Unary Resource Constraint with Optional Activities

Petr Vilím[1], Roman Barták[1], Ondřej Čepek[1,2]

[1] Charles University
Faculty of Mathematics and Physics
Malostranské náměstí 2/25, Praha 1, Czech Republic
[2] Institute of Finance and Administration - VŠFS
vilim@kti.mff.cuni.cz
bartak@kti.mff.cuni.cz
ondrej.cepek@mff.cuni.cz

Abstract. Scheduling is one of the most successful application areas of constraint programming mainly thanks to special global constraints designed to model resource restrictions. Among these global constraints, edge-finding filtering algorithm for unary resources is one of the most popular techniques. In this paper we propose a new $O(n \log n)$ version of the edge-finding algorithm that uses a special data structure called Θ-Λ-tree. This data structure is especially designed for "what-if" reasoning about a set of activities so we also propose to use it for handling so called optional activities, i.e. activities which may or may not appear on the resource. In particular, we propose new $O(n \log n)$ variants of filtering algorithms which are able to handle optional activities: overload checking, detectable precedences and not-first/not-last.

1 Introduction

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

Each activity $i \in T$ has the 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 techniques to solve this problem is *constraint programming*.

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

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

Filtering algorithms considered in this paper are:

Edge-finding. Paper [5] presents $O(n \log n)$ version, another two $O(n^2)$ versions of edge-finding can be found in [7, 8].

Not-first/not-last. $O(n \log n)$ version of the algorithm can be found in [10], two older $O(n^2)$ versions are in [2, 9].

Detectable precedences. This $O(n \log n)$ algorithm was introduced in [10].

All these filtering algorithms can be used together to join their filtering powers.

This paper introduces new version of the edge-finding algorithm with time complexity $O(n \log n)$. Experimental results shows that this new edge-finding algorithm is faster than the quadratic algorithms [7, 8] even for $n = 15$. Another asset of the algorithm is the introduction of the Θ-Λ-tree – a data structure which can be used to extend filtering algorithms to handle optional activities.

2 Edge-Finding using Θ-Λ-tree

2.1 Basic Notation

Let us establish the basic notation concerning a subset of activities. Let T be a set of all activities on the resource and let $\Theta \subseteq T$ be an arbitrary non-empty 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 defined as:

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

Often we need to estimate an earliest completion time of a set Θ:

$$\mathrm{ECT}_\Theta = \max \{\mathrm{est}_{\Theta'} + \mathrm{p}_{\Theta'},\ \Theta' \subseteq \Theta\} \tag{1}$$

To extend the definitions also for $\Theta = \emptyset$ let $\mathrm{est}_\emptyset = -\infty$, $\mathrm{lct}_\emptyset = \infty$, $\mathrm{p}_\emptyset = 0$ and $\mathrm{ECT}_\emptyset = -\infty$.

2.2 Edge-Finding Rules

Edge-finding is probably the most often used filtering algorithm for a unary resource constraint. Let us recall classical edge-finding rules [2]. Consider a set

$\Omega \subseteq T$ and an activity $i \notin \Omega$. If the following condition holds, then the activity i has to be scheduled after all activities from Ω:

$$\forall \Omega \subset T,\ \forall i \in (T \setminus \Omega):$$
$$\mathrm{est}_{\Omega\cup\{i\}} + \mathrm{p}_{\Omega\cup\{i\}} = \min\{\mathrm{est}_\Omega,\ \mathrm{est}_i\} + \mathrm{p}_\Omega + \mathrm{p}_i > \mathrm{lct}_\Omega \quad \Rightarrow \quad \Omega \ll i \qquad (2)$$

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

$$\Omega \ll i \quad \Rightarrow \quad \mathrm{est}_i := \max\{\mathrm{est}_i,\ \mathrm{ECT}_\Omega\} \qquad (3)$$

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 [7, 8], [5] presents a $O(n \log n)$ algorithm.

Proposition 1. *Let $\Theta(j) = \{k,\ k \in T\ \&\ \mathrm{lct}_k \leq \mathrm{lct}_j\}$. The rules (2), (3) are not stronger than the following rule:*

$$\forall j \in T,\ \forall i \in T \setminus \Theta(j):$$
$$\mathrm{ECT}_{\Theta(j)\cup\{i\}} > \mathrm{lct}_j \quad \Rightarrow \quad \Theta(j) \ll i \quad \Rightarrow \quad \mathrm{est}_i := \max\left\{est_i,\ \mathrm{ECT}_{\Theta(j)}\right\} \qquad (4)$$

Actually, the rules (2) and (3) are equivalent with the rule (4). However, the proof of their equivalence (the reverse implication) is rather technical and therefore it is not included in the main body of this paper. An interested reader can find this proof in the appendix of this paper.

Proof. Let us consider a set $\Omega \subseteq T$ and an activity $i \in T \setminus \Omega$. Let j be one of the activities from Ω for which $\mathrm{lct}_j = \mathrm{lct}_\Omega$. Thanks to this definition of j we have $\Omega \subseteq \Theta(j)$ and so (recall the definition (1) of ECT):

$$\mathrm{est}_{\Omega\cup\{i\}} + \mathrm{p}_{\Omega\cup\{i\}} = \min\{\mathrm{est}_\Omega,\ \mathrm{est}_i\} + \mathrm{p}_\Omega + \mathrm{p}_i \leq \mathrm{ECT}_{\Theta(j)\cup\{i\}}$$
$$\mathrm{ECT}_\Omega \leq \mathrm{ECT}_{\Theta(j)}$$

Thus: when the original rule (2) holds for Ω and i, then the new rule (4) holds for $\Theta(j)$ and i too, and the change of est_i is at least the same as the change by the rule (3). □

Property 1. The rule (4) has a very useful property. Let us consider an activity i and two different activities j_1 and j_2 for which the rule (4) holds. Moreover let $\mathrm{lct}_{j_1} \leq \mathrm{lct}_{j_2}$. Then $\Theta(j_1) \subseteq \Theta(j_2)$ and so $\mathrm{ECT}_{\Theta(j_1)} \leq \mathrm{ECT}_{\Theta(j_2)}$, therefore j_2 yields better propagation then j_1. Thus for a given activity i it is sufficient to look for the activity j for which (4) holds and lct_j is maximum.

2.3 Θ-Λ-tree

A Θ-Λ-tree is an extension of a Θ-tree introduced in [10]. Θ-tree is a data structure designed to represent a set of activities $\Theta \subseteq T$ and to quickly compute ECT_Θ. Θ-tree was already successfully used to speed up two filtering algorithms for unary resource: not-first/not-last and detectable precedences [10].

In a Θ-tree, activities are represented as nodes in a balanced binary search tree with respect to est. In the following we will not make a difference between an activity and the tree node which represents that activity. Besides an activity itself, each node k of a Θ-tree holds the following two values:

$$\Sigma\mathrm{P}_k = \sum_{j \in \mathrm{Subtree}(k)} \mathrm{p}_j$$

$$\mathrm{ECT}_k = \mathrm{ECT}_{\mathrm{Subtree}(k)} = \max\{\mathrm{est}_{\Theta'} + \mathrm{p}_{\Theta'},\ \Theta' \subseteq \mathrm{Subtree}(k)\}$$

where $\mathrm{Subtree}(k)$ is a set of all activities in a subtree rooted at node k (including activity k itself). The values $\Sigma\mathrm{P}_k$ and ECT_k can be computed recursively from the direct descendants of the node (for more details see [10]):

$$\Sigma\mathrm{P}_k = \Sigma\mathrm{P}_{\mathrm{left}(k)} + \mathrm{p}_k + \Sigma\mathrm{P}_{\mathrm{right}(k)} \tag{5}$$

$$\begin{aligned}\mathrm{ECT}_k = \max\big\{ & \mathrm{ECT}_{\mathrm{right}(k)}, \\ & \mathrm{est}_k + \mathrm{p}_k + \Sigma\mathrm{P}_{\mathrm{right}(k)}, \\ & \mathrm{ECT}_{\mathrm{left}(k)} + \mathrm{p}_k + \Sigma\mathrm{P}_{\mathrm{right}(k)} \big\}\end{aligned} \tag{6}$$

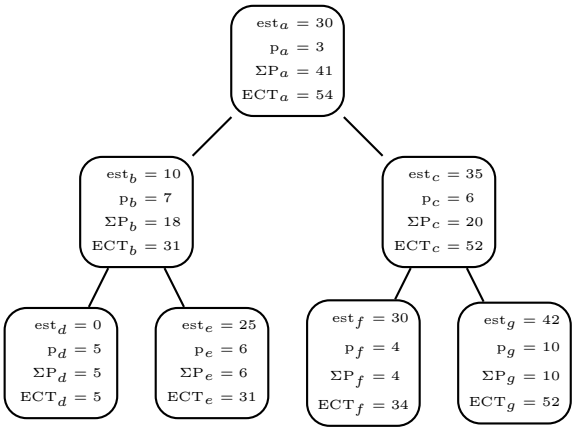

Fig. 1. An example of a Θ-tree for $\Theta = \{a, b, c, d, e, f, g\}$.

Let us now consider alternative edge-finding rule (4). We choose an arbitrary activity j and now we want to check the rule (4) for each applicable activity i,

i.e. we would like to find all activities i for which the following condition holds:

$$\mathrm{ECT}_{\Theta(j)\cup\{i\}} > \mathrm{lct}_j$$

Unfortunately, such an algorithm would be too slow: before the check can be performed, each particular activity i must be added into the Θ-tree, and after the check the activity i have to be removed back from the Θ-tree.

The idea how to surpass this problem is to extend the Θ-tree structure the following way: all applicable activities i will be also included in the tree, but as a *gray* nodes. A gray node represents an activity i which is not really in the set Θ. However, we are curious what would happen with ECT_Θ if we are allowed to include **one** of the gray activities into the set Θ. More exactly: let $\Lambda \subseteq T$ be a set of all gray activities, $\Lambda \cap \Theta = \emptyset$. The purpose of the Θ-Λ-tree is to compute the following value:

$$\overline{\mathrm{ECT}}(\Theta, \Lambda) = \max\left\{\{\mathrm{ECT}_\Theta\} \cup \left\{\mathrm{ECT}_{\Theta\cup\{i\}},\ i \in \Lambda\right\}\right\}$$

The meaning of the values ECT and ΣP in the new tree remains the same, however only regular (*white*) nodes are taken into account. Moreover, in order to compute $\overline{\mathrm{ECT}}(\Theta, \Lambda)$ quickly, we add the following two values into each node of the tree:

$$\begin{aligned}
\overline{\Sigma\mathrm{P}}_k &= \max\left\{\mathrm{p}_{\Theta'},\ \Theta' \subseteq \mathrm{Subtree}(k)\ \&\ |\Theta' \cap \Lambda| \leq 1\right\} \\
&= \max\left\{\{0\} \cup \{\mathrm{p}_i,\ i \in \mathrm{Subtree}(k) \cap \Lambda\}\right\} + \sum_{i \in \mathrm{Subtree}(k)\cap\Theta} \mathrm{p}_i \\
\overline{\mathrm{ECT}}_k &= \overline{\mathrm{ECT}}_{\mathrm{Subtree}(k)} = \max\left\{\mathrm{est}_{\Theta'} + \mathrm{p}_{\Theta'},\ \Theta' \subseteq \mathrm{Subtree}(k)\ \&\ |\Theta' \cap \Lambda| \leq 1\right\}
\end{aligned}$$

$\overline{\Sigma\mathrm{P}}$ is maximum sum of processing activities in a subtree if one of gray activities can be used. Similarly $\overline{\mathrm{ECT}}$ is an earliest completion time of a subtree with at most one gray activity included.

An idea how to compute values $\overline{\Sigma\mathrm{P}}_k$ and $\overline{\mathrm{ECT}}_k$ in node k follows. A gray activity can be used only once. Therefore when computing $\overline{\Sigma\mathrm{P}}_k$ and $\overline{\mathrm{ECT}}_k$, a gray activity can be used only in one of the following places: in the left subtree of k, by the activity k itself (if it is gray), or in the right subtree of k. Note that the gray activity used for $\overline{\Sigma\mathrm{P}}_k$ can be different from the gray activity used for $\overline{\mathrm{ECT}}_k$. The formulae (5) and (6) can be modified to handle gray nodes.

We distinguish two cases: node k is gray or node k is white. When k is white then:

$$\begin{aligned}
\overline{\Sigma\mathrm{P}}_k = \max\Big\{ & \overline{\Sigma\mathrm{P}}_{\mathrm{left}(k)} + \mathrm{p}_k + \Sigma\mathrm{P}_{\mathrm{right}(k)}, \\
& \Sigma\mathrm{P}_{\mathrm{left}(k)} + \mathrm{p}_k + \overline{\Sigma\mathrm{P}}_{\mathrm{right}(k)} \Big\} \\
\overline{\mathrm{ECT}}_k = \max\Big\{ & \overline{\mathrm{ECT}}_{\mathrm{right}(k)}, && \text{(a)} \\
& \mathrm{est}_k + \mathrm{p}_k + \overline{\Sigma\mathrm{P}}_{\mathrm{right}(k)}, && \text{(b)} \\
& \mathrm{ECT}_{\mathrm{left}(k)} + \mathrm{p}_k + \overline{\Sigma\mathrm{P}}_{\mathrm{right}(k)}, && \text{(c)} \\
& \overline{\mathrm{ECT}}_{\mathrm{left}(k)} + \mathrm{p}_k + \Sigma\mathrm{P}_{\mathrm{right}(k)} \Big\} && \text{(c)}
\end{aligned}$$

Line (a) considers all sets Θ' such that $\Theta' \subseteq \text{Subtree}(\text{right}(k))$ (see the definition (1) of ECT on page 2). Line (b) considers all sets Θ' such that $\Theta' \subseteq \text{Subtree}(\text{right}(k)) \cup \{k\}$ and $k \in \Theta'$. Finally lines (c) consider sets Θ' such that $\Theta' \cap \text{Subtree}(\text{left}(k)) \neq \emptyset$.

When k is gray then (the meaning of the labels (a), (b) and (c) remains the same):

$$\begin{aligned}
\overline{\Sigma\mathrm{P}}_k = \max\{\ & \overline{\Sigma\mathrm{P}}_{\mathrm{left}(k)} + \Sigma\mathrm{P}_{\mathrm{right}(k)}, \\
& \Sigma\mathrm{P}_{\mathrm{left}(k)} + \mathrm{p}_k + \Sigma\mathrm{P}_{\mathrm{right}(k)}, \\
& \Sigma\mathrm{P}_{\mathrm{left}(k)} + \overline{\Sigma\mathrm{P}}_{\mathrm{right}(k)}\ \} \\
\overline{\mathrm{ECT}}_k = \max\{\ & \overline{\mathrm{ECT}}_{\mathrm{right}(k)}, && \text{(a)} \\
& \mathrm{est}_k + \mathrm{p}_k + \Sigma\mathrm{P}_{\mathrm{right}(k)}, && \text{(b)} \\
& \overline{\mathrm{ECT}}_{\mathrm{left}(k)} + \Sigma\mathrm{P}_{\mathrm{right}(k)}, && \text{(c)} \\
& \mathrm{ECT}_{\mathrm{left}(k)} + \mathrm{p}_k + \Sigma\mathrm{P}_{\mathrm{right}(k)}, && \text{(c)} \\
& \mathrm{ECT}_{\mathrm{left}(k)} + \overline{\Sigma\mathrm{P}}_{\mathrm{right}(k)}\ \} && \text{(c)}
\end{aligned}$$

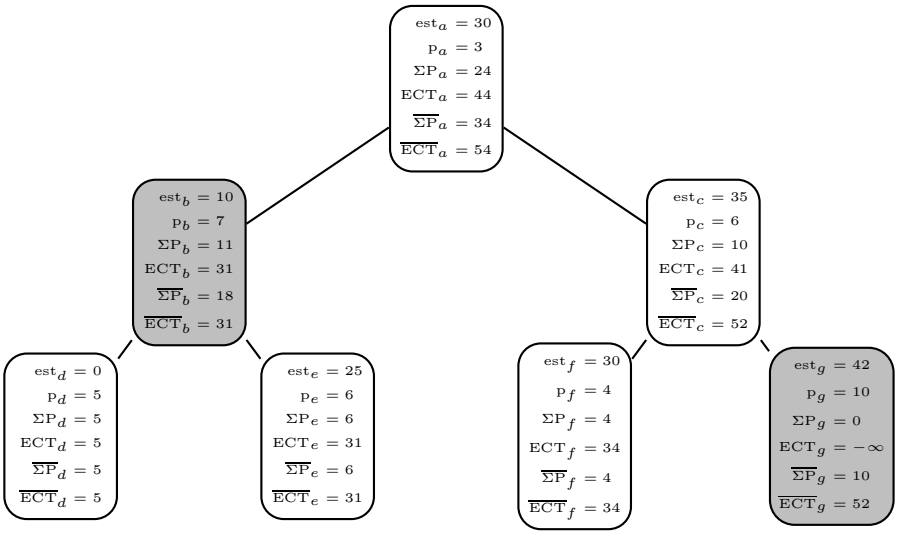

Fig. 2. An example of a Θ-Λ-tree for $\Theta = \{a, c, d, e, f\}$ and $\Lambda = \{b, g\}$.

Thanks to these recursive formulae, $\overline{\mathrm{ECT}}$ and $\overline{\Sigma\mathrm{P}}$ can be computed within usual operations with balanced binary trees without changing their time complexities. Note that together with $\overline{\mathrm{ECT}}$ we can compute for each node k the gray activity *responsible* for $\overline{\mathrm{ECT}}_k$. We need to know such responsible gray activity in the following algorithms.

Table 1 shows time complexities of some operations on Θ-Λ-tree.

Operation	Time Complexity
$(\Theta,\ \Lambda) := (\emptyset,\ \emptyset)$	$O(1)$
$(\Theta,\ \Lambda) := (T,\ \emptyset)$	$O(n \log n)$
$(\Theta,\ \Lambda) := (\Theta \setminus \{i\},\ \Lambda \cup \{i\})$	$O(\log n)$
$\Theta := \Theta \cup \{i\}$	$O(\log n)$
$\Lambda := \Lambda \cup \{i\}$	$O(\log n)$
$\Lambda := \Lambda \setminus \{i\}$	$O(\log n)$
$\overline{\mathrm{ECT}}(\Theta, \Lambda)$	$O(1)$
ECT_Θ	$O(1)$

Table 1. Time complexities of operations on Θ-Λ-tree.

2.4 Edge-Finding Algorithm

The algorithm starts with $\Theta = T$ and $\Lambda = \emptyset$. Activities are sequentially (in descending order by lct_j) moved from the set Θ into the set Λ, i.e. white nodes are discolored to gray. As soon as $\overline{\mathrm{ECT}}(\Theta,\ \Lambda) > \mathrm{lct}_\Theta$, a responsible gray activity i is updated. Thanks to the property 1 (page 3) the activity i cannot be updated better, therefore we can remove the activity i from the tree (i.e. remove it from the set Λ).

```
1  for i ∈ T do
2     est'_i := est_i ;
3  (Θ, Λ) := (T, ∅) ;
4  Q := queue of all activities j ∈ T in descending order of lct_j ;
5  j := Q.first ;
6  repeat
7     (Θ, Λ) := (Θ \ {j}, Λ ∪ {j}) ;
8     Q.dequeue ;
9     j := Q.first ;
10    if ECT_Θ > lct_j then
11       fail ; { Resource is overloaded }
12    while ECT̄(Θ, Λ) > lct_j do begin
13       i := gray activity responsible for ECT̄(Θ, Λ) ;
14       est'_i := max{est_i, ECT_Θ} ;
15       Λ := Λ \ {i} ;
16    end ;
17 until Q.size = 0 ;
18 for i ∈ T do
19    est_i := est'_i ;
```

Note that at line 13 there have to be some gray activity responsible for $\overline{\mathrm{ECT}}(\Theta,\ \Lambda)$ because otherwise we would end up by fail on line 11.

During the entire run of the algorithm, maximum number of iterations of the inner while loop is n, because each iteration removes an activity from the set Λ. Similarly, number of iterations of the repeat loop is n, because each time

an activity is removed from the queue Q. According to table 1 time complexity of each single line within the loops is $O(\log n)$ maximum. Therefore the time complexity of the whole algorithm is $O(n \log n)$.

Note that at the beginning $\Theta = T$ and $\Lambda = \emptyset$, hence there are no gray activities and therefore $\overline{\mathrm{ECT}}_k = \mathrm{ECT}_k$ and $\overline{\Sigma\mathrm{P}}_k = \Sigma\mathrm{P}_k$ for each node k. Hence we can save some time by building the initial Θ-Λ-tree as a "normal" Θ-tree.

3 Optional Activities

Nowadays, many practical scheduling problems have to deal with alternatives – activities which can choose their resource, or activities which exist only if a particular alternative of processing is chosen. From the resource point of view, it is not yet decided whether such activities will be processed or not. Therefore we will call such activities *optional*. For an optional activity, we would like to speculate what would happen if the activity actually would be processed by the resource.

Traditionally, resource constraints are not designed to handle optional activities properly. However, several different modifications are used to model them:

Dummy activities. It is basically a workaround for constraint solvers which do not allow to add more activities on the resource during problem solving (i.e. resource constraint is not dynamic [3]). Processing time of activities is turned from constants to domain variables. Several "dummy" activities with processing time domain $\langle 0,\ \infty)$ are added on the resource as a reserve for possible activity addition. Filtering algorithms work as usual, but they use minimum of possible processing time instead of original constant processing time. Note that dummy activities have no influence on other activities on the resource, because their processing time can be zero. Once an alternative is chosen, a dummy activity is turned into regular activity (i.e. minimum of processing time is no longer zero). In this approach, an impossibility of an alternative cannot be found before that alternative is actually tried.

Filtering of options. The idea is to run a filtering algorithm several times, each time with one of the optional activities added on the resource. When a fail is found, then the optional activity is rejected. Otherwise time bounds of the optional activity can be adjusted. [4] introduces so called PEX-edge-finding with time complexity $O(n^3)$. This is a pretty strong propagation, however rather time consuming.

Modified filtering algorithms. Regular and optional activities are treated differently: optional activities do not influence any other activity on the resource, however regular activities influence other regular activities and also optional activities [6]. Most of the filtering algorithms can be modified this way without changing their time complexities. However, this approach is a little bit weaker than the previous one, because previous approach also checked whether the addition of a optional activity would not cause an immediate fail.

Cumulative resources. If we have a set of similar alternative machines, this set can be modeled as a cumulative resource. This additional (redundant) constraint can improve the propagation before activities are distributed between the machines. There is also a special filtering algorithm [11] designed to handle this type of alternatives.

To handle optional activities we extend each activity i by a variable called existence_i with the domain {true, false}. When $\text{existence}_i = \text{true}$ then i is a regular activity, when $\text{existence}_i \in \{\text{true, false}\}$ then i is an optional activity. Finally when existence = false we simply exclude this activity from all our considerations.

To make the notation concerning optional activities easy, let R be the set of all regular activities and O the set of all optional activities.

For optional activities, we would like to consider the following issues:

1. If an optional activity should be processed by the resource (i.e. if an optional activity is changed to a regular activity), would the resource be overloaded? The resource is overloaded if there is such a set $\Omega \subseteq R$ that:

$$\text{lct}_\Omega - \text{est}_\Omega < \text{p}_\Omega$$

   Certainly, if a resource is overloaded then the problem has no solution. Hence if an addition of a optional activity i results in overloading then we can conclude that $\text{existence}_i = \text{false}$.
2. If the addition of an optional activity i does not result in overloading, what is the earliest possible start time and the latest possible completion time of the activity i with respect to regular activities on the resource? We would like to apply usual filtering algorithms for the activity i, however the activity i cannot cause change of any regular activity.
3. If we add an optional activity i, will the first run of a filtering algorithm result in a fail? For example algorithm detectable precedences can increase est_k of some activity k so much that $\text{est}_k + \text{p}_k > \text{lct}_k$. In that case we can also propagate $\text{existence}_i = \text{false}$.

We will consider the item 1 in the next section "Overload Checking with Optional Activities". Items 2 and 3 are discussed in section "Filtering with Optional Activities".

4 Overload Checking with Optional Activities

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

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

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

a set Θ:

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

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 the lct_Θ. For each lct_Θ we check whether $\mathrm{ECT}_\Theta > \mathrm{lct}_\Theta$ or not.

But what about optional activities? Let Λ be the following set:

$$\Lambda = \{j,\ j \in O\ \&\ \mathrm{lct}_j \leq \mathrm{lct}_i\}$$

An optional activity can cause overloading if and only if $\overline{\mathrm{ECT}}(\Theta, \Lambda) > \mathrm{lct}_i$. The following algorithm is an extension of the algorithm presented in [10]. Optional activities are represented by gray nodes in the Θ-Λ-tree.

The following algorithm deletes all optional activities k such that an addition of each activity k alone causes an overload. Of course, a combination of several optional activities that are not deleted may still cause an overload!

$(\Theta,\ \Lambda)\ :=\ (\emptyset,\ \emptyset)$;
for $i \in T$ in ascending order of lct_i **do begin**
  if i is a regular activity **then begin**
    $\Theta\ :=\ \Theta \cup \{i\}$;
    if $\mathrm{ECT}_\Theta > \mathrm{lct}_i$ **then**
      fail; {No solution exists}
  end else
    $\Lambda\ :=\ \Lambda \cup \{i\}$;
  while $\overline{\mathrm{ECT}}(\Theta,\ \Lambda) > \mathrm{lct}_i$ **do begin**
    $k\ :=$ optional activity responsible for $\overline{\mathrm{ECT}}(\Theta,\ \Lambda)$;
    $\mathrm{existence}_k\ :=$ false;
    $\Lambda\ :=\ \Lambda \setminus \{k\}$;
  end;
end;

The complexity of the algorithm is again $O(n \log n)$. The inner while loop is repeated n times maximum because each time an activity is removed from the set Λ. Outer for loop has also n iterations, time complexity of each single line is $O(\log n)$ maximum (see the table 1).

5 Filtering with Optional Activities

The following section is an example how to extend a certain class of filtering algorithms to handle optional activities. The idea is simple: if the original algorithm uses Θ-tree, we will use Θ-Λ-tree instead. The difference is that we represent optional activities by gray nodes. For propagation we still use ECT_Θ, however we can check $\overline{\mathrm{ECT}}(\Theta, \Lambda)$ also. If propagation using $\overline{\mathrm{ECT}}(\Theta, \Lambda)$ would result in an immediate fail we can exclude the optional activity responsible for that.

Let us demonstrate this idea on the detectable precedences algorithm:

```
(Θ, Λ) := ∅;
Q := queue of all activities j ∈ T in ascending order of lct_j − p_j;
for i ∈ T in ascending order of est_i + p_i do begin
  while est_i + p_i > lct_Q.first − p_Q.first do begin
    if i is a regular activity then
      Θ := Θ ∪ {Q.first};
    else
      Λ := Λ ∪ {Q.first};
    Q.dequeue;
  end;
  est'_i := max{est_i, ECT_{Θ\{i}}};
  if i is a regular activity then
    while ECT̄(Θ \ {i}, Λ) + p_i > lct_i then begin
      k := an optional activity responsible for ECT̄(Θ \ {i}, Λ);
      Λ := Λ \ {k};
      existence_k := false;
    end;
end;
for i ∈ T do
  est_i := est'_i;
```

The complexity of the algorithm remains the same: $O(n \log n)$.

The same idea can be used to extend the not-first/not-last algorithm presented in [10]. However, extending the edge-finding algorithm is not so easy: edge-finding algorithm already uses Θ-Λ-tree. We will consider this in our future work.

6 Experimental Results

We tested the new edge-finding algorithm on several benchmark jobshop problems taken from OR library [1]. The benchmark problem is to compute a destructive lower bound using the shaving technique. Destructive lower bound is the minimal makespan for which propagation is not able to find conflict without backtracking. Because destructive lower bound is computed too quickly, we use also shaving as suggested in [7]. Shaving is similar to the proof by a 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 decrease lct_i or increase est_i. Binary search is used to find the best shave. To limit CPU time, shaving was used for each activity only once.

Table 2 shows the results. We measured the CPU[3] time needed to prove the lower bound, i.e. the propagation is done twice: with the upper bound LB and LB-1. Times T1–T3 show running time for different implementations of the edge-finding algorithm: T1 is the new algorithm, T2 is the algorithm [7] and

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

T3 is the algorithm [8]. As can be seen, the new algorithm is quite competitive for $n = 10$ and $n = 15$, for $n \geq 20$ it is faster than the other two edge-finding algorithms.

Prob.	**Size**	**LB**	**T1**	**T2**	**T3**
abz5	10 x 10	1196	1.430	1.421	1.466
abz6	10 x 10	941	1.773	1.762	1.815
orb01	10 x 10	1017	1.773	1.783	1.841
orb02	10 x 10	869	1.491	1.486	1.529
ft10	10 x 10	911	1.616	1.618	1.669
la21	15 x 10	1033	0.752	0.784	0.815
la22	15 x 10	925	3.486	3.597	3.763
la36	15 x 15	1267	5.376	5.520	5.768
la37	15 x 15	1397	2.498	2.572	2.667
ta01	15 x 15	1224	9.113	9.304	9.652
ta02	15 x 15	1210	7.097	7.264	7.586
la26	20 x 10	1218	0.749	0.838	0.899
la27	20 x 10	1235	0.908	0.994	1.054
la29	20 x 10	1119	3.357	3.609	3.816
abz7	20 x 15	651	3.283	3.446	3.579
abz8	20 x 15	621	12.00	12.54	13.14
ta11	20 x 15	1295	14.72	15.31	16.03
ta12	20 x 15	1336	17.54	18.30	19.26
ta21	20 x 20	1546	38.43	39.79	41.90
ta22	20 x 20	1501	25.47	26.25	27.37
yn1	20 x 20	816	26.79	27.58	28.91
yn2	20 x 20	842	22.86	23.59	24.69
ta31	30 x 15	1764	4.788	5.485	5.936
ta32	30 x 15	1774	6.515	7.390	7.946
swv11	50 x 10	2983	15.70	19.70	21.62
swv12	50 x 10	2972	19.21	23.43	25.23
ta51	50 x 15	2760	11.68	14.58	15.88
ta52	50 x 15	2756	12.07	15.04	16.32
ta71	100 x 20	5464	131.6	173.6	189.3
ta72	100 x 20	5181	132.0	174.8	190.8

Table 2. Destructive Lower Bounds

Optional activities were tested on modified 10x10 jobshop instances. In each job, activities on 5th and 6th place were taken as alternatives. Therefore in each problem there are 20 optional activities and 80 regular activities. Table 3 shows the results. Column LB is the destructive lower bound computed without shaving, column Opt is the optimal makespan. Column CH is the number of choicepoints needed to find the optimal solution and prove the optimality (i.e. optimal makespan used as the initial upper bound). Finally the column T is the CPU time in seconds.

As can be seen in the table, propagation is strong, all of the problems were solved surprisingly quickly. However more test should be made, especially on real life problem instances.

Prob.	Size	LB	Opt	CH	T
abz5-alt	10 x 10	1031	1093	283	0.336
abz6-alt	10 x 10	791	822	17	0.026
orb01-alt	10 x 10	894	947	9784	12.776
orb02-alt	10 x 10	708	747	284	0.328
ft10-alt	10 x 10	780	839	4814	6.298
la16-alt	10 x 10	838	842	27	0.022
la17-alt	10 x 10	673	676	24	0.021
la18-alt	10 x 10	743	750	179	0.200
la19-alt	10 x 10	686	731	84	0.103
la20-alt	10 x 10	809	809	14	0.014

Table 3. Alternative activities

Acknowledgements: Authors would like to thank all the anonymous referees for their helpful comments and advises. This work has been supported by the Czech Science Foundation under the contract no. 201/04/1102.

References

[1] OR library. URL http://mscmga.ms.ic.ac.uk/info.html.

[2] 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.

[3] Roman Barták. Dynamic global constraints in backtracking based environments. *Annals of Operations Research*, 118:101–118, 2003.

[4] J. Christopher Beck and Mark S. Fox. Scheduling alternative activities. In *AAAI/IAAI*, pages 680–687, 1999.

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

[6] F. Focacci, P. Laborie, and W. Nuijten. Solving scheduling problems with setup times and alternative resources. In *Proceedings of the 5th International Conference on Artificial Intelligence Planning and Scheduling*, 2000.

[7] 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.

[8] Claude Le Pape Philippe Baptiste and Wim Nuijten. *Constraint-Based Scheduling: Applying Constraint Programming to Scheduling Problems.* Kluwer Academic Publishers, 2001.
[9] Philippe Torres and Pierre Lopez. On not-first/not-last conditions in disjunctive scheduling. *European Journal of Operational Research*, 1999.
[10] Petr Vilím. $O(n \log n)$ filtering algorithms for unary resource constraint. In *Proceedings of CP-AI-OR 2004*. Springer-Verlag, 2004.
[11] Armin Wolf and Hans Schlenker. Realizing the alternative resources constraint problem with single resource constraints. In *To appear in proceedings of the INAP workshop 2004*, 2004.

Appendix

7 Equivalence of the Edge-Finding Rules

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:

$$\mathrm{lct}_\Omega - \mathrm{est}_\Omega < \mathrm{p}_\Omega \quad \Rightarrow \quad \text{fail} \tag{7}$$

Note that it is useless to continue filtering when a fail was fired. Therefore in the following we will assume that the resource is not overloaded.

Proposition 2. *The rule (4) is not stronger than the original rules (2) and (3).*

Proof. Let us consider a pair of activities i, j for which the new rule (4) holds. We define a set Ω' as a subset of $\Theta(j) \cup \{i\}$ for which:

$$\mathrm{ECT}_{\Theta(j)\cup\{i\}} = \mathrm{est}_{\Omega'} + \mathrm{p}_{\Omega'} \tag{8}$$

Note that thanks to the definition (1) of ECT such a set Ω' must exist.

If $i \notin \Omega'$ then $\Omega' \subseteq \Theta(j)$, therefore

$$\mathrm{est}_{\Omega'} + \mathrm{p}_{\Omega'} \overset{(8)}{=} \mathrm{ECT}_{\Theta(j)\cup\{i\}} \overset{(4)}{>} \mathrm{lct}_j \geq \mathrm{lct}_{\Omega'}$$

So the resource is overloaded (see the overload rule (7)) and fail should have already been fired.

Thus $i \in \Omega'$. Let us define $\Omega = \Omega' \setminus \{i\}$. We will assume that $\Omega \neq \emptyset$, because otherwise $\mathrm{est}_i \geq \mathrm{ECT}_{\Theta(j)}$ and rule (4) changes nothing. For this set Ω we have:

$$\min\{\mathrm{est}_\Omega,\ \mathrm{est}_i\} + \mathrm{p}_\Omega + \mathrm{p}_i = \mathrm{est}_{\Omega'} + \mathrm{p}_{\Omega'} \overset{(8)}{=} \mathrm{ECT}_{\Theta(j)\cup\{i\}} \overset{(4)}{>} \mathrm{lct}_j \geq \mathrm{lct}_\Omega$$

Hence the rule (2) holds for the set Ω. To complete the proof we have to show that both rules (3) and (4) adjust est_i equivalently, i.e. $\mathrm{ECT}_\Omega = \mathrm{ECT}_{\Theta(j)}$. We

already know that $\mathrm{ECT}_{\Omega} \leq \mathrm{ECT}_{\Theta(j)}$ because $\Omega \subseteq \Theta(j)$. Suppose now for a contradiction that

$$\mathrm{ECT}_{\Omega} < \mathrm{ECT}_{\Theta(j)} \tag{9}$$

Let Φ be a set $\Phi \subseteq \Theta(j)$ such that:

$$\mathrm{ECT}_{\Theta(j)} = \mathrm{est}_{\Phi} + \mathrm{p}_{\Phi} \tag{10}$$

Therefore:

$$\mathrm{est}_{\Omega} + \mathrm{p}_{\Omega} \leq \mathrm{ECT}_{\Omega} \overset{(9)}{<} \mathrm{ECT}_{\Theta(j)} \overset{(10)}{=} \mathrm{est}_{\Phi} + \mathrm{p}_{\Phi} \tag{11}$$

Because the set $\Omega' = \Omega \cup \{i\}$ defines the value of $\mathrm{ECT}_{\Theta(j)\cup\{i\}}$ (i.e. $\mathrm{est}_{\Omega'} + \mathrm{p}_{\Omega'} = \mathrm{ECT}_{\Theta(j)\cup\{i\}}$), it has the following property (see the definition (1) of ECT):

$$\forall k \in \Theta(j) \cup \{i\} : \quad \mathrm{est}_k \geq \mathrm{est}_{\Omega'} \;\Rightarrow\; k \in \Omega'$$

And because $\Omega = \Omega' \setminus \{i\}$:

$$\forall k \in \Theta(j) : \quad \mathrm{est}_k \geq \mathrm{est}_{\Omega'} \;\Rightarrow\; k \in \Omega \tag{12}$$

Similarly, the set Φ defines the value of $\mathrm{ECT}_{\Theta(j)}$:

$$\forall k \in \Theta(j) : \quad \mathrm{est}_k \geq \mathrm{est}_{\Phi} \;\Rightarrow\; k \in \Phi \tag{13}$$

Combining properties (12) and (13) together we have that either $\Omega \subseteq \Phi$ (if $\mathrm{est}_{\Omega'} \geq \mathrm{est}_{\Phi}$) or $\Phi \subseteq \Omega$ (if $\mathrm{est}_{\Omega'} \leq \mathrm{est}_{\Phi}$). However, $\Phi \subseteq \Omega$ is not possible, because in this case $\mathrm{est}_{\Phi} + \mathrm{p}_{\Phi} \leq \mathrm{ECT}_{\Omega}$ what contradicts the inequality (11). The result is that $\Omega \subsetneq \Phi$, and so $\mathrm{p}_{\Omega} < \mathrm{p}_{\Phi}$.

Now we are ready to prove the contradiction:

$$\begin{aligned}
\mathrm{ECT}_{\Theta(j)\cup\{i\}} &\overset{(8)}{=} \mathrm{est}_{\Omega'} + \mathrm{p}_{\Omega'} \\
&= \min\{\mathrm{est}_{\Omega},\ \mathrm{est}_i\} + \mathrm{p}_{\Omega} + \mathrm{p}_i && \text{because } \Omega = \Omega' \setminus \{i\} \\
&= \min\{\mathrm{est}_{\Omega} + \mathrm{p}_{\Omega} + \mathrm{p}_i,\ \mathrm{est}_i + \mathrm{p}_{\Omega} + \mathrm{p}_i\} \\
&< \min\{\mathrm{est}_{\Phi} + \mathrm{p}_{\Phi} + \mathrm{p}_i,\ \mathrm{est}_i + \mathrm{p}_{\Phi} + \mathrm{p}_i\} && \text{by (11) and } \mathrm{p}_{\Omega} < \mathrm{p}_{\Phi} \\
&\leq \mathrm{ECT}_{\Theta(j)\cup\{i\}} && \text{because } \Phi \subseteq \Theta(j)
\end{aligned}$$

□