Extension of $O(n \log n)$ filtering algorithms for the unary resource constraint to optional activities

Petr Vilím (vilim@kti.mff.cuni.cz), Roman Barták
(bartak@kti.mff.cuni.cz) and Ondřej Čepek
(ondrej.cepek@mff.cuni.cz)
Charles University
Faculty of Mathematics and Physics
Malostranské náměstí 2/25, Praha 1, Czech Republic

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 and not-first/not-last are the most popular filtering algorithms for unary resources. In this paper we introduce new $O(n \log n)$ versions of these two filtering algorithms and one more $O(n \log n)$ filtering algorithm called detectable precedences. These algorithms use a special data structures Θ-tree and Θ-Λ-tree. These data structures are especially designed for "what-if" reasoning about a set of activities so we also propose to use them 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. This problem is long known to be computationally difficult [7][1]. 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

[1] Appears as problem [SS1] on page 236. It is NP-hard in the strong sense, so there is little hope even for a pseudo-polynomial algorithm. Therefore the use of CP is well justified here.

the 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.*

Due to the NP-hardness of the problem, 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 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 [8, 9].

Not-first/not-last. $O(n \log n)$ version of the algorithm can be found in [12], two older $O(n^2)$ versions are in [2, 10].

Detectable precedences. This $O(n \log n)$ algorithm was introduced in [12].

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

In this paper, we present not-first/not-last algorithm with time complexity $O(n \log n)$ and edge-finding algorithm with the same time complexity, which is quite simpler then the algorithm by Carlier and Pinson [5] and it is faster then quadratic algorithms which are widely used today. 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.

1.1. 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:

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

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. Θ-Tree

Θ-tree was first introduced in [12]. In this paper we present a slightly different version of this data structure. This new version is easier to implement and runs a little bit faster then the old one.

The purpose of a Θ-tree is to quickly recompute ECT_Θ when an activity is inserted or removed from the set Θ. Because the set represented by the tree will always be named Θ in this paper, we will call the tree Θ-tree.

A Θ-tree is a balanced binary tree. Activities from the set Θ are represented by leaf nodes[2]. Internal nodes of the tree are used to hold some precomputed values. In the following we do not make a difference between an activity and the leaf node representing that activity.

Let v be an arbitrary node of the Θ-tree (an internal node or a leaf). We define $\mathrm{Leaves}(v)$ to be the set of all activities represented in the leaves of the subtree rooted at the node v. Further let:

$$\Sigma\mathrm{P}_v = \sum_{j \in \mathrm{Leaves}(v)} \mathrm{p}_j$$
$$\mathrm{ECT}_v = \mathrm{ECT}_{\mathrm{Leaves}(v)} = \max\left\{\mathrm{est}_{\Theta'} + \mathrm{p}_{\Theta'},\ \Theta' \subseteq \mathrm{Leaves}(v)\right\}$$

Clearly, for an activity $i \in \Theta$ we have $\Sigma\mathrm{P}_i = \mathrm{p}_i$ and $\mathrm{ECT}_i = \mathrm{est}_i + \mathrm{p}_i$. Also, for root node r we have $\mathrm{ECT}_r = \mathrm{ECT}_\Theta$.

For internal node v the value $\Sigma\mathrm{P}_v$ can be easily computed from the direct descendants $\mathrm{left}(v)$ and $\mathrm{right}(v)$:

$$\Sigma\mathrm{P}_v = \Sigma\mathrm{P}_{\mathrm{left}(v)} + \Sigma\mathrm{P}_{\mathrm{right}(v)} \tag{2}$$

In order to compute also ECT_v quickly, the activities cannot be stored in the leaves randomly, but in the ascending order by est from left to right. I.e. for any two activities $i, j \in \Theta$, if $\mathrm{est}_i < \mathrm{est}_j$ then the activity i is stored on the left from the activity j. Thanks to this property the following inequality holds ($\mathrm{Left}(v)$ is a shortcut for $\mathrm{Leaves}(\mathrm{left}(v))$, similarly $\mathrm{Right}(v)$):

$$\forall i \in \mathrm{Left}(v), \forall j \in \mathrm{Right}(i): \quad \mathrm{est}_i \leq \mathrm{est}_j \tag{3}$$

[2] This is the main difference from [12]. The tree is deeper by one level, however a simpler computation of ECT and ΣP compensates that.

Proposition 1. For an internal node v, the value ECT_v can be computed by the following recursive formula:

$$\mathrm{ECT}_v = \max\left\{\mathrm{ECT}_{\mathrm{right}(v)},\ \mathrm{ECT}_{\mathrm{left}(v)} + \Sigma\mathrm{P}_{\mathrm{right}(v)}\right\} \tag{4}$$

Proof. From the definition (1), the value ECT_v is:

$$\mathrm{ECT}_v = \mathrm{ECT}_{\mathrm{Leaves}(v)} = \max\left\{\mathrm{est}_{\Theta'} + \mathrm{p}_{\Theta'},\ \Theta' \subseteq \mathrm{Leaves}(v)\right\}$$

With respect to the node v we will split the sets Θ' into the following two categories:

1. $\mathrm{Left}(v) \cap \Theta' = \emptyset$, i.e. $\Theta' \subseteq \mathrm{Right}(v)$. Clearly:

   $$\max\left\{\mathrm{est}_{\Theta'} + \mathrm{p}_{\Theta'},\ \Theta' \subseteq \mathrm{Right}(v)\right\} = \mathrm{ECT}_{\mathrm{Right}(v)} = \mathrm{ECT}_{\mathrm{right}(v)}$$

2. $\mathrm{Left}(v) \cap \Theta' \neq \emptyset$. Then $\mathrm{est}_{\Theta'} = \mathrm{est}_{\Theta' \cap \mathrm{Left}(v)}$ because of the property (3). Let S be the set of all possible Θ' considered now:

   $$S = \left\{\Theta',\ \Theta' \subseteq \Theta\ \&\ \Theta' \cap \mathrm{Left}(v) \neq \emptyset\right\}$$

   Thus:

   $$\begin{aligned}
   &\max\left\{\mathrm{est}_{\Theta'} + \mathrm{p}_{\Theta'},\ \Theta' \subseteq S\right\} = \\
   &= \max\left\{\mathrm{est}_{\Theta'\cap\mathrm{Left}(v)} + \mathrm{p}_{\Theta'\cap\mathrm{Left}(v)} + \mathrm{p}_{\Theta'\cap\mathrm{Right}(v)},\ \Theta' \subseteq S\right\} = \\
   &= \max\left\{\mathrm{est}_{\Theta'\cap\mathrm{Left}(v)} + \mathrm{p}_{\Theta'\cap\mathrm{Left}(v)},\ \Theta' \subseteq S\right\} + \mathrm{p}_{\mathrm{Right}(v)} = \\
   &= \mathrm{ECT}_{\mathrm{left}(v)} + \Sigma\mathrm{P}_{\mathrm{right}(v)}
   \end{aligned}$$

Therefore the formula (4) is correct. □

Figure 1 shows an example a Θ-tree.

Thanks to the recursive formulae (2) and (4), the values ECT_v and $\Sigma\mathrm{P}_v$ can be easily computed within usual operations with a balanced binary tree without changing their time complexities. Table I summarizes time complexities of different operations with a Θ-tree.

Notice that so far Θ-tree does not require any particular way of balancing. The only requirement is a time complexity $O(\log n)$ for inserting or deleting a leaf, and time complexity $O(1)$ for finding a root node.

According to authors experience, the fastest way to implement Θ-tree is to make the shape of the tree fixed during the whole computation. I.e. we start with the perfectly balanced tree which represents all activities on the resource. To indicate that an activity i is not in the set Θ it is enough to set $\Sigma\mathrm{P}_i = 0$ and $\mathrm{ECT}_i = -\infty$. Clearly, these

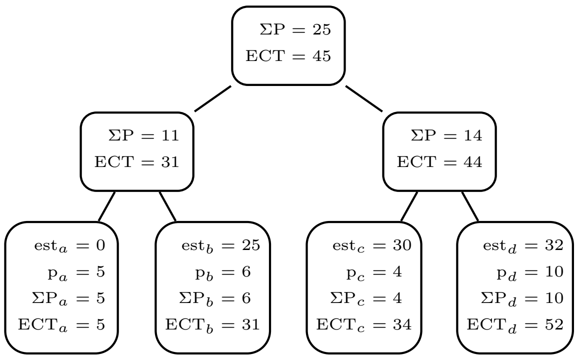

Figure 1. An example of a Θ-tree for $\Theta = \{a, b, c, d\}$.

Table I. Time complexities of operations on 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	$O(1)$

additional "empty" leaves will not interfere with the formulae (2) and (4).

3. Overload checking using Θ-tree

Let us consider an arbitrary set $\Omega \subseteq T$. The overload rule (see e.g. [13]) says that if the set Ω cannot be processed within its time bounds then no solution exists:

$$\forall \Omega \subseteq T : \quad (\mathrm{est}_\Omega + \mathrm{p}_\Omega > \mathrm{lct}_\Omega \quad \Rightarrow \quad \mathrm{fail}) \tag{5}$$

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

$$\Theta(j) = \{k,\ k \in T \ \&\ \mathrm{lct}_k \leq \mathrm{lct}_j\}$$

Overloaded set Ω with $\mathrm{lct}_\Omega = \mathrm{lct}_j$ exists if and only if $\mathrm{ECT}_{\Theta}(j) > \mathrm{lct}_j$. The idea of the algorithm is to take activities j in ascending order by lct_j and adjust the set Θ accordingly.

```
Θ := ∅;
for $j \in T$ in ascending order of $\mathrm{lct}_j$ do begin
   $\Theta := \Theta \cup \{j\}$;
   if $\mathrm{ECT}_\Theta > \mathrm{lct}_j$ 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 a 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:

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

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:

$$\mathrm{lct}_i := \min\left\{\mathrm{lct}_i,\ \max\left\{\mathrm{lct}_j - \mathrm{p}_j,\ j \in \Omega\right\}\right\} \tag{7}$$

There are two versions of the not-first/not-last algorithms: [2] and [10]. Both of them have time complexity $O(n^2)$. The first algorithm [2] 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 [10] is simpler and faster, but more iterations of the algorithm may be needed to reach a fixpoint.

The algorithm presented here may also need more iterations to reach a fixpoint then the algorithm [2] maybe even more than the algorithm [10]. However, time complexity is reduced from $O(n^2)$ to $O(n \log n)$.

Let us suppose that we have chosen a particular activity i and now we want to update lct_i according to the not-last rule. To really achieve some change of lct_i using the rule (7), the set Ω must fulfil the 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 [10]: Because the search for the *best* update of lct_i may be time consuming we will simply search for *some* update of lct_i. If lct_i can be updated better, it will be done in subsequent runs of the algorithm. In fact, our algorithm updates lct_i to

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

which is the "smallest" possible update among all possible updates[3].

Let us define the set $\Theta(i)$:

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

Thus: lct_i can be changed according to the rule not-last if and only if there is some set $\Omega \subseteq (\Theta(i) \setminus \{i\})$ for which the inequality (6) holds. The only problem is to decide whether such a set Ω exists or not.

Let us recall the definition (1) of ECT and use it on the set $\Theta(i)\setminus\{i\}$:

$$\mathrm{ECT}_{\Theta(i)\setminus\{i\}} = \max\left\{\mathrm{est}_\Omega + \mathrm{p}_\Omega,\ \Omega \subseteq \Theta(i) \setminus \{i\}\right\}$$

Notice, that $\mathrm{ECT}_{\Theta(i)\setminus\{i\}}$ is exactly the maximum value which can be on the left side of the inequality (6). Therefore there is a set Ω for which the inequality (6) holds if and only if:

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

The algorithm proceeds as follows. Activities i are taken in the ascending order of lct_i. For each single activity i the set $\Theta(i)$ is computed by extending the set $\Theta(i)$ of previous activity i[4]. For each i $\mathrm{ECT}_{\Theta(i)\setminus\{i\}}$ is checked and lct_i is eventually updated:

```
1  **for** $i \in T$ **do**
2     $\mathrm{lct}'_i$ := $\mathrm{lct}_i$ ;
3  $\Theta$ := $\emptyset$ ;
4  Q := queue of all activities $j \in T$ in ascending order of $\mathrm{lct}_j - \mathrm{p}_j$ ;
5  **for** $i \in T$ in ascending order of $\mathrm{lct}_i$ **do begin**
```

[3] Notice that the condition $i \neq j$ is not really necessary. If there exists at least one $j \neq i$ such that $\mathrm{lct}_j - \mathrm{p}_j < \mathrm{lct}_i$ then even in the case $\mathrm{lct}_i - \mathrm{p}_i = \max\left\{\mathrm{lct}_j - \mathrm{p}_j,\ j \in T \ \&\ \mathrm{lct}_j - \mathrm{p}_j < \mathrm{lct}_i\right\}$ this value provides legal update of lct_i. This property is used at line 13 of the algorithm.

[4] Using the simple fact that $\mathrm{lct}_k \leq \mathrm{lct}_l$ implies $\Theta(k) \subseteq \Theta(l)$, i.e. the $\Theta(i)$ sets are nested in the ascending order of lct_i.

```
6      while $\mathrm{lct}_i > \mathrm{lct}_{\mathrm{Q.first}} - \mathrm{p}_{\mathrm{Q.first}}$ do begin
9        $j$ := Q.first;
10       $\Theta$ := $\Theta \cup \{j\}$;
11       Q.dequeue;
12     end;
13     if $\mathrm{ECT}_{\Theta \setminus \{i\}} > \mathrm{lct}_i - \mathrm{p}_i$ then
14       $\mathrm{lct}'_i$ := $\min\left\{\mathrm{lct}_j - \mathrm{p}_j,\ \mathrm{lct}'_i\right\}$;
15   end;
16   for $i \in T$ do
17     $\mathrm{lct}_i$ := $\mathrm{lct}'_i$;
```

Lines 9–11 are repeated n times maximum over all iterations of the for cycle on the line 5, because each time an activity is removed from the queue. Check on the line 13 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 Q.first just before the insertion of the activity Q.first into the set Θ (i.e. after the line 6):

```
7        if $\mathrm{ECT}_{\Theta} > \mathrm{lct}_{\mathrm{Q.first}} - \mathrm{p}_{\mathrm{Q.first}}$ then
8          $\mathrm{lct}'_{\mathrm{Q.first}}$ := $\mathrm{lct}_j - \mathrm{p}_j$;
```

This modification may in some cases save few iterations of the algorithm.

5. Detectable Precedences

An idea of detectable precedences was introduced in [11] for a *batch resource with sequence dependent setup times*, which is an extension of a unary resource.

The figure 2 is taken from [11]. It shows a situation when neither edge-finding nor the not-first/not-last algorithm can change any time bound, but a propagation of detectable precedences can (see section 6 for details on edge-finding algorithm).

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 the time bounds of these two activities:

$$\mathrm{est}_i + \mathrm{p}_i > \mathrm{lct}_j - \mathrm{p}_j \quad \Rightarrow \quad j \ll i \tag{8}$$

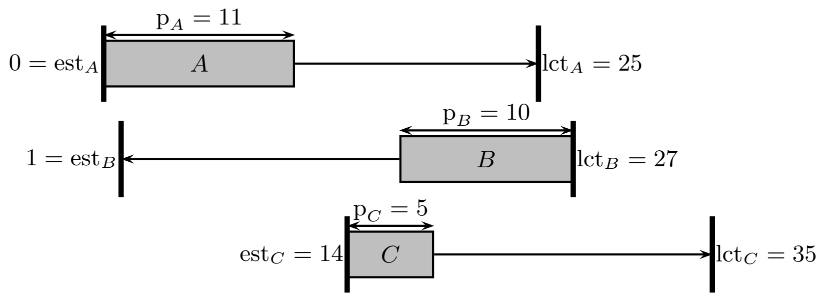

Figure 2. A sample problem for detectable precedences

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 $i \in T$ do begin
  $m := -\infty$;
  for $j \in T$ in non-decreasing order of $\mathrm{est}_j$ do
    if $j \ll i$ then
      $m := \max\{m,\ \mathrm{est}_j\} + \mathrm{p}_j$;
  $\mathrm{est}_i := \max\{m,\ \mathrm{est}_i\}$;
end;
```

A symmetric algorithm adjusts lct_i.

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

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

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

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

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

Our algorithm is based on the observation that set $\Theta(i)$ does not have to be constructed from scratch for each activity i. Rather, the set $\Theta(i)$ can be computed incrementally in the similar way as in the previous section.

```
 1  $\Theta$ := $\emptyset$;
 2  Q := queue of all activities $j \in T$ in ascending order of $\mathrm{lct}_j - \mathrm{p}_j$;
 3  for $i \in T$ in ascending order of $\mathrm{est}_i + \mathrm{p}_i$ do begin
 4    while $\mathrm{est}_i + \mathrm{p}_i > \mathrm{lct}_{\mathrm{Q.first}} - \mathrm{p}_{\mathrm{Q.first}}$ do begin
 5      $\Theta$ := $\Theta \cup \{\mathrm{Q.first}\}$;
 6      Q.dequeue;
 7    end;
 8    $\mathrm{est}'_i$ := $\max\left\{\mathrm{est}_i,\ \mathrm{ECT}_{\Theta \setminus \{i\}}\right\}$;
 9  end;
10  for $i \in T$ do
11    $\mathrm{est}_i$ := $\mathrm{est}'_i$;
```

Initial sorts takes $O(n \log n)$. Lines 5 and 6 are repeated n times maximum over all iterations of the for cycle, 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. Edge-Finding using Θ-Λ-tree

Edge-finding is probably the most often used filtering algorithm for a unary resource constraint. Let us first 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:

$$\forall \Omega \subset T,\ \forall i \in (T \setminus \Omega):$$
$$\mathrm{est}_{\Omega \cup \{i\}} + \mathrm{p}_{\Omega \cup \{i\}} = \min\{\mathrm{est}_\Omega,\ \mathrm{est}_i\} + \mathrm{p}_\Omega + \mathrm{p}_i > \mathrm{lct}_\Omega \quad \Rightarrow \quad \Omega \ll i \tag{9}$$

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

$$\Omega \ll i \quad \Rightarrow \quad \mathrm{est}_i := \max\{\mathrm{est}_i,\ \mathrm{ECT}_\Omega\} \tag{10}$$

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 [8, 9], [5] presents a $O(n \log n)$ algorithm.

In the following we present another edge-finding algorithm with time complexity $O(n \log n)$. It is quite simpler than the algorithm by Carlier and Pinson [5] and it is faster than the quadratic algorithms [8, 9] which are widely used today.

Proposition 2. Let $\Theta(j) = \{k,\ k \in T \ \&\ \mathrm{lct}_k \le \mathrm{lct}_j\}$. Filtering according to rules (9), (10) is equivalent with filtering by the following rule:

$$\begin{aligned}
&\forall j \in T,\ \forall i \in T \setminus \Theta(j) : \\
&\qquad \mathrm{ECT}_{\Theta(j)\cup\{i\}} > \mathrm{lct}_j \ \Rightarrow\ \Theta(j) \ll i \ \Rightarrow \\
&\qquad\qquad \Rightarrow\ \mathrm{est}_i := \max\left\{est_i,\ \mathrm{ECT}_{\Theta(j)}\right\} \qquad (11)
\end{aligned}$$

Proof. We will prove the equivalence by proving both implications. First, let us prove that the new rule (11) generates all the changes which the original rules (9) and (10) do.

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):

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

Thus: when the original rule (9) holds for Ω and i, then the new rule (11) holds for $\Theta(j)$ and i too, and the change of est_i is at least the same as the change by the rule (10). Hence the first implication is proved.

Now we will prove the second implication: filtering according to the new rule (11) will not generate any changes which the old rules (9) and (10) cannot prove too.

Let us consider an arbitrary set $\Omega \subseteq T$. Overload rule (5) says that if the set Ω cannot be processed within its time bounds then no solution exists:

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

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

Let us consider a pair of activities i, j for which the new rule (11) holds. We define a set Ω' as a subset of $\Theta(j) \cup \{i\}$ for which:

$$\mathrm{ECT}_{\Theta(j)\cup\{i\}} = \mathrm{est}_{\Omega'} + \mathrm{p}_{\Omega'} \qquad (12)$$

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

$$\mathrm{est}_{\Omega'} + \mathrm{p}_{\Omega'} \overset{(12)}{=} \mathrm{ECT}_{\Theta(j)\cup\{i\}} \overset{(11)}{>} \mathrm{lct}_j \ge \mathrm{lct}_{\Omega'}$$

So the resource is overloaded (see the overload rule (5)) 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 (11) changes nothing. For this set Ω we have:

$$\min\{\mathrm{est}_\Omega,\ \mathrm{est}_i\} + \mathrm{p}_\Omega + \mathrm{p}_i = \mathrm{est}_{\Omega'} + \mathrm{p}_{\Omega'} \overset{(12)}{=} \mathrm{ECT}_{\Theta(j)\cup\{i\}} \overset{(11)}{>} \mathrm{lct}_j \geq \mathrm{lct}_\Omega$$

Hence the rule (9) holds for the set Ω. To complete the proof we have to show that both rules (10) and (11) 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

$$\mathrm{ECT}_\Omega < \mathrm{ECT}_{\Theta(j)} \tag{13}$$

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

$$\mathrm{ECT}_{\Theta(j)} = \mathrm{est}_\Phi + \mathrm{p}_\Phi \tag{14}$$

Therefore:

$$\mathrm{est}_\Omega + \mathrm{p}_\Omega \leq \mathrm{ECT}_\Omega \overset{(13)}{<} \mathrm{ECT}_{\Theta(j)} \overset{(14)}{=} \mathrm{est}_\Phi + \mathrm{p}_\Phi \tag{15}$$

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

$$\forall k \in \Theta(j) : \quad \mathrm{est}_k \geq \mathrm{est}_{\Omega'} \ \Rightarrow\ k \in \Omega \tag{16}$$

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

$$\forall k \in \Theta(j) : \quad \mathrm{est}_k \geq \mathrm{est}_\Phi \ \Rightarrow\ k \in \Phi \tag{17}$$

Combining properties (16) and (17) 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 $\mathrm{est}_\Phi + \mathrm{p}_\Phi \leq \mathrm{ECT}_\Omega$ which contradicts the inequality (15). The result is that $\Omega \subsetneq \Phi$, and so $\mathrm{p}_\Omega < \mathrm{p}_\Phi$.

Now we are ready to prove the contradiction:

$$\begin{aligned}
&\mathrm{ECT}_{\Theta(j)\cup\{i\}} \overset{(12)}{=} \\
&\quad = \mathrm{est}_{\Omega'} + \mathrm{p}_{\Omega'} \\
&\quad = \min\{\mathrm{est}_\Omega,\ \mathrm{est}_i\} + \mathrm{p}_\Omega + \mathrm{p}_i && \text{because } \Omega = \Omega' \setminus \{i\} \\
&\quad = \min\{\mathrm{est}_\Omega + \mathrm{p}_\Omega + \mathrm{p}_i,\ \mathrm{est}_i + \mathrm{p}_\Omega + \mathrm{p}_i\} \\
&\quad < \min\{\mathrm{est}_\Phi + \mathrm{p}_\Phi + \mathrm{p}_i,\ \mathrm{est}_i + \mathrm{p}_\Phi + \mathrm{p}_i\} && \text{by (15) and } \mathrm{p}_\Omega < \mathrm{p}_\Phi \\
&\quad \leq \mathrm{ECT}_{\Theta(j)\cup\{i\}} && \text{because } \Phi \subseteq \Theta(j)
\end{aligned}$$

□

Property 1. The rule (11) has a very useful property. Let us consider an activity i and two different activities j_1 and j_2 for which the rule (11) 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 (11) holds and lct_j is maximum.

6.1. Θ-Λ-tree

Let us consider the alternative edge-finding rule (11). We choose an arbitrary activity j and check the rule (11) 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 in 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:

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

$$\overline{\mathrm{ECT}}_v = \overline{\mathrm{ECT}}_{\mathrm{Leaves}(v)} = \max\left\{\mathrm{est}_{\Theta'} + \mathrm{p}_{\Theta'},\ \Theta' \subseteq \mathrm{Leaves}(v)\ \&\ |\Theta' \cap \Lambda| \leq 1\right\}$$

$\overline{\Sigma\mathrm{P}}$ 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. For example of the Θ-Λ-tree see figure 3.

An idea how to compute values $\overline{\Sigma\mathrm{P}}_v$ and $\overline{\mathrm{ECT}}_v$ for internal node v follows. A gray activity can be used only once: in the left subtree of v or in the right subtree of v. Note that the gray activity used for $\overline{\Sigma\mathrm{P}}_v$ can be different from the gray activity used for $\overline{\mathrm{ECT}}_v$. The formulae (2) and (4) can be modified to handle gray nodes:

$$\overline{\Sigma\mathrm{P}}_v = \max\left\{\overline{\Sigma\mathrm{P}}_{\mathrm{left}(v)} + \Sigma\mathrm{P}_{\mathrm{right}(v)},\ \Sigma\mathrm{P}_{\mathrm{left}(v)} + \overline{\Sigma\mathrm{P}}_{\mathrm{right}(v)}\right\}$$

$$\overline{\mathrm{ECT}}_v = \max\{\,\overline{\mathrm{ECT}}_{\mathrm{right}(v)}, \qquad \text{(a)}$$

$$\mathrm{ECT}_{\mathrm{left}(v)} + \overline{\Sigma\mathrm{P}}_{\mathrm{right}(v)},\ \overline{\mathrm{ECT}}_{\mathrm{left}(v)} + \Sigma\mathrm{P}_{\mathrm{right}(v)}\,\} \qquad \text{(b)}$$

Line (a) considers all sets Θ' such that $\Theta' \cap \mathrm{Leaves}(\mathrm{left}(v)) \neq \emptyset$ (see the definition (1) of ECT on page 2). Line (b) considers all sets Θ' such that $\Theta' \cap \mathrm{Leaves}(\mathrm{left}(v)) \neq \emptyset$.

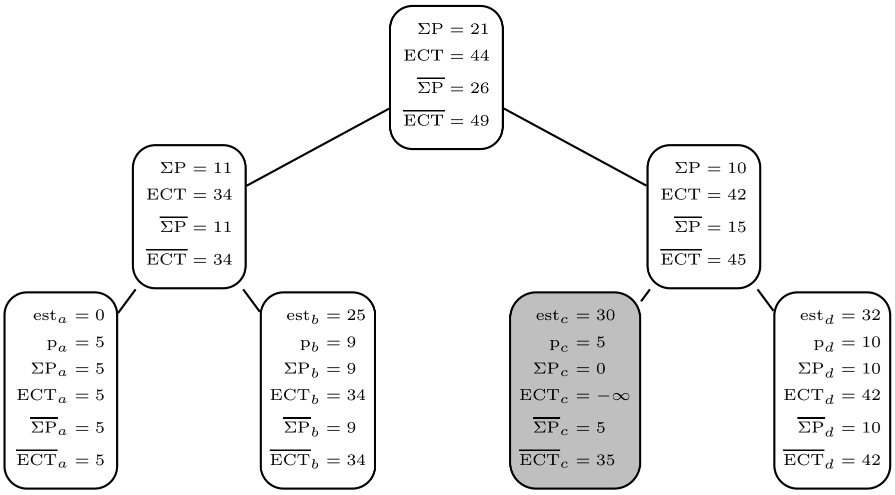
$\Sigma\mathrm{P} = 21$
$\mathrm{ECT} = 44$
$\overline{\Sigma\mathrm{P}} = 26$
$\overline{\mathrm{ECT}} = 49$
$\Sigma\mathrm{P} = 11$
$\mathrm{ECT} = 34$
$\overline{\Sigma\mathrm{P}} = 11$
$\overline{\mathrm{ECT}} = 34$
$\Sigma\mathrm{P} = 10$
$\mathrm{ECT} = 42$
$\overline{\Sigma\mathrm{P}} = 15$
$\overline{\mathrm{ECT}} = 45$
$\mathrm{est}_a = 0$, $\mathrm{p}_a = 5$, $\Sigma\mathrm{P}_a = 5$, $\mathrm{ECT}_a = 5$, $\overline{\Sigma\mathrm{P}}_a = 5$, $\overline{\mathrm{ECT}}_a = 5$
$\mathrm{est}_b = 25$, $\mathrm{p}_b = 9$, $\Sigma\mathrm{P}_b = 9$, $\mathrm{ECT}_b = 34$, $\overline{\Sigma\mathrm{P}}_b = 9$, $\overline{\mathrm{ECT}}_b = 34$
$\mathrm{est}_c = 30$, $\mathrm{p}_c = 5$, $\Sigma\mathrm{P}_c = 0$, $\mathrm{ECT}_c = -\infty$, $\overline{\Sigma\mathrm{P}}_c = 5$, $\overline{\mathrm{ECT}}_c = 35$
$\mathrm{est}_d = 32$, $\mathrm{p}_d = 10$, $\Sigma\mathrm{P}_d = 10$, $\mathrm{ECT}_d = 42$, $\overline{\Sigma\mathrm{P}}_d = 10$, $\overline{\mathrm{ECT}}_d = 42$

Figure 3. An example of a Θ-Λ-tree for $\Theta = \{a, b, d\}$ and $\Lambda = \{c\}$.

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 v the gray activity which is *responsible* for $\overline{\mathrm{ECT}}_k$. We need to know such responsible gray activity in the following algorithms.

Table II shows time complexities of selected operations on Θ-Λ-tree.

6.2. 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.

Table II. Time complexities of 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	$O(1)$

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 13) 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 $i \in T$ do
2     $\mathrm{est}'_i := \mathrm{est}_i$ ;
3  $(\Theta,\ \Lambda) := (T,\ \emptyset)$ ;
4  Q := queue of all activities $j \in T$ in descending order of $\mathrm{lct}_j$ ;
5  $j$ := Q.first ;
6  repeat
7     $(\Theta,\ \Lambda) := (\Theta \setminus \{j\},\ \Lambda \cup \{j\})$ ;
8     Q.dequeue ;
9     $j$ := Q.first ;
10    if $\mathrm{ECT}_\Theta > \mathrm{lct}_j$ then
11       fail ; {Resource is overloaded}
12    while $\overline{\mathrm{ECT}}(\Theta,\ \Lambda) > \mathrm{lct}_j$ do begin
13       $i$ := gray activity responsible for $\overline{\mathrm{ECT}}(\Theta,\ \Lambda)$ ;
14       $\mathrm{est}'_i := \max\{\mathrm{est}_i,\ \mathrm{ECT}_\Theta\}$ ;
15       $\Lambda := \Lambda \setminus \{i\}$ ;
16    end ;
17 until Q.size = 0 ;
18 for $i \in T$ do
19    $\mathrm{est}_i := \mathrm{est}'_i$ ;
```

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, the maximum number of iterations of the inner while loop is n, because each iteration removes an activity from the set Λ. Similarly, the number of iterations of the repeat loop is n, because each time an activity is removed from the queue Q. According to table II 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.

7. 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 times of activities are changed from constants to domain variables. Several "dummy" activities with possible processing times $\langle 0, \infty)$ are added on the resource as a reserve for later activity addition. Filtering algorithms work as usual, but they use minimal 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 a regular activity (i.e. minimal processing time is no longer zero). The main disadvantage of this approach is that an impossibility of a particular 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 the previous one 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 [13] designed to handle this type of alternatives.

To handle optional activities we extend each activity i by a variable called $\mathrm{existence}_i$ with the domain $\{\mathrm{true},\ \mathrm{false}\}$. When $\mathrm{existence}_i = \mathrm{true}$ then i is a regular activity, when $\mathrm{existence}_i \in \{\mathrm{true},\ \mathrm{false}\}$ then i is an optional activity. Finally when $\mathrm{existence} = \mathrm{false}$ 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:
$$\mathrm{est}_\Omega + \mathrm{p}_\Omega > \mathrm{lct}_\Omega$$
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 $\mathrm{existence}_i = \mathrm{false}$.
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 $\mathrm{est}_k + \mathrm{p}_k > \mathrm{lct}_k$. In that case we can also propagate $\mathrm{existence}_i = \mathrm{false}$.

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".

8. Overload Checking with Optional Activities

In this section we present modified overload checking algorithm which can handle optional activities. Basically, the original overload rule (5) remains valid, however we must consider regular activities R only:

$$\forall \Omega \subseteq R: \quad (\mathrm{lct}_\Omega - \mathrm{est}_\Omega < \mathrm{p}_\Omega \quad \Rightarrow \quad \mathrm{fail})$$

In section 3 we showed that this rule is equivalent with:

$$\forall j \in R: \quad \left(\mathrm{ECT}_{\Theta(j)} > \mathrm{lct}_j \quad \Rightarrow \quad \mathrm{fail}\right) \tag{18}$$

where $\Theta(j)$ is:

$$\Theta(j) = \{k,\ k \in R\ \&\ \mathrm{lct}_k \leq \mathrm{lct}_j\}$$

Let us now take into account an optional activity $o \in O$. If the processing of this activity would result in overloading, then the activity can never be processed by the resource:

$$\forall o \in O,\ \forall \Omega \subseteq R:$$
$$\left(\mathrm{est}_{\Omega \cup \{o\}} + \mathrm{p}_{\Omega \cup \{o\}} > \mathrm{lct}_{\Omega \cup \{o\}} \quad \Rightarrow \quad \mathrm{existence}_o := \mathrm{false}\right) \tag{19}$$

Let the set $\Lambda(j)$ be defined in the following way:

$$\Lambda(j) = \{o,\ o \in O\ \&\ \mathrm{lct}_o \leq \mathrm{lct}_j\}$$

The rule (19) is applicable if and only if there is such an activity $j \in T$ such that $\overline{\mathrm{ECT}}(\Theta(j), \Lambda(j)) > \mathrm{lct}_j$. In this case, the activity which is responsible for $\overline{\mathrm{ECT}}(\Theta(j), \Lambda(j))$ can be excluded from the resource.

The following algorithm detects overloading, it also deletes all optional activities k such that an addition of this activity k alone causes an overload. Of course, a combination of several optional activities may still cause an overload.

```
(Θ, Λ) := (∅, ∅);
for $i \in T$ in ascending order[5] of $\mathrm{lct}_i$ do begin
  if $i$ is a optional activity then
    $\Lambda := \Lambda \cup \{i\}$;
  else begin
    $\Theta := \Theta \cup \{i\}$;
    if $\mathrm{ECT}_\Theta > \mathrm{lct}_i$ then
      fail; {No solution exists}
    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;
end;
```

The time 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. The outer for loop has also n iterations, time complexity of each single line is $O(\log n)$ maximum (see the table II).

9. 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, the modified algorithm uses Θ-Λ-tree instead. Optional activities are represented by gray nodes of the tree. For regular propagation, value ECT_Θ is used the same way as before. However, also $\overline{\mathrm{ECT}}(\Theta, \Lambda)$ is used now: If propagation using $\overline{\mathrm{ECT}}(\Theta, \Lambda)$ would result in an immediate fail then the optional activity responsible for that is excluded from the resource.

Let us demonstrate this idea on the detectable precedences algorithm:

```
$(\Theta, \Lambda) := (\emptyset, \emptyset)$;
Q := queue of all activities $j \in T$ in ascending order of $\mathrm{lct}_j - \mathrm{p}_j$;
for $i \in T$ in ascending order of $\mathrm{est}_i + \mathrm{p}_i$ do begin
  while $\mathrm{est}_i + \mathrm{p}_i > \mathrm{lct}_{\mathrm{Q.first}} - \mathrm{p}_{\mathrm{Q.first}}$ do begin
    if $i$ is a regular activity then
      $\Theta := \Theta \cup \{\mathrm{Q.first}\}$;
    else
      $\Lambda := \Lambda \cup \{\mathrm{Q.first}\}$;
```

[5] In case of ties optional activities should be taken before regular activities.

```
 9        Q.dequeue;
10     end;
11     $\mathrm{est}'_i$ := max$\left\{\mathrm{est}_i,\ \mathrm{ECT}_{\Theta\setminus\{i\}}\right\}$;
12     if $i$ is a regular activity then
13        while $\overline{\mathrm{ECT}}(\Theta \setminus \{i\}, \Lambda) > \mathrm{lct}_i - \mathrm{p}_i$ then begin
14           $k$ := an optional activity responsible for $\overline{\mathrm{ECT}}(\Theta \setminus \{i\}, \Lambda)$;
15           $\Lambda$ := $\Lambda \setminus \{k\}$;
16           $\mathrm{existence}_k$ := false;
17        end;
18  end;
19  for $i \in T$ do
20     $\mathrm{est}_i$ := $\mathrm{est}'_i$;
```

Optional activities are stored in the set Λ, therefore they are not taken into account when the new value est'_i is computed at the line 11.

At line 13 there is a check what would happen if one of the optional activities become regular one. In that case est'_i would be $\overline{\mathrm{ECT}}(\Theta \setminus \{i\}, \Lambda)$. If $\mathrm{est}'_i > \mathrm{lct}_i - \mathrm{p}_i$ then the activity k can be excluded from the resource.

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:

```
 1  for $i \in T$ do
 2     $\mathrm{lct}'_i$ := $\mathrm{lct}_i$;
 3  $(\Theta, \Lambda)$ := $(\emptyset, \emptyset)$;
 4  Q := queue of all activities $j \in T$ in ascending order of $\mathrm{lct}_j - \mathrm{p}_j$;
 5  for $i \in T$ in ascending order of $\mathrm{lct}_i$ do begin
 6     while $\mathrm{lct}_i > \mathrm{lct}_{\mathrm{Q.first}} - \mathrm{p}_{\mathrm{Q.first}}$ do begin
 9        if Q.first is regular activity then begin
10           $j$ := Q.first;
11           $j'$ := Q.first;
12           $\Theta$ := $\Theta \cup \{j\}$;
13        end else begin
14           $j'$ := Q.first;
15           $\Lambda$ := $\Lambda \cup \{j'\}$;
16        end;
17        Q.dequeue;
18     end;
19     if $\mathrm{ECT}_{\Theta\setminus\{i\}} > \mathrm{lct}_i - \mathrm{p}_i$ then
20        $\mathrm{lct}'_i$ := min$\left\{\mathrm{lct}_j - \mathrm{p}_j,\ \mathrm{lct}'_i\right\}$;
21     if $i$ is regular activity and $\mathrm{lct}_{j'} - \mathrm{p}_{j'} < \mathrm{est}_i + \mathrm{p}_i$ then
22        while $\overline{\mathrm{ECT}}(\Theta \setminus \{i\}, \Lambda) > \mathrm{lct}_i - \mathrm{p}_i$ do begin
```

```
23            $k$ := an optional activity responsible for $\overline{\mathrm{ECT}}(\Theta \setminus \{i\}, \Lambda)$;
24            $\Lambda$ := $\Lambda \setminus \{k\}$;
25            existence$_k$ := false;
26         end;
27   end;
28   for $i \in T$ do
29      lct$_i$ := lct$'_i$;
```

Again, optional activities are stored in the gray nodes of the Θ-Λ-tree, therefore they do not influence new time bound lct'_i computed at the line 20. At the line 25 we are allowed to set $\mathrm{existence}_k$:= false, because at this point we know that if the activity k become regular, then the propagation would take place (compare lines 22 and 19), however such propagation would result in fail (because $\mathrm{lct}'_i = \mathrm{lct}_{j'} - \mathrm{p}_{j'} < \mathrm{est}_i + \mathrm{p}_i$, see line 21). Time complexity of this algorithm is $O(n \log n)$.

Unfortunately, extending the edge-finding algorithm is not so easy because this algorithm already uses Θ-Λ-tree. We will consider this in our future work.

10. Experimental Results

First, we tested our propagation algorithms without optional activities on several jobshop problems taken from the OR library [1]. The benchmark problem is to compute a destructive lower bound using the shaving technique [8]. The destructive lower bound is the minimal makespan for which propagation alone is not able to find conflict without backtracking. 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. To limit CPU time, shaving was used for each activity only once. For more details about shaving see [8].

Table III shows the results. We measured the CPU[6] time in seconds needed to prove the lower bound. I.e. the propagation is done twice: with the upper bound LB and LB-1. Column T1 shows total running time when presented $O(n \log n)$ filtering algorithms are used (overload checking, detectable precedences, not-first/not-last and edge-finding). Column T2 shows total running times when quadratic algorithms are used: quadratic overload checking, not-first/not-last from [10], edge-finding from [9] and $O(n \log n)$ detectable precedences.

As can be seen, the new algorithms are strictly faster and the speedup is increasing with a growing number of jobs.

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

Table III. Destructive Lower Bounds

Prob.	**Size**	**LB**	**T1**	**T2**
abz5	10 x 10	1196	1.363	1.696
abz6	10 x 10	941	1.717	2.156
ft10	10 x 10	911	1.530	1.949
orb01	10 x 10	1017	1.649	2.123
orb02	10 x 10	869	1.415	1.796
la21	15 x 10	1033	0.691	1.030
la22	15 x 10	925	3.230	4.902
la36	15 x 15	1267	5.012	7.854
la37	15 x 15	1397	2.369	3.584
ta01	15 x 15	1224	8.641	13.66
ta02	15 x 15	1210	6.618	10.41
la26	20 x 10	1218	0.597	1.030
la27	20 x 10	1235	0.745	1.260
la29	20 x 10	1119	2.949	4.954
abz7	20 x 15	651	2.973	4.967
abz8	20 x 15	621	10.71	18.36
ta11	20 x 15	1295	13.24	23.26
ta12	20 x 15	1336	15.64	26.60
ta21	20 x 20	1546	34.98	61.63
ta22	20 x 20	1501	23.06	40.83
yn1	20 x 20	816	24.35	42.70
yn2	20 x 20	842	20.77	36.19
ta31	30 x 15	1764	3.397	7.901
ta32	30 x 15	1774	4.829	11.64
swv11	50 x 10	2983	12.02	39.24
swv12	50 x 10	2972	15.43	46.32
ta51	50 x 15	2760	7.695	26.14
ta52	50 x 15	2756	8.056	27.37
ta71	100 x 20	5464	72.07	432.5
ta72	100 x 20	5181	72.15	432.1

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 IV 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 experiments should be made, especially on real life problem instances.

Table IV. Alternative activities

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

10.0.1. *Acknowledgements:*
This work has been supported by the Czech Science Foundation under the contract no. 201/04/1102.

References

1. 'OR Library'.
2. Baptiste, P. and C. Le Pape: 1996, 'Edge-Finding Constraint Propagation Algorithms for Disjunctive and Cumulative Scheduling'. In: *Proceedings of the Fifteenth Workshop of the U.K. Planning Special Interest Group.*
3. Barták, R.: 2003, 'Dynamic Global Constraints in Backtracking Based Environments'. *Annals of Operations Research* **118**, 101–118.
4. Beck, J. C. and M. S. Fox: 1999, 'Scheduling Alternative Activities'. In: *AAAI/IAAI.* pp. 680–687.
5. Carlier, J. and E. Pinson: 1994, 'Adjustments of Head and Tails for the Job-Shop Problem'. *European Journal of Operational Research* **78**, 146–161.
6. Focacci, F., P. Laborie, and W. Nuijten: 2000, 'Solving scheduling problems with setup times and alternative resources'. In: *Proceedings of the 5th International Conference on Artificial Intelligence Planning and Scheduling.*
7. Garey, M. R. and D. S. Johnson: 1979, *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H.Freeman and Company, San Francisco.

8. Martin, P. and D. B. Shmoys: 1996, ‘A New Approach to Computing Optimal Schedules for the Job-Shop Scheduling Problem’. In: W. H. Cunningham, S. T. McCormick, and M. Queyranne (eds.): *Proceedings of the 5th International Conference on Integer Programming and Combinatorial Optimization, IPCO’96*. Vancouver, British Columbia, Canada, pp. 389–403.
9. Philippe Baptiste, C. L. P. and W. Nuijten: 2001, *Constraint-Based Scheduling: Applying Constraint Programming to Scheduling Problems*. Kluwer Academic Publishers.
10. Torres, P. and P. Lopez: 1999, ‘On Not-First/Not-Last conditions in disjunctive scheduling’. *European Journal of Operational Research*.
11. Vilím, P.: 2002, ‘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.
12. Vilím, P.: 2004, ‘$O(n \log n)$ Filtering Algorithms for Unary Resource Constraint’. In: *Proceedings of CP-AI-OR 2004*.
13. Wolf, A. and H. Schlenker: 2004, ‘Realizing the Alternative Resources Constraint Problem with Single Resource Constraints’. In: *To appear in proceedings of the INAP workshop 2004*.