta \setminus \{i\}}$	$O(\log n)$

3 Overload checking using Θ-tree

Let us consider an arbitrary set $\Omega \subseteq T$. Overload rule says that if the set Ω cannot be processed within its time bounds then no solution exists:

$$\mathrm{lct}_\Omega - \mathrm{est}_\Omega < \mathrm{p}_\Omega \quad \Rightarrow \quad \text{fail}$$

Let us suppose for a while that we are given an activity $i \in T$ and we want to check this rule only for these sets $\Omega \subseteq T$ which have $\mathrm{lct}_\Omega = \mathrm{lct}_i$. Now consider a set Θ:

$$\Theta = \{j,\ j \in T \ \&\ \mathrm{lct}_j \leq \mathrm{lct}_i\}$$

Overloaded set Ω with $\mathrm{lct}_\Omega = \mathrm{lct}_i$ exists if and only if $\mathrm{ECT}_\Theta > \mathrm{lct}_i = \mathrm{lct}_\Theta$. The idea of an algorithm is to gradually increase the set Θ by increasing lct_Θ. For each lct_Θ we check whether $\mathrm{ECT}_\Theta > \mathrm{lct}_\Theta$ or not.

$\Theta := \emptyset$;
for $i \in T$ in ascending order of lct_i **do begin**
  $\Theta := \Theta \cup \{i\}$;
  if $\mathrm{ECT}_\Theta \geq \mathrm{lct}_i$ **then**
    fail; {No solution exists}
end;

Time complexity of this algorithm is $O(n \log n)$: the activities have to be sorted and n-times an activity is inserted into the set Θ.

4 Not-first/not-last using Θ-tree

Not-first and not-last are two symmetric propagation algorithms for an unary resource. From these two, we will consider only the not-last algorithm.

Let us consider a set $\Omega \subseteq T$ and an activity $i \in (T \setminus \Omega)$. The activity i cannot be scheduled after the set Ω (*i.e.* i is not last within $\Omega \cup \{i\}$) if:

$$\mathrm{est}_\Omega + \mathrm{p}_\Omega > \mathrm{lct}_i - \mathrm{p}_i \tag{1}$$

In that case, at least one activity from the set Ω must be scheduled after the activity i. Therefore the value lct_i can be updated:

$$\mathrm{lct}_i := \min\left\{\mathrm{lct}_i,\ \max\left\{\mathrm{lct}_j - \mathrm{p}_j,\ j \in \Omega\right\}\right\} \tag{2}$$

There are two versions of the not-first/not-last algorithms: [1] and [8]. Both of them have time complexity $O(n^2)$. The first algorithm [1] finds all the reductions resulting from the previous rules in one pass. Still, after this propagation, next run of the algorithm may find more reductions (not-first and not-last rules are not idempotent). Therefore the algorithm should be repeated until no more reduction is found (*i.e.* a fixpoint is reached). The second algorithm [8] is simpler and faster, but more iterations of the algorithm may be needed to reach a fixpoint.

The algorithm presented here can also needs more iteration to reach a fixpoint then the algorithm [1] maybe even more then the algorithm [8]. However, time complexity is reduced from $O(n^2)$ to $O(n \log n)$.

Suppose we want to update the lct_i according to the rule not-last. To really achieve some change of lct_i using the rule (2), the set Ω must fulfil following property:

$$\max\left\{\mathrm{lct}_j - \mathrm{p}_j,\ j \in \Omega\right\} < \mathrm{lct}_i$$

Therefore:

$$\Omega \subseteq \left\{j,\ j \in T\ \&\ \mathrm{lct}_j - \mathrm{p}_j < \mathrm{lct}_i\ \&\ j \neq i\right\}$$

We will use the same trick as [8]: lets not slow down the algorithm by searching the *best* update of lct_i. Rather, find *some* update: if lct_i can be updated better, let it be done in the next run of the algorithm. Therefore our goal is to update lct_i to $\max\left\{\mathrm{lct}_j - \mathrm{p}_j,\ j \in T\ \&\ \mathrm{lct}_j - \mathrm{p}_j < \mathrm{lct}_i\right\}$.

Let us define the set Θ:

$$\Theta = \left\{j,\ j \in T\ \&\ \mathrm{lct}_j - \mathrm{p}_j < \mathrm{lct}_i\right\}$$

Thus: the lct_i can be changed according to the rule not-last if and only if there is some set $\Omega \subseteq (\Theta \setminus \{i\})$ for which the inequality (1) holds:

$$\mathrm{est}_\Omega + \mathrm{p}_\Omega > \mathrm{lct}_i - \mathrm{p}_i$$

And such a set Ω exists iff:

$$\mathrm{ECT}_{\Theta\setminus\{i\}} > \mathrm{lct}_i - \mathrm{p}_i$$

The algorithm proceeds as follows. Activities i are taken in the ascending order of lct_i. For each one activity i the set Θ is computed using the set Θ of previous activity i. Then $\mathrm{ECT}_{\Theta\setminus\{i\}}$ is checked and lct_i is eventually updated:

```
1  Θ := ∅;
2  Q := queue of all activities j ∈ T in ascending order of lct_j − p_j;
3  for i ∈ T in ascending order of lct_i do begin
4    while lct_i > lct_Q.first − p_Q.first do begin
7      j := Q.first;
8      Θ := Θ ∪ {j};
9      Q.dequeue;
10   end;
11   if ECT_Θ\{i} > lct_i − p_i then
12     lct_i := lct_j − p_j;
13 end;
```

Lines 7–9 are repeated n times maximum, because each time an activity is removed from the queue. Check at the line 11 can be done in $O(\log n)$. Therefore the time complexity of the algorithm is $O(n \log n)$.

Without changing the time complexity, the algorithm can be slightly improved: the not-last rule can be also checked for the activity j just before the insertion of the activity j into the set Θ (*i.e.* after the line 4):

```
5      if ECT_Θ > lct_Q.first − p_Q.first then
6        lct_Q.first := lct_j − p_j ;
```

5 Detectable Precedences

An idea of detectable precedences was introduced in [9] for a *batch resource with sequence dependent setup times*, what is an extension of an unary resource.

Following figure is taken from [9]. It shows a situation when neither edge-finding nor not-first/not-last can change any time bound, but a propagation of detectable precedences can.

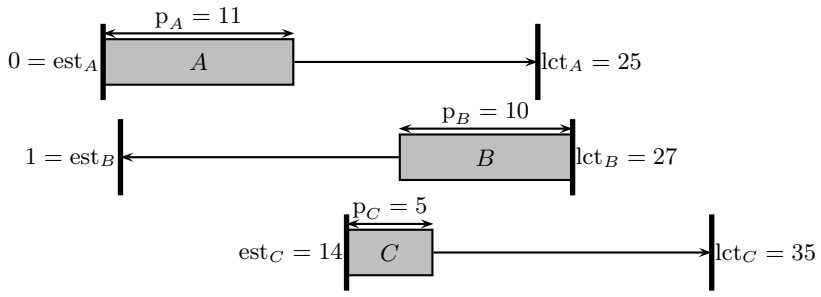

Fig. 2. A sample problem for detectable precedences

Edge-finding algorithm recognizes that the activity A must be processed before the activity C, i.e. $A \ll C$, and similarly $B \ll C$. Still, each of these precedences alone is weak: they do not enforce any change of any time bound. However, from the knowledge $\{A, B\} \ll C$ we can deduce $\mathrm{est}_C \geq \mathrm{est}_A + \mathrm{p}_A + \mathrm{p}_B = 21$.

A precedence $j \ll i$ is called *detectable*, if it can be "discovered" only by comparing time bounds of these two activities:

$$\mathrm{est}_i + \mathrm{p}_i > \mathrm{lct}_j - \mathrm{p}_j \quad \Rightarrow \quad j \ll i \tag{3}$$

Notice that both precedences $A \ll C$ and $B \ll C$ are detectable.

There is a simple quadratic algorithm, which propagates all known precedences on a resource. For each activity i build a set $\Omega = \{j \in T,\ j \ll i\}$. Note that precedences $j \ll i$ can be of any type: detectable precedences, search decisions or initial constraints. Using such set Ω, est_i can be adjusted: $\mathrm{est}_i := \max\{\mathrm{est}_i,\ \mathrm{ECT}_\Omega\}$ because $\Omega \ll i$.

```
for i ∈ T do begin
  m := −∞;
  for j ∈ T in non-decreasing order of est_j do
    if j ≪ i then
      m := max{m, est_j} + p_j;
  est_i := max{m, est_i};
end;
```

A symmetric algorithm adjusts lct_i.

However, propagation of detectable precedences only can be done within $O(n \log n)$. Let Θ be following set of activities:

$$\Theta = \left\{ j,\ j \in T \ \&\ \mathrm{est}_i + \mathrm{p}_i > \mathrm{lct}_j - \mathrm{p}_j \right\}$$

Thus $\Theta \setminus \{i\}$ is a set of all activities which must be processed before the activity i because of detectable precedences. Using the set $\Theta \setminus \{i\}$ the value est_i can be adjusted:

$$\mathrm{est}_i := \max\left\{ \mathrm{est}_i,\ \mathrm{ECT}_{\Theta \setminus \{i\}} \right\}$$

There is also a symmetric rule for precedences $j \gg i$, but we will not consider it here, nor resulting symmetric algorithm.

An algorithm is based on an observation that the set Θ does not have to be constructed from scratch for each activity i. Rather, the set Θ can be computed incrementally.

```
1  Θ := ∅;
2  Q := queue of all activities j ∈ T in ascending order of lct_j − p_j;
3  for i ∈ T in ascending order of est_i + p_i do begin
4    while est_i + p_i > lct_Q.first − p_Q.first do begin
5      Θ := Θ ∪ {Q.first};
6      Q.dequeue;
7    end;
8    est_i := max{est_i, ECT_{Θ\{i}}};
9  end;
```

Initial sorts takes $O(n \log n)$. Lines 5 and 6 are repeated n times maximum, because each time an activity is removed from the queue. Line 8 can be done in $O(\log n)$. Therefore the time complexity of the algorithm is $O(n \log n)$.

6 Properties of Detectable Precedences

There is an interesting connection between edge-finding algorithm and detectable precedences:

Proposition 1. *When edge-finding is unable to find any further time bound adjustment then all precedences which edge-finding found are detectable.*

Proof. First, brief introduction of edge-finding algorithm. Consider a set $\Omega \subseteq T$ and an activity $i \notin \Omega$. The activity i has to be scheduled after all activities from Ω, if:

$$\forall \Omega \subset T,\ \forall i \in (T \setminus \Omega): \quad \min(\mathrm{est}_\Omega,\ \mathrm{est}_i) + \mathrm{p}_\Omega + \mathrm{p}_i > \mathrm{lct}_\Omega \quad \Rightarrow \quad \Omega \ll i \tag{4}$$

Once we know that the activity i must be scheduled after the set Ω, we can adjust est_i:

$$\Omega \ll i \quad \Rightarrow \quad \mathrm{est}_i := \max(\mathrm{est}_i,\ \max\{\mathrm{est}_{\Omega'} + \mathrm{p}_{\Omega'},\ \Omega' \subseteq \Omega\}) \tag{5}$$

Edge-finding algorithm propagates according to this rule and its symmetric version. There are several implementations of edge-finding algorithm, two different quadratic algorithms can be found in [6, 7], [3] presents a $O(n \log n)$ algorithm.

Let us suppose that edge-finding proved $\Omega \ll i$. We will show that for an arbitrary activity $j \in \Omega$, edge-finding made est_i big enough to make the precedence $j \ll i$ detectable.

Edge-finding proved $\Omega \ll i$ so the condition (4) was true before the filtering:

$$\min(\mathrm{est}_\Omega,\ \mathrm{est}_i) + \mathrm{p}_\Omega + \mathrm{p}_i > \mathrm{lct}_\Omega$$

However, increase of any est or decrease of any lct cannot invalidate this condition, therefore it has to be valid now. And so:

$$\mathrm{est}_\Omega > \mathrm{lct}_\Omega - \mathrm{p}_\Omega - \mathrm{p}_i \tag{6}$$

Because edge-finding is unable to further change any time bound, according to (5) we have:

$$\begin{aligned} \mathrm{est}_i &\geq \max\{\mathrm{est}_{\Omega'} + \mathrm{p}_{\Omega'},\ \Omega' \subseteq \Omega\} \\ \mathrm{est}_i &\geq \mathrm{est}_\Omega + \mathrm{p}_\Omega \end{aligned}$$

In this inequality, est_Ω can be replaced by the right side of the inequality (6):

$$\begin{aligned} \mathrm{est}_i &> \mathrm{lct}_\Omega - \mathrm{p}_\Omega - \mathrm{p}_i + \mathrm{p}_\Omega \\ \mathrm{est}_i &> \mathrm{lct}_\Omega - \mathrm{p}_i \end{aligned}$$

And $\mathrm{lct}_\Omega \geq \mathrm{lct}_j$ because $j \in \Omega$:

$$\begin{aligned} \mathrm{est}_i &> \mathrm{lct}_j - \mathrm{p}_i \\ \mathrm{est}_i + \mathrm{p}_i &> \mathrm{lct}_j - \mathrm{p}_j \end{aligned}$$

So the condition (3) holds and the precedence $j \ll i$ is detectable.

The proof for the precedences resulting from $i \ll \Omega$ is symmetrical. □

Previous proposition has also one negative consequence. Papers [4, 8, 10] mention following improvement of edge-finding: whenever $\Omega \ll i$ is found, propagate also $j \ll i$ for all $j \in \Omega$. *I.e.* change also lct_j. However, these precedences are detectable and so the second run of edge-finding would propagate them anyway. Therefore such improvement can save some iterations of edge-finding, but do not enforce better pruning in the end.

Several authors (*e.g.* [2, 5]) suggest to compute a transitive closure of precedences. Detectable precedences has also an interesting property in such transitive closure.

Lets us call a precedence $i \ll j$ *propagated* iff the activities i and j fulfill following two inequalities:

$$\mathrm{est}_j \geq \mathrm{est}_i + \mathrm{p}_i$$
$$\mathrm{lct}_i \leq \mathrm{lct}_j - \mathrm{p}_j$$

Note that edge-finding and precedence propagation algorithm make all known precedences propagated.

Following proposition has an easy consequence: detectable precedences can be skipped when computing a transitive closure.

Proposition 2. *Let $a \ll b$, $b \ll c$ and one of these precedences is detectable and the second one propagated. Then the precedence $a \ll c$ is detectable.*

Proof. We distinguish two cases:

1. **$\mathbf{a \ll b}$ is detectable and $\mathbf{b \ll c}$ is propagated.** Because the precedence $b \ll c$ is propagated:
$$\mathrm{est}_c \geq \mathrm{est}_b + \mathrm{p}_b$$
and because the precedence $a \ll b$ is detectable:
$$\mathrm{est}_b + \mathrm{p}_b > \mathrm{lct}_a - \mathrm{p}_a$$
$$\mathrm{est}_c > \mathrm{lct}_a - \mathrm{p}_a$$
Thus the precedence $a \ll c$ is detectable.
2. **$\mathbf{a \ll b}$ is propagated and $\mathbf{b \ll c}$ is detectable.** Because the precedence $a \ll b$ is propagated:
$$\mathrm{lct}_b - \mathrm{p}_b \geq \mathrm{lct}_a$$
And because the second precedence $b \ll c$ is detectable:
$$\mathrm{est}_c + \mathrm{p}_c > \mathrm{lct}_b - \mathrm{p}_b$$
$$\mathrm{est}_c + \mathrm{p}_c > \mathrm{lct}_a$$
Once again, the precedence $a \ll c$ is detectable. □

7 Experimental Results

A reduction of a time complexity of an algorithm is generally a "good think". However for small n, an easy and short algorithm can outperform a complicated algorithm with better time complexity. It is therefore reasonable to ask whether it is the case of the new not-first/not-last algorithm. Another question is a filtering power of the detectable precedences. Following benchmark should bring answers to these questions.

The benchmark is based on a computation of destructive lower bounds for several jobshop benchmark problems. Destructive lower bound is a minimum length of the schedule, for which we are not able to proof infeasibility without backtracking. Lower bounds computation is a good benchmark problem because there is no influence of a search heuristic. Four different destructive lower bounds where computed. Lower bound LB1 is computed using only edge-finding algorithm [6][1] and new version of not-first/not-last:

```
repeat
  repeat
    edge finding;
  until no more propagation;
  not-first/not-last;
until no more propagation;
```

Detectable precedences were used for computation of LB2:

```
repeat
  repeat
    repeat
      detectable precedences;
    until no more propagation;
    not-first/not-last;
  until no more propagation;
  edge finding;
until no more propagation;
```

Note that the order of the filtering algorithms affects total time however it does not influence resulting fixpoint. The reason is that even after an arbitrary propagation, all used reduction rules remain valid and propagates the same or even better.

Another two lower bounds where computed using shaving as suggested in [6]. Shaving is like a proof by contradiction. We choose an activity i, limit its est_i or lct_i and propagate. If an infeasibility is found, then the limitation was invalid and so we can increase est_i or decrease lct_i. Binary search is used to find the best shave. To limit CPU time, shaving is used for each activity only once.

Often detectable precedences improve the filtering, however do not increase the lower bound. Therefore a new column R is introduced. After the propagation

[1] Note that it is a quadratic algorithm

with LB as upper bound, domains are compared with a state when only binary precedences were propagated. The result is an average domain size in percents.

CPU[2] time was measured only for shaving (columns T, T1–T3 in seconds). It is a time needed to proof the lower bound, *i.e.* propagation is done twice: with upper bound LB and LB-1. Times T1–T3 shows the difference between the not-first/not-last algorithms: new algorithm for T1, [8] for T2 and [1] for T3.

For improving readability, when LB1=LB2, then dash is reported in LB2. The same rule was applied to shaving lower bounds and columns R.

				Shaving EF+NFNL			Shaving DP+NFNL+EF				
Prob.	**Size**	**LB1**	**LB2**	**LB**	**R**	**T**	**LB**	**R**	**T1**	**T2**	**T3**
abz5	10 x 10	1126	1127	1195	76.81	1.337	1196	76.95	1.393	1.392	1.447
abz6	10 x 10	889	890	940	58.76	1.540	941	66.92	1.743	1.745	1.808
abz7	20 x 15	651	-	651	62.15	3.539	-	62.08	3.236	3.332	3.561
abz8	20 x 15	608	-	621	85.88	12.28	-	85.51	11.88	12.06	12.66
orb01	10 x 10	975	-	1017	76.50	1.822	-	76.15	1.743	1.748	1.814
orb02	10 x 10	812	815	865	56.66	1.681	869	84.52	1.465	1.464	1.521
la21	15 x 10	1033	-	1033	72.99	0.751	-	72.93	0.743	0.759	0.810
la22	15 x 10	913	-	924	57.81	3.475	925	69.53	3.448	3.511	3.685
la26	20 x 10	1218	-	1218	97.78	0.806	-	-	0.724	0.760	0.880
la27	20 x 10	1235	-	1235	91.06	1.055	-	-	0.875	0.915	1.040
la36	15 x 15	1233	-	1267	87.77	5.761	-	87.66	5.303	5.414	5.686
la37	15 x 15	1397	-	1397	66.23	2.731	-	66.21	2.471	2.527	2.658
ta01	15 x 15	1190	1193	1223	73.62	9.840	1224	71.38	9.034	9.174	9.536
ta02	15 x 15	1167	-	1210	84.38	7.585	-	75.76	7.012	7.137	7.470
ta11	20 x 15	1269	-	1295	73.74	18.51	-	70.61	14.55	14.86	15.50
ta12	20 x 15	1314	-	1336	86.63	21.03	-	86.22	17.25	17.66	18.55
ta21	20 x 20	1508	-	1546	82.62	43.68	-	80.96	38.27	39.33	40.32
ta22	20 x 20	1441	-	1499	82.84	31.16	-	92.85	25.19	25.64	26.66
yn1	20 x 20	784	-	816	86.66	29.09	-	86.33	26.39	26.94	28.04
yn2	20 x 20	819	835	841	84.91	24.40	842	88.61	22.65	23.06	24.09

Table 1. Destructive Lower Bounds

The table shows that detectable precedences improved the filtering but not much. However there is another interesting point: detectable precedences speed up the propagation, compare T and T1 *e.g.* for ta21. It is because detectable precedences are able "steal" a lot of work from edge-finding and do it faster.

Quite surprisingly, new not-first/not-last algorithm is about the same fast as [8] for $n = 10$, for bigger n it begins to be be faster. Note that the most filtering is done by detectable precedences, therefore speed of a not-first/not-last algorithm has only minor influence to total time.

[2] Benchmarks were performed on Intel Pentium Centrino 1300MHz

References

[1] Philippe Baptiste and Claude Le Pape. Edge-finding constraint propagation algorithms for disjunctive and cumulative scheduling. In *Proceedings of the Fifteenth Workshop of the U.K. Planning Special Interest Group*, 1996.

[2] Peter Brucker. Complex scheduling problems, 1999. URL http://citeseer.nj.nec.com/brucker99complex.html.

[3] Jacques Carlier and Eric Pinson. Adjustements of head and tails for the job-shop problem. *European Journal of Operational Research*, 78:146–161, 1994.

[4] Yves Caseau and Francois Laburthe. Disjunctive scheduling with task intervals. In *Technical report, LIENS Technical Report 95-25*. Ecole Normale Suprieure Paris, Franc, 1995.

[5] W. Nuijten F. Foccaci, P. Laborie. Solving scheduling problems with setup times and alternative resources. In *Proceedings of the 4th International Conference on AI Planning and Scheduling, AIPS'00*, pages 92–101, 2000.

[6] Paul Martin and David B. Shmoys. A New Approach to Computing Optimal Schedules for the Job-Shop Scheduling Problem. In W. H. Cunningham, S. T. McCormick, and M. Queyranne, editors, *Proceedings of the 5th International Conference on Integer Programming and Combinatorial Optimization, IPCO'96*, pages 389–403, Vancouver, British Columbia, Canada, 1996.

[7] Caude Le Pape Philippe Baptiste and Wim Nuijten. *Constraint-Based Scheduling: Applying Constraint Programming to Scheduling Problems*. Kluwer Academic Publishers, 2001.

[8] Philippe Torres and Pierre Lopez. On not-first/not-last conditions in disjunctive scheduling. *European Journal of Operational Research*, 1999.

[9] Petr Vilím. Batch processing with sequence dependent setup times: New results. In *Proceedings of the 4th Workshop of Constraint Programming for Decision and Control, CPDC'02*, Gliwice, Poland, 2002.

[10] Armin Wolf. Pruning while sweeping over task intervals. In *Principles and Practice of Constraint Programming - CP 2003*, Kinsale, Ireland, 2003.