Archives of Control Sciences
Volume 18(LIV), 2008
No. 2, *pages 159–202*

Filtering algorithms for the unary resource constraint

PETR VILÍM

Scheduling is one of the most successful application areas of constraint programming mainly due to special global constraints designed to model resource restrictions. Among scheduling constraints, the most useful and most studied constraint is probably the unary resource constraint. This paper presents state-of-the-art filtering algorithms for this important constraint. These algorithms are very fast (almost all of them has time complexity $O(n \log n)$) and furthermore they are able to take into account so called optional activities, that is, activities which may or may not appear in the schedule depending for example on a resolution of an alternative processing rule(s). In particular, this paper presents the following algorithms: overload checking, edge finding, not-first/not-last, detectable precedences and precedence energy.

Key words: constraint programming, scheduling, optional activities, unary resource, filtering algorithms

1. Introduction

In this article we consider a scheduling problem which is to assign exact dates to a set of known activities so that the resulting schedule satisfy all user-defined constraints. Typically these constraints include:

- *Release dates:* an activity cannot start before given date (for example the activity waits for a delivery of a material by an external supplier).
- *Due dates:* an activity must end before given date (for example there are some deadlines which must be respected).
- *Processing times:* each activity has some duration. For simplicity we will assume in this paper only non-interruptible activities with predetermined processing times.
- *Precedences:* typically there are dependencies between activities which require, for example, that one particular activity can start only after another particular activity completes.

P. Vilím is with ILOG S.A., 9, rue de Verdun, BP 85, F-94253 Gentilly Cedex, France, e-mail: pvilim@ilog.fr

Received 13.05.2008.

- *Unary resource constraints:* some activities may require a unique resource during their processing (e.g. a particular skilled worker, a particular machine or a particular room) which cannot be used by two activities at the same time. Therefore these activities cannot overlap in time.

- *Cumulative resource constraints:* can be used to model a limited pool of equivalent workers. Each activity may require some number of workers during its processing, the constraint assures that at any time in the resulting schedule the total number of used workers does not exceed the size of the pool.

- *Reservoirs:* can be used to model a quantity of an intermediate product during time. Some activities may produce this intermediate product (that is they increase the level of the reservoir) while others may consume it (they decrease the level of the reservoir). The constraint assures that the level never gets bellow zero and never exceeds given amount (maximum storage capacity).

In the first part of the paper we will assume that all of the activities must be executed (that is, we have to assign starting times to all activities). In the second part we will extend the problem by so called optional activities, which does not have to be necessarily executed. For example, there may be some alternative production recipes and part of the scheduling problem is to decide which one of them best suits current needs (and activities in all other alternatives are abandoned).

In general the scheduling problem described above is NP-hard. Already one unary resource constraint over activities with release and due dates is NP-hard in the strong sense (see problem [SS1] on page 236 in [15]). That is one of the reasons why Constraint Programming (CP) turned out to be one of the most effective approaches in solving such problems.

In CP we see each activity i as a *decision variable* with the following attributes:

- the earliest possible starting time est_i,

- the latest possible completion time lct_i,

- the processing time p_i.

The values est_i and lct_i are initialized by release dates and due dates (or they are set to $-\infty$ and ∞ if there are no such dates) but they will not stay constant. The key idea of CP is that although we do not know exact dates yet, we try to eliminate infeasible dates from the time windows $\langle \mathrm{est}_i, \mathrm{lct}_i \rangle$ by carefully reasoning about each constraint separately[1]. Each constraint has associated so called *propagation* (or *filtering*) algorithm which removes such infeasible values by contracting intervals $\langle \mathrm{est}_i, \mathrm{lct}_i \rangle$.

[1] To be precise, sometimes we even consider several constraints at the same time in the reasoning – see for example Precedence Energy in Section 2.6

Due to the NP-hardness of some constraints, it is not tractable to remove all infeasible values. Instead, it is customary to use fast but incomplete filtering algorithms which can remove only some of the impossible assignments.

Another key point of CP is that filtering algorithms for different constraints are able to cooperate, i.e., when one filtering algorithm removes an infeasible value, other algorithms are able to 'react': they take this new information into account and try to further restrict other decision variables. This way, filtering algorithms are usually repeated until they reach a fixpoint. For more information about that technique and CP in general, see for example [13].

When a fixpoint is reached, it is usually still not a solution. Therefore it is necessary to use some search technique (depth first search for example) to explore the remaining search space. However, filtering algorithms are still in use to further restrict the variables after each decision. Thus filtering algorithms are repeated in every node of the search tree and everything depends on their filtering power and their speed. Note that with depth first search, it is necessary to restore all values of est_i and lct_i upon backtracking (for example using a trail).

This article has the following structure. First, we finish the introduction by giving a brief historical review on the subject and deeper explanation of the fixpoint. Then Section 2 'Propagation Rules' presents the mathematical rules which drive the algorithms and prove their correctness. This is followed by Section 3 'Filtering Algorithms' which presents the actual algorithms and the data structures behind them. And finally Section 4 'Optional Activities' describe how to take into account optional activities. The paper does not contain any experimental results of discussed algorithms, interested reader can find them in [26, 27, 28].

1.1. Historical review: unary resource

During the time there were designed several filtering algorithms for the unary resource constraint. Often these algorithms remove different types of inconsistent values therefore they can be used together.

The following paragraphs are dedicated to the brief history of the most famous of them. More details about each algorithm will be provided in the next chapters.

1.1.1. Edge finding

Edge Finding is the oldest and the most famous filtering algorithm for the unary resource constraint. The first version of this algorithm was proposed by Jacques Carlier and Eric Pinson in [10]. Their algorithm has time complexity $O(n \log n)$, but it is quite complicated to implement.

Later other two versions of this algorithm were developed by Paul Martin with David B. Shmoys [18] and Wim Nuijten [22]. These versions have time complexity $O(n^2)$, but are much easier to implement and therefore they are widely used today.

This article describes a new version of this algorithm with time complexity $O(n \log n)$, which is not so hard to implement as the algorithm in [10]. Together with Roman Barták and Ondřej Čepek I published the algorithm in [27] and [28].

1.1.2. Not-first/not-last

The first Not-First/Not-Last algorithm was introduced by Philippe Baptiste and Claude Le Pape in [5], its time complexity is $O(n^2)$.

Later Philippe Torres and Pierre Lopez designed a simpler and faster version of this algorithm [24], but its worst-case time complexity remains the same – $O(n^2)$. To achieve the same filtering as the original algorithm [5], more iterations of the algorithm may be needed. Nevertheless this algorithm is faster than the previous one.

In this article I describe a new version of this algorithm with worst-case time complexity $O(n \log n)$. I already published the algorithm in [26] and [28]. Like the algorithm [24], more iterations may be needed in order to achieve the filtering of the original algorithm [5], but the algorithm is faster than [24].

1.1.3. Detectable precedences

This is a new algorithm with worst-case time complexity $O(n \log n)$ which I introduced in [26]. The algorithm was also published in [28].

1.1.4. Precedence energy (precedence graph)

Propagation based on precedence energy was mentioned by several authors, e.g., by Brucker in [8] or by Focacci, Laborie and Nuijten in [14]. In this article I present a simple $O(n^2)$ algorithm which was also published in [26]. This algorithm is tightly related to Detectable Precedences algorithm.

1.1.5. Overload checking

Overload Checking was originally part of the Edge Finding [10, 18, 22]. However, for quick computation of the fixpoint (see the next chapter), it is useful to separate these two algorithms.

Overload Checking is not a true filtering algorithm – it does not propagate. However by detecting so called *overloaded intervals* it stops the propagation when no solution can exist.

1.1.6. Other algorithms

There is a number of other algorithms which are not described in this paper. For example:

- *Sweeping:* Sweeping is another propagation technique for unary resource constraint. Sweeping algorithm proposed by Wolf in [30] has the same propagatison power as algorithms Edge Finding and Not-First/Not-Last together. The worst-case time complexity of this algorithm is $O(n^2)$.

- *Task Intervals:* Propagation using task intervals is presented in [11, 12]. Unlike the algorithms mentioned above this technique stores some data from one run of the algorithm to another. Using this data it is able to react to the changes since the last run more quickly. The theoretical worst-case time complexity of this algorithm is $O(n^3)$ however in practice it should be considerably faster.
- *Input-or-Output Test:* is an $O(n^2)$ algorithm for more sophisticated detection of necessary precedences on an unary resource, see [9].
- *Position-based propagation:* which improves temporal bounds by reasoning about sequence number of each activity on an unary resource [20].

1.2. Other scheduling constraints

Although the main topic of this paper is the unary resource constraint, let us quickly mention some filtering algorithms for other kind of resource constraints.

1.2.1. Cumulative resource constraint

As cumulative resource can be seen as an extension of unary resource constraint, it is possible to extend some filtering algorithms for unary resource to the cumulative version. Examples of such algorithms are Overload Checking [31], Edge Finding [19] and Not-First/Not-Last [23]. There are also specific filtering algorithms for cumulative resource, for example Energetic Reasoning [3, 4]. With exception of Overload Checking these algorithms have bigger time complexity than algorithms for unary resource constraint, for this reason it pays off to use also Time Table propagation [21] which is less powerful but incremental and therefore very fast.

1.2.2. Reservoir

Reservoir is the most complicated of the mentioned resource constraints. It is possible to use Time Table propagation for it, a good example of more powerful filtering algorithm is Balance Constraint [17].

1.3. Fixpoint

With the exception of the Overload Checking, all filtering algorithms which will be further discussed in this paper are not idempotent. It means that after a successful run of the algorithm, a subsequent run of the same algorithm can find more changes. To achieve the maximum pruning we have to iterate the algorithm until no more changes are found.

Moreover several filtering algorithms can be used together because each one removes different types of inconsistencies. Thus we compute a fixpoint – a state when no filtering algorithm is able to find any more changes. The process of computation of the fixpoint is illustrated by Algorithm 1.

Algorithm 1: Computation of the fixpoint

```
1  repeat
2      repeat
3          repeat
4              repeat
5                  if not Overload_Checking then
6                      fail;
7                  Detectable_Precedences;
8              until no more propagation;
9              Not_First/Not_Last;
10         until no more propagation;
11         Edge_Finding;
12     until no more propagation;
13     Precedence_Energy;
14 until no more propagation;
```

As Krzysztof Apt proved by the Domain Reduction Theorem [2], if all propagation rules are monotonic (and that is our case) then the sequence in which the filtering algorithms are called is not important – the resulting fixpoint will be always the same. However total running time depends on the sequence significantly. It pays off to call first algorithms which are fast and find most updates, for example as proposed in Algorithm 1.

Thus monotonicity of the filtering algorithms is quite important. Informally, a filtering algorithm f is monotonic if it cannot miss a propagation just because another filtering algorithm g was executed before it. That is, let us consider two input data for the filtering algorithm f, the second input data are derived from the first one by applying filtering algorithm g. If f is monotonic then its application on the first input data cannot yield better result than on the second one.

2. Propagation rules

In this chapter we define rules which are the roots of propagation algorithms for the unary resource in this article. Sometimes we will study equivalence of different formulations of the same rule (for example for Edge Finding). In some cases we will study how different rules relate with each other (Detectable Precedences and Precedence Energy). The algorithms themselves will be presented later in Section 3.

2.1. Basic notation

In this section we establish basic notation about unary resource and activities. For an activity i we already defined the earliest start time est_i, the latest completion time lct_i and the processing time p_i. Now, we will extend this notation for a set of activities.

Let T be the set of all activities on the resource and let $\Omega \subseteq T$ be an arbitrary non-empty subset of T. The earliest starting time est_Ω, the latest completion time lct_Ω and the processing time p_Ω of the set Ω are defined as:

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

Often we will need a lower bound of the earliest completion time of the set Ω. Computation of the true lower bound would be slow because the problem is NP-hard. Therefore we use the following lower bound instead:

$$\mathrm{ect}_\Omega = \max\left\{\mathrm{est}_{\Omega'} + \mathrm{p}_{\Omega'},\ \Omega' \subseteq \Omega\right\} \tag{1}$$

Symmetrically we define an estimation of the latest start time:

$$\mathrm{lst}_\Omega = \min\left\{\mathrm{lct}_{\Omega'} - \mathrm{p}_{\Omega'},\ \Omega' \subseteq \Omega\right\} \tag{2}$$

To extend the definitions also for $\Omega = \emptyset$ let:

$$\mathrm{est}_\emptyset = -\infty \quad \mathrm{lct}_\emptyset = \infty \quad \mathrm{p}_\emptyset = 0 \quad \mathrm{ect}_\emptyset = -\infty \quad \mathrm{lst}_\emptyset = \infty$$

2.2. Overload checking

The rule for Overload Checking is the simplest of the presented rules. This rule checks whether it can be quickly proved that the problem has no solution. In that case we can stop the filtering.

The easiest way how to detect an infeasibility in CP is to find a variable with empty domain. For an activity i it means that $\mathrm{est}_i + \mathrm{p}_i > \mathrm{lct}_i$. This check is used whenever est_i or lct_i is changed. For simplicity we do not emphasize this in the filtering algorithms.

However, this simple check can be further extended as follows. Let us consider an arbitrary set $\Omega \subseteq T$. The overload rule (see e.g. [29]) says that if the set Ω cannot be processed within its bounds then no solution exists (see Figure 1):

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

This is a classical formulation of the Overload Checking rule. However for implementation of this rule, an equivalent formulation may be more convenient.

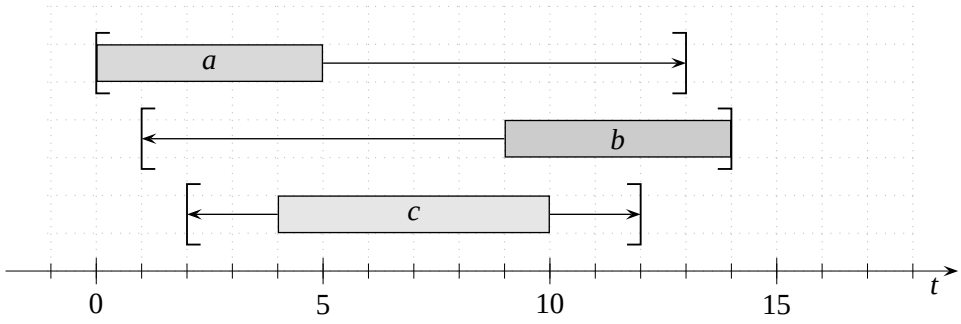

Figure 1. A sample problem for Overload Checking

Let us define a 'left cut by the activity j' as a set:

$$\mathrm{LCut}(T,j) = \left\{k,\ k \in T \ \&\ \mathrm{lct}_k \leqslant \mathrm{lct}_j\right\} \tag{3}$$

The new rule is:

$$\forall j \in T: \quad \left(\mathrm{ect}_{\mathrm{LCut}(T,j)} > \mathrm{lct}_{\mathrm{LCut}(T,j)} \quad \Rightarrow \quad \mathrm{fail}\right) \tag{OL'}$$

On Figure 1, $\mathrm{LCut}(T,b) = \{a,b,c\}$, $\mathrm{est}_{\mathrm{LCut}(T,b)} = 0+5+5+6 = 16$ and $\mathrm{lct}_{\mathrm{LCut}(T,b)} = 14$, therefore the rule (OL') recognizes that no solution exists.

Proposition 1 *The rules (OL) and (OL') are equivalent.*

Proof The proof has two parts. First we prove that when the rule (OL) detects an inconsistency, then the rule (OL') will detect it too. After that we will prove the opposite implication.

1. Let the rule (OL) detects an inconsistency. Let j be such an activity from the set Ω that $\mathrm{lct}_j = \mathrm{lct}_\Omega$. Clearly $\mathrm{lct}_j = \mathrm{lct}_{\mathrm{LCut}(T,j)}$ holds. Moreover $\Omega \subseteq \mathrm{LCut}(T,j)$ and therefore $\mathrm{est}_\Omega + \mathrm{p}_\Omega \leqslant \mathrm{ect}_{\mathrm{LCut}(T,j)}$. So we have:

$$\mathrm{lct}_{\mathrm{LCut}(T,j)} = \mathrm{lct}_j = \mathrm{lct}_\Omega < \mathrm{est}_\Omega + \mathrm{p}_\Omega \leqslant \mathrm{ect}_{\mathrm{LCut}(j)}$$
$$\mathrm{lct}_{\mathrm{LCut}(T,j)} < \mathrm{ect}_{\mathrm{LCut}(T,j)}$$

   and the rule (OL') also detects an inconsistency.

2. Let the rule (OL') detects an inconsistency. We define Ω to be such a subset of $\mathrm{LCut}(T,j)$, that $\mathrm{est}_\Omega + \mathrm{p}_\Omega = \mathrm{ect}_{\mathrm{LCut}(T,j)}$ (due to the definition (1) such a set must exists). And thus:

$$\mathrm{est}_\Omega + \mathrm{p}_\Omega = \mathrm{ect}_{\mathrm{LCut}(T,j)} > \mathrm{lct}_{\mathrm{LCut}(T,j)} \geqslant \mathrm{lct}_\Omega$$
$$\mathrm{est}_\Omega + \mathrm{p}_\Omega > \mathrm{lct}_\Omega$$

   and so the rule (OL) also detects an inconsistency.

□

2.3. Edge finding

Edge Finding is probably the most frequently used filtering algorithm for a unary resource constraint. The algorithm is based on the following rule (see e.g., [5]).

Let us 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 the set Ω (see also Figure 2):

$$\forall \Omega \subset T,\ \forall i \in (T \setminus \Omega): \quad \left(\mathrm{est}_{\Omega\cup\{i\}} + \mathrm{p}_{\Omega\cup\{i\}} > \mathrm{lct}_{\Omega} \ \Rightarrow\ \Omega \ll i\right)$$

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

$$\Omega \ll i \quad \Rightarrow \quad \mathrm{est}_i := \max\{\mathrm{est}_i,\ \mathrm{ect}_{\Omega}\} \tag{4}$$

The whole rule is:

$$\forall \Omega \subset T,\ \forall i \in (T \setminus \Omega): \left(\mathrm{est}_{\Omega\cup\{i\}} + \mathrm{p}_{\Omega\cup\{i\}} > \mathrm{lct}_{\Omega} \ \Rightarrow\ \Omega \ll i \ \Rightarrow\ \mathrm{est}_i := \max\{\mathrm{est}_i,\ \mathrm{ect}_{\Omega}\}\right) \quad \text{(EF)}$$

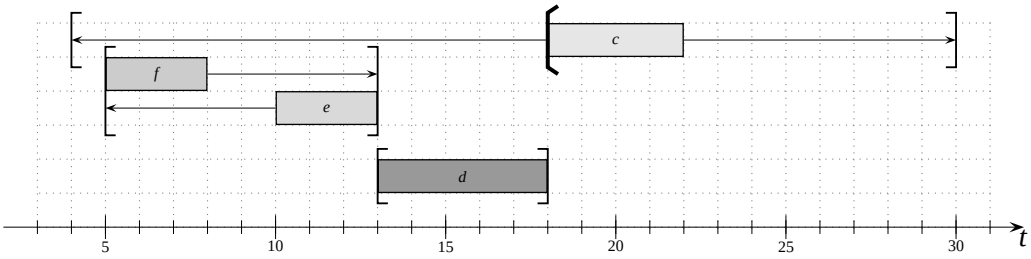

Figure 2. A sample problem for Edge Finding: est_C can be updated from 4 to 18.

There is also a symmetric rule, which adjust lct_i:

$$\forall \Omega \subset T,\ \forall i \in (T \setminus \Omega): \left(\mathrm{lct}_{\Omega\cup\{i\}} - \mathrm{p}_{\Omega\cup\{i\}} < \mathrm{est}_{\Omega} \ \Rightarrow\ i \ll \Omega \ \Rightarrow\ \mathrm{lct}_i := \min\{\mathrm{lct}_i,\ \mathrm{lst}_{\Omega}\}\right)$$

Since the rules are symmetric, in the following we will consider only the first rule (EF).

The traditional rule can be rewritten into an equivalent form, which is more suitable for the algorithm presented later in Section 3.7:

$$\forall j \in T,\ \forall i \in T \setminus \mathrm{LCut}(T, j):$$
$$\mathrm{ect}_{\mathrm{LCut}(T,j)\cup\{i\}} > \mathrm{lct}_j \ \Rightarrow\ \mathrm{LCut}(T, j) \ll i \ \Rightarrow\ \mathrm{est}_i := \max\left\{\mathrm{est}_i,\ \mathrm{ect}_{\mathrm{LCut}(T,j)}\right\} \quad \text{(EF')}$$

Proposition 2 *When the resource is not overloaded according to the rule (OL), then the rules (EF) and (EF') are equivalent.*

Note that if the resource is overloaded then it is needless to compare the rules (EF) and (EF') because the propagation will end by fail anyway.

Proof We will prove the equivalence by proving both implications.

1. First, let us prove that the new rule (EF') generates all the changes which the original rule (EF) does.

   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 $\mathrm{lct}_j = \mathrm{lct}_\Omega$. Due to this definition of j we have $\Omega \subseteq \mathrm{LCut}(T, j)$ and so (recall the definition of ect):

   $$\mathrm{ect}_\Omega \leqslant \mathrm{ect}_{\mathrm{LCut}(T,j)}$$

   And also:

   $$\mathrm{est}_{\Omega\cup\{i\}} + \mathrm{p}_{\Omega\cup\{i\}} = \min\{\mathrm{est}_\Omega, \mathrm{est}_i\} + \mathrm{p}_\Omega + \mathrm{p}_i \leqslant \mathrm{ect}_{\mathrm{LCut}(T,j)\cup\{i\}}$$

   Thus: when the original rule (EF) holds for Ω and i, then the new rule (EF') holds for $\mathrm{LCut}(T, j)$ and i too, and the change of est_i is at least the same as the change by the rule (EF). Hence the first implication is proved.

2. Now we will prove the second implication: filtering according to the new rule (EF') will not generate any changes which the old rule (EF) cannot generate too.

   Let us consider a pair of activities i, j for which the new rule (EF') holds. We define a set Ω' as a subset of $\mathrm{LCut}(T, j) \cup \{i\}$ for which:

   $$\mathrm{ect}_{\mathrm{LCut}(T,j)\cup\{i\}} = \mathrm{est}_{\Omega'} + \mathrm{p}_{\Omega'} \tag{5}$$

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

   If $i \notin \Omega'$ then $\Omega' \subseteq \mathrm{LCut}(T, j)$, therefore

   $$\mathrm{est}_{\Omega'} + \mathrm{p}_{\Omega'} \overset{(5)}{=} \mathrm{ect}_{\mathrm{LCut}(T,j)\cup\{i\}} \overset{(EF')}{>} \mathrm{lct}_j \geqslant \mathrm{lct}_{\Omega'}$$

   So according to the rule (OL) the resource is overloaded. As was noted above, in this case we do not care whether the rules are equivalent or not because the propagation will end by fail anyway.

   Thus $i \in \Omega'$. Let us define $\Omega = \Omega' \setminus \{i\}$. We will assume that $\Omega \neq \emptyset$, because otherwise $\mathrm{est}_i \geqslant \mathrm{ect}_{\mathrm{LCut}(T,j)}$ and the rule (EF') achieves nothing. For this set Ω we have:

   $$\min\{\mathrm{est}_\Omega, \mathrm{est}_i\} + \mathrm{p}_\Omega + \mathrm{p}_i = \mathrm{est}_{\Omega'} + \mathrm{p}_{\Omega'} \overset{(5)}{=} \mathrm{ect}_{\mathrm{LCut}(T,j)\cup\{i\}} \overset{(EF')}{>} \mathrm{lct}_j \geqslant \mathrm{lct}_\Omega$$

Hence the rule (EF) holds for the set Ω.

To complete the proof we have to show that both rules (EF) and (EF') adjust est_i equivalently, i.e., $\text{ect}_\Omega = \text{ect}_{\text{LCut}(T,j)}$. We already know that $\text{ect}_\Omega \leqslant \text{ect}_{\text{LCut}(T,j)}$ because $\Omega \subseteq \text{LCut}(T,j)$. Suppose now for a contradiction that:

$$\text{ect}_\Omega < \text{ect}_{\text{LCut}(T,j)} \tag{6}$$

Let Φ be a set $\Phi \subseteq \text{LCut}(T,j)$ such that:

$$\text{ect}_{\text{LCut}(T,j)} = \text{est}_\Phi + \text{p}_\Phi \tag{7}$$

Therefore:

$$\text{est}_\Omega + \text{p}_\Omega \leqslant \text{ect}_\Omega \overset{(6)}{<} \text{ect}_{\text{LCut}(T,j)} \overset{(7)}{=} \text{est}_\Phi + \text{p}_\Phi \tag{8}$$

Because the set $\Omega' = \Omega \cup \{i\}$ defines the value of $\text{ect}_{\text{LCut}(T,j)\cup\{i\}}$ (because $\text{est}_{\Omega'} + \text{p}_{\Omega'} = \text{ect}_{\text{LCut}(T,j)\cup\{i\}}$), it has the following property (see the definition of ect):

$$\forall k \in \text{LCut}(T,j) \cup \{i\} : \quad \text{est}_k \geqslant \text{est}_{\Omega'} \;\Rightarrow\; k \in \Omega'$$

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

$$\forall k \in \text{LCut}(T,j) : \quad \text{est}_k \geqslant \text{est}_{\Omega'} \;\Rightarrow\; k \in \Omega \tag{9}$$

Similarly, the set Φ defines the value of $\text{ect}_{\text{LCut}(T,j)}$:

$$\forall k \in \text{LCut}(T,j) : \quad \text{est}_k \geqslant \text{est}_\Phi \;\Rightarrow\; k \in \Phi \tag{10}$$

Combining properties (9) and (10) together we have that either $\Omega \subseteq \Phi$ (if $\text{est}_{\Omega'} \geqslant \text{est}_\Phi$) or $\Phi \subseteq \Omega$ (if $\text{est}_{\Omega'} \leqslant \text{est}_\Phi$). However, $\Phi \subseteq \Omega$ is not possible, because in this case $\text{est}_\Phi + \text{p}_\Phi \leqslant \text{ect}_\Omega$ which contradicts the inequality (8). The result is that $\Omega \subsetneq \Phi$, and so $\text{p}_\Omega < \text{p}_\Phi$.

Now we are ready to prove the contradiction:

$$\begin{aligned}
&\text{ect}_{\text{LCut}(T,j)\cup\{i\}} \overset{(5)}{=} \\
&\quad = \text{est}_{\Omega'} + \text{p}_{\Omega'} \\
&\quad = \min\{\text{est}_\Omega,\, \text{est}_i\} + \text{p}_\Omega + \text{p}_i && \text{because } \Omega = \Omega' \setminus \{i\} \\
&\quad = \min\{\text{est}_\Omega + \text{p}_\Omega + \text{p}_i,\, \text{est}_i + \text{p}_\Omega + \text{p}_i\} \\
&\quad < \min\{\text{est}_\Phi + \text{p}_\Phi + \text{p}_i,\, \text{est}_i + \text{p}_\Phi + \text{p}_i\} && \text{by (8) and } \text{p}_\Omega < \text{p}_\Phi \\
&\quad \leqslant \text{ect}_{\text{LCut}(T,j)\cup\{i\}} && \text{because } \Phi \subseteq \text{LCut}(T,j)
\end{aligned}$$

□

The rule (EF') has a very useful property, which will be used in the algorithm:

Property 1 *To achieve maximum propagation by the rule (EF') for a given activity $i \in T$, it is sufficient to look for an activity $j \in (T \setminus \{i\})$ such that (EF') holds and* lct_j *is maximum.*

Proof Let us consider an activity i and two different activities j_1 and j_2 for which the detection part of the rule (EF') holds. Moreover let $\mathrm{lct}_{j_1} \leqslant \mathrm{lct}_{j_2}$. Then $\mathrm{LCut}(T, j_1) \subseteq \mathrm{LCut}(T, j_2)$ and so $\mathrm{ect}_{\mathrm{LCut}(T,j_1)} \leqslant \mathrm{ect}_{\mathrm{LCut}(T,j_2)}$. Therefore j_2 yields a better propagation than j_1. □

2.4. Not-first/not-last

Not-First and Not-Last are two symmetric propagation algorithms for the unary resource. From these two, we will consider only the Not-Last algorithm.

The algorithm is based on the following rule. Let us consider a set $\Omega \subsetneq T$ 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:

$$\mathrm{est}_\Omega + \mathrm{p}_\Omega > \mathrm{lct}_i - \mathrm{p}_i \tag{11}$$

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:

$$\mathrm{lct}_i := \min\left\{\mathrm{lct}_i,\ \max\left\{\mathrm{lct}_j - \mathrm{p}_j,\ j \in \Omega\right\}\right\} \tag{12}$$

The full Not-Last rule is:

$$\forall \Omega \subsetneq T,\ \forall i \in (T \setminus \Omega):$$
$$\mathrm{est}_\Omega + \mathrm{p}_\Omega > \mathrm{lct}_i - \mathrm{p}_i \ \Rightarrow\ \mathrm{lct}_i := \min\left\{\mathrm{lct}_i,\ \max\left\{\mathrm{lct}_j - \mathrm{p}_j,\ j \in \Omega\right\}\right\} \quad \text{(NL)}$$

For a demonstration, see Figure 3.

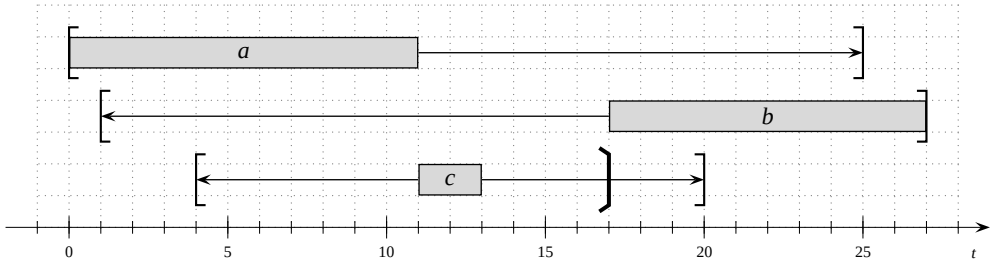

Figure 3. A sample problem for Not-Last: lct_C can be changed from 20 to 17.

The only algorithm which is based directly on this rule is by Baptiste and Le Pape [5]. The main difficulty of this algorithm is to find the set Ω which achieves the maximum filtering of lct_i for a given activity i.

Torres and Lopez [24] modified this rule in the following way. In order to achieve the fixpoint the algorithm must be iterated (see Chapter 1.3). Therefore it is not necessary to achieve the best filtering in the first iteration, the filtering can be further improved in the next runs of the algorithm.

Let the activity i be fixed. If there is a set Ω for which the rule (NL) propagates it must be a subset of:

$$\mathrm{NLSet}(T,i) = \left\{ j,\ j \in T \ \&\ \mathrm{lct}_j - \mathrm{p}_j < \mathrm{lct}_i \ \&\ j \neq i \right\} \tag{13}$$

Because otherwise $\max\left\{\mathrm{lct}_j - \mathrm{p}_j,\ j \in \Omega\right\} \geqslant \mathrm{lct}_i$. The question is whether there is such a set $\Omega \subseteq \mathrm{NLSet}(T,i)$ for which also the detection part of the rule (NL) holds, i.e., $\mathrm{est}_\Omega + \mathrm{p}_\Omega > \mathrm{lct}_i - \mathrm{p}_i$. It exists if and only if:

$$\max\left\{\mathrm{est}_\Omega + \mathrm{p}_\Omega,\ \Omega \subseteq \mathrm{NLSet}(T,i)\right\} = \mathrm{ect}_{\mathrm{NLSet}(T,i)} > \mathrm{lct}_i - \mathrm{p}_i$$

It is not necessary to search for the actual set Ω. By the definition of the set $\mathrm{NLSet}(T,i)$ it holds:

$$\max\left\{\mathrm{lct}_j - \mathrm{p}_j,\ j \in \Omega\right\} \leqslant \max\left\{\mathrm{lct}_j - \mathrm{p}_j,\ j \in \mathrm{NLSet}(T,i)\right\} < \mathrm{lct}_i$$

Thus the value lct_i can be updated to $\max\left\{\mathrm{lct}_j - \mathrm{p}_j,\ j \in \mathrm{NLSet}(T,i)\right\}$. And if it can be updated better it will be done in the next runs of the algorithm.

The whole modified not-last rule is:

$$\forall i \in T: \quad \mathrm{ect}_{\mathrm{NLSet}(T,i)} > \mathrm{lct}_i - \mathrm{p}_i \quad \Rightarrow \quad \mathrm{lct}_i := \max\left\{\mathrm{lct}_j - \mathrm{p}_j,\ j \in \mathrm{NLSet}(T,i)\right\} \quad \text{(NL')}$$

Let us demonstrate the difference between rules (NL) and (NL') on sample problem on Figure 4. Rule (NL) immediately updates lct_e using $\Omega = \{b,c\}$ to $\max\{\mathrm{lct}_b - \mathrm{p}_b,\ \mathrm{lct}_c - \mathrm{p}_c\} = 15$. However $\mathrm{NLSet}(e) = \{b,c,d\}$ and therefore rule (NL') updates lct_e to $\max\{\mathrm{lct}_b - \mathrm{p}_b,\ \mathrm{lct}_c - \mathrm{p}_c,\ \mathrm{lct}_d - \mathrm{p}_d\} = 19$. But during the following application of rule (NL') the activity d is no longer in $\mathrm{NLSet}(e)$ (because of the changed $\mathrm{lct}_e = 19$) and therefore lct_e is finally updated to 15.

Proposition 3 *Considering an activity i, at most $n-1$ iterative applications of the rule (NL') achieve the same filtering as one application of the rule (NL).*

Proof Let Ω be the set which induces the maximum change of the value lct_i by the rule (NL). Until the same value of lct_i is reached, in each iteration of the rule (NL') holds that $\Omega \subseteq \mathrm{NLSet}(T,i)$. The reason follows. Because the rule (NL) propagates for the set Ω it must be that $\max\{\mathrm{lct}_j - \mathrm{p}_j,\ j \in \Omega\} < \mathrm{lct}_i$. Thus:

$$\forall j \in \Omega: \quad \mathrm{lct}_j - \mathrm{p}_j < \mathrm{lct}_i$$

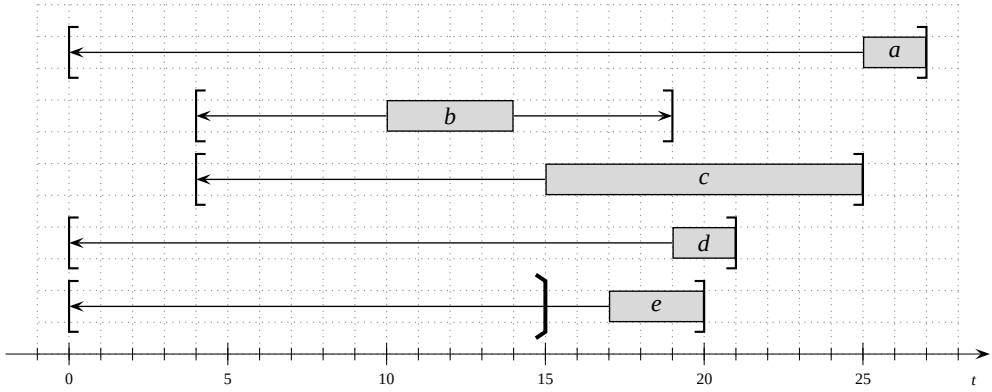

Figure 4. The difference between rules (NL) and (NL'). Rule (NL) immediately updates lct_e to 15 while rule (NL') updates lct_e first to to 19 and only then to 15.

And because $i \notin \Omega$ we get that $\Omega \subseteq \mathrm{NLSet}(T,i)$. Thus:

$$\mathrm{ect}_{\mathrm{NLSet}(T,i)} \geqslant \mathrm{est}_\Omega + \mathrm{p}_\Omega > \mathrm{lct}_i - \mathrm{p}_i$$

and the rule (NL') holds and propagates.

After each successful application of the rule (NL') the value lct_i is decreased. This removes at least one activity from the set $\mathrm{NLSet}(T,i)$. Therefore the final value of lct_i must be reached after at most $n-1$ iterations and it is the same as for the rule (NL). □

Note that the rule (NL') does not have to be iterated for each activity i separately and the iterations can be even mixed with other algorithms. Because both the rules (NL) and (NL') are monotonic, the resulting fixpoint will be the same. However the number of iterations can differ, the maximum number of $n-1$ iterations is not guaranteed in this case.

2.5. Detectable precedences

The idea of detectable precedences was introduced in [25] for a batch resource with sequence dependent setup times and then simplified for the unary resource in [26]. Figure 5 provides an example when neither Edge Finding nor Not-First/Not-Last algorithm is able to change any bounds.

Not-First 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 change of any bound. However, from the knowledge $\{A,B\} \ll C$ we can deduce that $\mathrm{est}_C \geqslant \mathrm{est}_A + \mathrm{p}_A + \mathrm{p}_B = 10$. This is exactly what the Detectable Precedences algorithm does.

Definition 1 *Let i and j be two different activities from the same resource. A precedence $j \ll i$ is called* detectable, *if it can be 'discovered' only by comparing bounds of the two*

activities:

$$\mathrm{est}_i + \mathrm{p}_i > \mathrm{lct}_j - \mathrm{p}_j \quad \Rightarrow \quad j \ll i \tag{14}$$

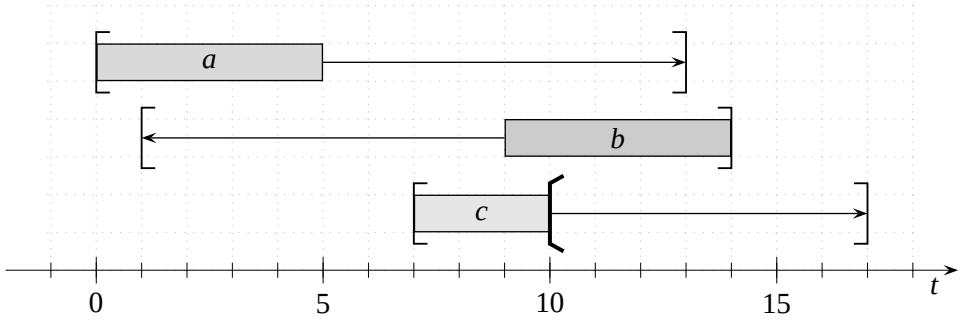

Figure 5. A sample problem for Detectable Precedences. est_C can be changed from 7 to 10.

Notice that in Figure 5 both precedences $A \ll C$ and $B \ll C$ are detectable.

The propagation rule follows. Let us define a set of all 'detectable predecessors' of the activity i as a set:

$$\mathrm{DPrec}(T,i) = \left\{ j,\ j \in T \ \&\ \mathrm{est}_i + \mathrm{p}_i > \mathrm{lct}_j - \mathrm{p}_j \ \&\ j \neq i \right\} \tag{15}$$

Because $\mathrm{DPrec}(T,i) \ll i$, we can adjust the earliest starting time of the activity i (notice similarity with the Edge Finding rule (4)):

$$\forall i \in T: \quad \mathrm{est}_i := \max\left\{ \mathrm{est}_i,\ \mathrm{ect}_{\mathrm{DPrec}(T,i)} \right\} \tag{DP}$$

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

2.6. Precedence energy

Filtering power of the previous rule can be further increased if we consider all types of precedences and not only the detectable ones. In particular:

1. Precedences coming from the original problem itself.
2. Precedences added later during the search as search decisions.
3. Detectable precedences.
4. Precedences discovered by Edge Finding, see rule (EF).
5. New precedences can be also detected by combining some precedences we already know. For example if $a \ll b$ is a search decision and $b \ll c$ is a detectable precedence then $a \ll c$ is also a valid precedence. This is what we call a transitive closure of precedences (which will be defined later).

Taking all these precedences into account we can create so called precedence graph and define a set of all predecessors of an activity i on a resource as:

$$\mathrm{Prec}(i) = \{j,\ j \in T \ \&\ j \ll i\} \tag{16}$$

where the precedence $j \ll i$ can be of arbitrary type.

The propagation rule is (notice the similarity with the rules (4) and (DP)):

$$\forall i \in T: \quad \mathrm{est}_i := \max\left\{\mathrm{est}_i,\ \mathrm{ect}_{\mathrm{Prec}(i)}\right\} \tag{PE}$$

There is also a symmetric version for adjustment of lct_i.

The main difficulty is to find all these precedences, i.e., construct the precedence graph. After that propagation according to this rule is quite easy, the algorithm is presented in Chapter 3.1.

In the rest of this chapter we will present several propositions which help us to build the set $\mathrm{Prec}(i)$ more easily.

2.6.1. Detectable precedences

A useful property of detectable precedences is that once a precedence is detectable it stays detectable:

Proposition 4 *Let $j \ll i$ be a detectable precedence. Then it stays detectable after any additional propagation and/or any addition search decisions.*

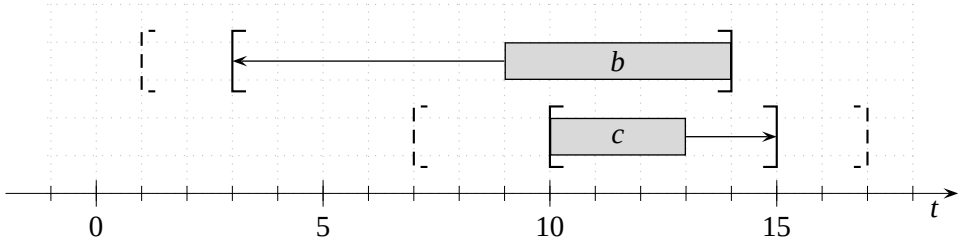

Figure 6. Detectable precedence $b \ll c$ from Figure 5 after some propagation: est_b has changed from 1 to 3, est_c from 7 to 10 and lct_c from 17 to 15. The precedence $b \ll c$ is still detectable.

Proof For a demonstration see Figure 6.

Let est_i and lct_j be bounds of activities i and j in the time when $j \ll i$ is detectable. And let est'_i and lct'_j be values of the same bounds after some propagation and/or search decisions[2].

The precedence $j \ll i$ was detectable therefore:

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

[2] Values p_i and p_j cannot change, we assume they are constant.

Propagation and search decisions can only increase the value est_i and decrease the value lct_j therefore it holds:

$$\text{est}'_i \geqslant \text{est}_i$$
$$\text{lct}'_i \leqslant \text{lct}_i$$

Thus:

$$\text{est}'_i + \text{p}_i > \text{lct}'_j - \text{p}_j$$

Therefore the precedence $j \ll i$ is still detectable. □

2.6.2. Precedences from edge finding

Proposition 5 *When Edge Finding is unable to find any further bound adjustment then all precedences which Edge Finding found are detectable.*

Proof 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 $\Omega \ll i$ therefore the triggering condition in the rule (EF) was valid before the filtering:

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

Since that, the bounds of all activities could have changed: est could have been increased and lct could have been decreased. However these changes cannot invalidate this condition, therefore it has to be still valid. And so:

$$\text{est}_\Omega > \text{lct}_\Omega - \text{p}_\Omega - \text{p}_i \tag{17}$$

Edge Finding is unable to further change any bound. According to the rule (EF) it means that:

$$\text{est}_i \geqslant \max\{\text{est}_{\Omega'} + \text{p}_{\Omega'},\ \Omega' \subseteq \Omega\}$$
$$\text{est}_i \geqslant \text{est}_\Omega + \text{p}_\Omega$$

In this inequality, est_Ω can be replaced by the right side of the inequality (17):

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

$\text{lct}_\Omega \geqslant \text{lct}_j$ because $j \in \Omega$. Using this we get:

$$\text{est}_i > \text{lct}_j - \text{p}_i$$
$$\text{est}_i + \text{p}_i > \text{lct}_j - \text{p}_j$$

So the condition (14) holds and the precedence $j \ll i$ is detectable.

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

So when we are looking for all precedences on a resource, we do not have to remember all precedences found by Edge Finding because they become detectable anyway. The item 4 from the page 173 is only a subset of the item 3.

2.6.3. Propagated precedences

Before we continue, let us define a *propagated* precedence. Once we know about some precedence, we propagate it. Depending on the type of the precedence we use different algorithms: binary precedence constraint, Detectable Precedences, Edge Finding or Precedence Energy. In all these cases the adjustment is done according to the same idea (see rules(DP), (EF), (PE) and Figure 7):

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

This allows to define a propagated precedence as follows.

Definition 2 *Let i and j be two different activities on the same resource and let $j \ll i$. The precedence $j \ll i$ is called* propagated *iff the activities i and j fulfill the following two inequalities:*

$$\mathrm{est}_i \geqslant \mathrm{est}_j + \mathrm{p}_j$$
$$\mathrm{lct}_j \leqslant \mathrm{lct}_i - \mathrm{p}_i$$

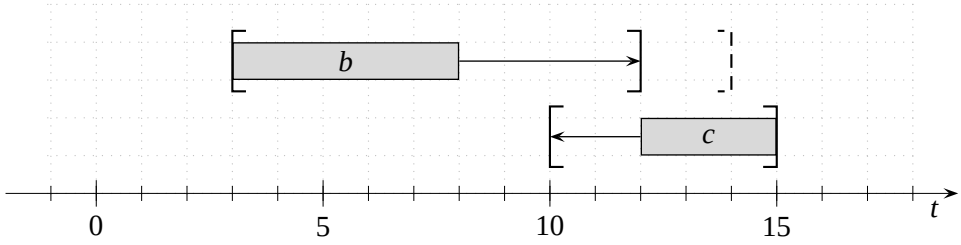

Figure 7. Detectable precedence $b \ll c$ from Figure 6 after being propagated (lct_b was changed from 14 to 12). I.e. $b \ll c$ is now both detectable and propagated (but note that not all propagated precedences must be detectable).

In the following we will assume that we do not miss a propagation of any known precedence and therefore all known precedences eventually become propagated.

2.6.4. Transitive closure of precedences

First let us define a transitive closure:

Definition 3 *Precedences on a resource forms* transitive closure *iff:*

$$\forall i, j, k \in T:\ i \ll j\ \&\ j \ll k\ \Rightarrow i \ll k$$

The idea is that we can achieve better filtering by taking into account the precedences resulting from the transitivity rule above, see for example [8] or [14]. In this section we will show how to compute the transitive closure more effectively.

Let us summarize the results of the previous paragraphs. We defined two types of precedences on an unary resource:

A. Detectable precedences. These are the most easy precedences to find. They originate from:
   (a) the Detectable Precedences algorithm,
   (b) the Edge Finding algorithm.

B. Non-detectable but propagated precedences. They originate from:
   (a) the original problem itself,
   (b) search decisions.

The question is: will be a precedence resulting from the transitivity rule detectable or not? As we will see in the following, in the most cases it will be detectable (and thus already known). This is the key to make computation of transitive closure more effective.

Lemma 1 *Let $a \ll b$, $b \ll c$ and one of these precedences is detectable and the other one is propagated. Then the precedence $a \ll c$ is detectable.*

Proof We distinguish two cases:

1. $a \ll b$ is detectable and $b \ll c$ is propagated.
   Because the precedence $b \ll c$ is propagated:
   $$\mathrm{est}_c \geqslant \mathrm{est}_b + \mathrm{p}_b$$
   and because the precedence $a \ll b$ is detectable:
   $$\begin{aligned} \mathrm{est}_b + \mathrm{p}_b &> \mathrm{lct}_a - \mathrm{p}_a \\ \mathrm{est}_c &> \mathrm{lct}_a - \mathrm{p}_a \end{aligned}$$
   Thus the precedence $a \ll c$ is detectable.

2. $a \ll b$ is propagated and $b \ll c$ is detectable.
   Because the precedence $a \ll b$ is propagated:
   $$\mathrm{lct}_b - \mathrm{p}_b \geqslant \mathrm{lct}_a$$
   And because the second precedence $b \ll c$ is detectable:
   $$\begin{aligned} \mathrm{est}_c + \mathrm{p}_c &> \mathrm{lct}_b - \mathrm{p}_b \\ \mathrm{est}_c + \mathrm{p}_c &> \mathrm{lct}_a \end{aligned}$$
   Once again, the precedence $a \ll c$ is detectable.

□

Now we are able to prove the following proposition:

Proposition 6 *Let $i_1, i_2, \ldots, i_n \in T$ and let $i_1 \ll i_2 \ll \cdots \ll i_n$ be precedences such that one of them ($i_k \ll i_{k+1}$) is detectable and all remaining precedences are propagated. Then the precedence $i_1 \ll i_n$ is detectable.*

Proof By induction on n. For $n = 1$ the proposition is trivially true, for $n = 2$ the proposition is proved by Lemma 1. Now lets suppose that the proposition is valid for n, we will prove that it is valid also for $n + 1$. For a demonstration see Figure 2.

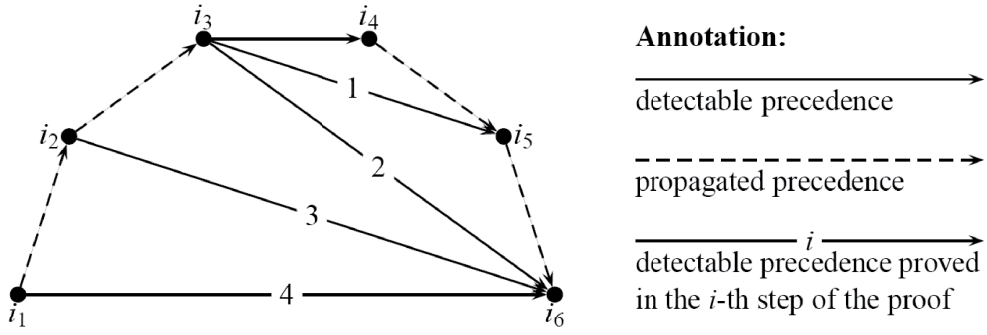

Figure 8. Computation of transitive closure.

According to Lemma 1:

- If $k \leqslant n-1$ then $i_k \ll i_{k+1} \ll i_{k+2}$ can be replaced by detectable precedence $i_k \ll i_{k+2}$,
- otherwise $i_{k-1} \ll i_k \ll i_{k+1}$ can be replaced by detectable precedence $i_{k-1} \ll i_{k+1}$.

In both cases resulting sequence of precedences has length only n and contains detectable precedence. Therefore (according to the induction assumption) $i_1 \ll i_n$ is detectable precedence. □

As mentioned earlier, all precedences eventually become propagated. Thus (using the last proposition) the transitive closure can be computed as a union of:

1. Detectable precedences.
2. transitive closure of non-detectable precedences.

The set of non-detectable precedences is mostly static – usually non-detectable precedences are introduced during a search only as search decisions. This strongly limits the number of times when the transitive closure must be recalculated.

3. Filtering algorithms

In this chapter we describe filtering algorithms for the propagation rules presented earlier. Most of the algorithms are based on the data structure called Θ-tree, which is described in a special chapter.

To make the description of the algorithms more comprehensible the algorithms are presented in a different order than the propagation rules.

3.1. Precedence energy

Propagation according to the precedence energy is the only presented algorithm for the unary resource constraint which does not use the Θ-trees, therefore it is presented first.

Algorithm 2 requires a transitive closure of non-detectable precedences and its extension whenever new non-detectable precedence (search decision) is added into the system (as was described in Section 2.6). Note that upon backtracking it is necessary to return the precedence graph into its previous state.

The idea of Algorithm 2 is as follows. For each activity i we compute the value $\mathrm{ect}_{\mathrm{Prec}(i)}$ and store it in the variable m. The test $j \ll i$ on the line 4 considers all types of precedences – detectable as well as non-detectable (including precedences from the transitive closure). The worst-case time complexity of the algorithm is $O(n^2)$.

Algorithm 2: Precedence Energy Based Filtering

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

A symmetric algorithm adjusts lct_i.

3.2. Θ-trees

One of the main complexities of the filtering algorithms for an unary resource is to quickly compute the earliest completion time ect_Θ of some set Θ. The following data structure can help us to quickly recompute the value ect_Θ whenever the set Θ is changed. The name Θ-tree comes from the fact that the represented set will be always named Θ.

The idea of Θ-trees was first introduced in [26] and then slightly modified in [28]. In the following, we will use this modified version of Θ-trees.

A Θ-tree is a balanced binary tree. Activities from the set Θ are represented by leaf nodes[3]. In the following we do not make a difference between an activity and the leaf node representing that activity. Internal nodes of the tree are used to hold some precomputed values. For an example of a Θ-tree see Figure 9.

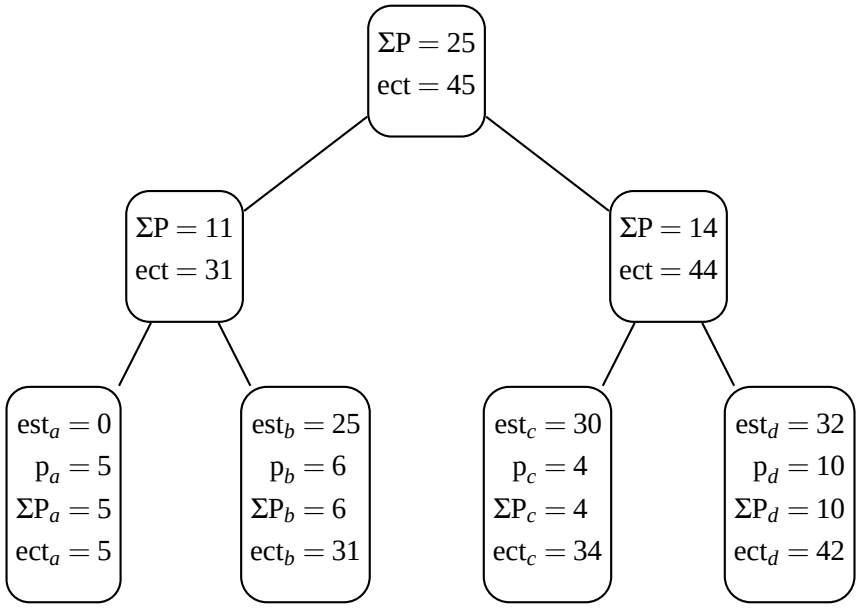

Figure 9. An example of a Θ-tree for $\Theta = \{a, b, c, d\}$.

Let v be an arbitrary node of a 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 $\mathrm{ect}_i = \mathrm{est}_i + \mathrm{p}_i$. And for the root node r we have $\mathrm{ect}_r = \mathrm{ect}_\Theta$.

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

$$\Sigma \mathrm{P}_v = \Sigma \mathrm{P}_{\mathrm{left}(v)} + \Sigma \mathrm{P}_{\mathrm{right}(v)} \tag{18}$$

[3]This is the main difference from [26]. The tree is deeper by one level, however a simpler computation of ect and ΣP compensates that.

In order to compute also ect_v recursively, the activities have to be stored in the leaves in the non-decreasing order by est from left to right. In particular for any two activities $i, j \in \Theta$, if $\text{est}_i < \text{est}_j$ then the activity i is stored on the left from the activity j. Due to this property the following inequality holds:

$$\forall i \in \text{Left}(v), \forall j \in \text{Right}(v): \quad \text{est}_i \leqslant \text{est}_j \tag{19}$$

where $\text{Left}(v)$ is a shortcut for $\text{Leaves}(\text{left}(v))$, similarly $\text{Right}(v)$.

Proposition 7 *For an internal node v, the value* ect_v *can be computed by the following recursive formula:*

$$\text{ect}_v = \max\left\{\text{ect}_{\text{right}(v)},\ \text{ect}_{\text{left}(v)} + \Sigma\text{P}_{\text{right}(v)}\right\} \tag{20}$$

Proof From the definition (1), the value ect_v is:

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

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

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

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

2. $\text{Left}(v) \cap \Theta' \neq \emptyset$. Then $\text{est}_{\Theta'} = \text{est}_{\Theta' \cap \text{Left}(v)}$ because of the property (19). Let S be the set of all possible Θ' considered in this part of the proof:

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

Then:

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

We used the fact that the maximum is achieved only by such a set Θ' for which $\text{Right}(v) \subsetneq \Theta'$ and therefore $\text{p}_{\Theta' \cap \text{Right}(v)} = \text{p}_{\text{Right}(v)}$. Also we used the fact that $\Theta' \cap \text{Left}(v)$ enumerates all possible subsets of $\text{Left}(v)$ and thus:

$$\max\left\{\text{est}_{\Theta' \cap \text{Left}(v)} + \text{p}_{\Theta' \cap \text{Left}(v)},\ \Theta' \subseteq S\right\} = \text{ect}_{\text{left}(v)}$$

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	$O(1)$

Table 1. Worst-case time complexities of operations on Θ-trees.

Combining the results of parts 1 and 2 together we see that the formula (20) is correct. □

Due to the recursive formulas (18) and (20), the values ect_v and ΣP_v can be easily computed within usual operations with a balanced binary tree without changing their time complexities. Time complexities of operations with Θ-trees are summarized in Table 1.

Notice that Θ-trees can be implemented as any type of a balanced binary tree. The only requirement is the time complexity $O(\log n)$ for inserting or deleting a leaf, and the time complexity $O(1)$ for finding the root node.

According to the author's experience, the most efficient implementation of Θ-trees is to make the shape of the tree fixed during the whole computation. I.e., instead of adding and removing leaves in the tree we just activate and deactivate them. A deactivated leaf representing the fact that an activity i is not in the set Θ has $\Sigma P_i = 0$ and $\mathrm{ect}_i = -\infty$. Clearly, these additional 'empty' leaves do not interfere with the formulas (18) and (20).

3.3. Overload checking

Let us recall the definition (3) of the set $\mathrm{LCut}(t, j)$:

$$\mathrm{LCut}(T, j) = \left\{k,\ k \in T \ \&\ \mathrm{lct}_k \leqslant \mathrm{lct}_j\right\}$$

The algorithm is based on the following observation:

Observation 1 *Let* $T = \{j_1, j_2, \ldots, j_n\}$ *and* $\mathrm{lct}_{j_1} \leqslant \cdots \leqslant \mathrm{lct}_{j_n}$. *Then* $\mathrm{LCut}(T, j_1) \subseteq \mathrm{LCut}(T, j_2) \subseteq \cdots \subseteq \mathrm{LCut}(T, j_n)$.

The idea is to get activities in non-decreasing order of lct_j and to compute $\Theta = \mathrm{LCut}(T, j)$. Due to the ordering of the activities the set $\mathrm{LCut}(T, j)$ can be quickly recomputed from the previous set.

Algirithm 3: Overload Checking

```
1  Θ := Ø;
2  for j ∈ T in non-decreasing order of lct_j do begin
3        Θ := Θ ∪ {j};
4        if ect_Θ > lct_j then
5              fail; {No solution exists}
6  end;
```

Notice that if there are two activities j_k and j_{k+1} such that $\mathrm{lct}_{j_k} = \mathrm{lct}_{j_{k+1}}$ then the set Θ is not really $\mathrm{LCut}(T, j_k)$ until the second activity j_{k+1} is also included into the set Θ. Nevertheless, this does not influence soundness of the algorithm.

The worst-case 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.

3.4. Detectable precedences

Let us recall the definition of the detectable predecessors of the activity i (15):

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

Unfortunately the sets $\mathrm{DPrec}(T,i)$ are not nested in each other in a similar way as the sets $\mathrm{LCut}(T,i)$. The problem is the condition $i \neq j$. Thus let us define the set $\mathrm{DPrec}'(T,i)$ as:

$$\mathrm{DPrec}'(T,i) = \left\{ j,\ j \in T \ \&\ \mathrm{est}_i + \mathrm{p}_i > \mathrm{lct}_j - \mathrm{p}_j \right\}$$

Clearly $\mathrm{DPrec}(T,i) = \mathrm{DPrec}'(T,i) \setminus \{i\}$. The sets $\mathrm{DPrec}'(T,i)$ are nested in the following way:

Observation 2 *Let* $T = \{i_1, i_2, \ldots, i_n\}$ *and* $\mathrm{est}_1 + \mathrm{p}_1 \leqslant \mathrm{est}_2 + \mathrm{p}_2 \leqslant \cdots \leqslant \mathrm{est}_n + \mathrm{p}_n$. *Then* $\mathrm{DPrec}'(T,i_1) \subseteq \mathrm{DPrec}'(T,i_2) \subseteq \cdots \subseteq \mathrm{DPrec}'(T,i_n)$.

Algorithm 4 is based on a incremental computation of the sets $\mathrm{DPrec}'(T,i)$. Initial sorts take $O(n \log n)$. 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 worst-case time complexity of the algorithm is $O(n \log n)$.

Algirithm 4: Detectable Precedences

```
1   Θ := Ø;
2   Q := queue of all activities j ∈ T in non-decreasing order of lct_j − p_j;
3   for i ∈ T in non-decreasing 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;
10  for i ∈ T do
11  est_i := est'_i;
```

3.4.1. Note about idempotency

We did not even try to make the algorithm idempotent since it is necessary to repeat all propagation algorithms until a fixpoint is reached (see Chapter 1.3). And we do the same in all following algorithms.

However it is possible to improve the algorithm in two ways:

a) The current algorithm does not change est_i immediately, it stores the new bound in est'_i and apply the change at the end. But it would be possible to change est_i immediately and rebalance the Θ-tree (if $i \in \Theta$). This would not lead to an idempotent algorithm, but it can save some iterations for reaching a fixpoint.

b) Make the algorithm idempotent. Note that Detectable Precedences algorithm has two parts - the one which updates est_i (presented above) and the symmetrical one which updates lct_i. Even if we make both parts idempotent, together they will not be idempotent. Therefore it is necessary to search for fixpoint even with such algorithm.

I would like to consider these improvements in my future work.

3.5. Not-first/not-last

Let us recall the definition of the set $\mathrm{NLSet}(T,i)$:

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

Again the sets $\mathrm{NLSet}(T,i)$ are not nested in each other because of the condition $j \neq i$. Let us define:

$$\mathrm{NLSet}'(T,i) = \left\{ j,\ j \in T \ \&\ \mathrm{lct}_j - \mathrm{p}_j < \mathrm{lct}_i \right\}$$

Hence $\mathrm{NLSet}(T,i) = \mathrm{NLSet}'(T,i) \setminus \{i\}$. The sets $\mathrm{NLSet}'(T,i)$ are now nested in the following way:

Observation 3 *Let* $T = \{i_1, i_2, \ldots, i_n\}$ *and* $\mathrm{lct}_1 \leqslant \mathrm{lct}_2 \leqslant \cdots \leqslant \mathrm{lct}_n$. *Then:*

$$\mathrm{NLSet}'(T,i_1) \subseteq \mathrm{NLSet}'(T,i_2) \subseteq \cdots \subseteq \mathrm{NLSet}'(T,i_n)$$

The idea of the Algorithm 5 is to compute the sets $\mathrm{NLSet}'(T,i)$ incrementally in the variable Θ. Lines 9–11 are repeated n times maximum because each time an activity is removed from the queue. The check on the line 13 can be done in $O(\log n)$. Therefore the worst-case time complexity of the whole algorithm is $O(n \log n)$.

Algorithm 5: Not-Last

```
1   for i ∈ T do
2       lct'_i := lct_i;
3   Θ := Ø;
4   Q := queue of all activities j ∈ T in non-decreasing order of lct_j − p_j;
5   for i ∈ T in non-decreasing order of lct_i do begin
6       while lct_i > lct_Q.first − p_Q.first do begin
9           j := Q.first;
10          Θ := Θ ∪ {j};
11          Q.dequeue;
12      end;
13      if ect_Θ\{i} > lct_i − p_i then
14          lct'_i := min{lct_j − p_j, lct'_i};
15  end;
16  for i ∈ T do
17      lct_i := lct'_i;
```

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 ect_Θ > lct_Q.first − p_Q.first then
8               lct'_Q.first := lct_j − p_j;
```

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

3.6. Θ-Λ-trees

Before we continue with the Edge Finding algorithm, let us introduce an extension of the Θ-tree data structure called a Θ-Λ-tree. The extension is motivated by the alternative Edge Finding rule (EF'):

$$\forall j \in T,\ \forall i \in T \setminus \mathrm{LCut}(T,j):\quad \mathrm{ect}_{\mathrm{LCut}(T,j)\cup\{i\}} > \mathrm{lct}_j \ \Rightarrow\ \mathrm{LCut}(T,j) \ll i$$

Suppose that we have chosen one particular activity j, constructed the set $\Theta = \mathrm{LCut}(T,j)$ and now we want the check this rule for each applicable activity i.

Unfortunately this would be too slow using the standard Θ-trees. For each activity i we would have to:

1. add the activity i into the set Θ,
2. check whether $\mathrm{ect}_\Theta > \mathrm{lct}_j$,
3. remove the activity i from the set Θ.

Time complexity of this procedure is $O(\log n)$ for each activity i.

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 *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 precisely: let $\Lambda \subseteq T$ be a set of all gray activities, $\Lambda \cap \Theta = \emptyset$. The purpose of the Θ-Λ-tree data structure is to compute the following value:

$$\overline{\mathrm{ect}}(\Theta,\Lambda) = \max\left(\{\mathrm{ect}_\Theta\} \cup \left\{\mathrm{ect}_{\Theta\cup\{i\}},\ i \in \Lambda\right\}\right) \tag{21}$$

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 the following two values are added into each node of the tree:

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

$\overline{\Sigma\mathrm{P}}$ is the maximum processing time of the activities in a subtree if one gray activity can be used. Similarly $\overline{\mathrm{ect}}$ is the earliest completion time of a subtree with at most one gray activity included. For example of a Θ-Λ-tree see Figure 10.

The idea how to compute values $\overline{\Sigma\mathrm{P}}_v$ and $\overline{\mathrm{ect}}_v$ for an internal node v follows. In both cases 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 $\overline{\Sigma\mathrm{P}}_v$ (or we say the gray activity responsible for

$\overline{\Sigma \mathrm{P}}_v$) can be different from the gray activity used for (responsible for) $\overline{\mathrm{ect}}_v$. The following formulas are modifications of (18) and (20) to handle gray nodes:

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

$$\overline{\mathrm{ect}}_v = \max\left\{\overline{\mathrm{ect}}_{\mathrm{right}(v)}, \right. \quad \text{(a)}$$

$$\left. \mathrm{ect}_{\mathrm{left}(v)} + \overline{\Sigma \mathrm{P}}_{\mathrm{right}(v)},\ \overline{\mathrm{ect}}_{\mathrm{left}(v)} + \Sigma \mathrm{P}_{\mathrm{right}(v)}\right\} \quad \text{(b)}$$

Line (a) considers all sets Θ' such that $\Theta' \cap \mathrm{Left}(v) = \emptyset$ (see (1) of ect on page 165). Line (b) considers all sets Θ' such that $\Theta' \cap \mathrm{Left}(v) \neq \emptyset$.

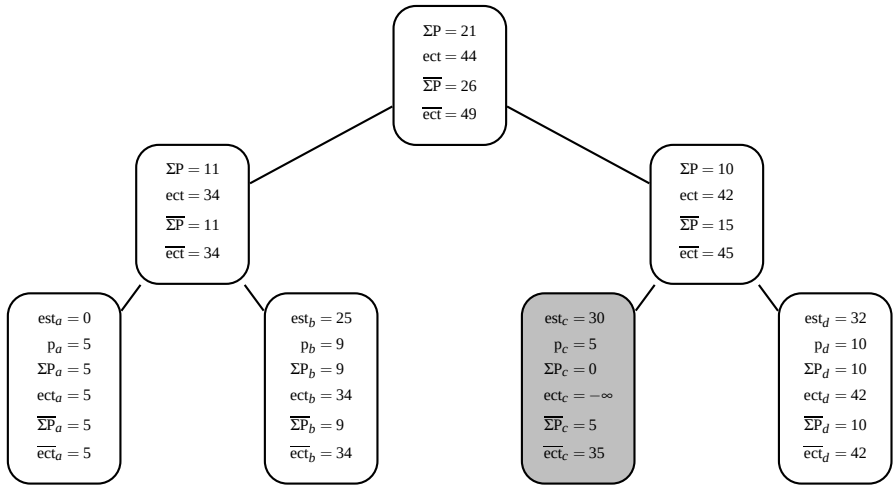

Figure 10. An example of a Θ-Λ-tree for $\Theta = \{a, b, d\}$ and $\Lambda = \{c\}$.

For each node v we can also compute the gray activity which is *responsible* for $\overline{\mathrm{ect}}_v$ or $\overline{\Sigma \mathrm{P}}_v$. If v is a leaf node (an activity i) then:

$$\mathrm{responsible}_{\overline{\Sigma \mathrm{P}}}(i) = \begin{cases} i & \text{if } i \text{ is gray,} \\ \text{undef} & \text{otherwise.} \end{cases}$$

$$\mathrm{responsible}_{\overline{\mathrm{ect}}}(i) = \begin{cases} i & \text{if } i \text{ is gray,} \\ \text{undef} & \text{otherwise.} \end{cases}$$

And for an internal node v:

$$\mathrm{responsible}_{\overline{\Sigma \mathrm{P}}}(i) = \begin{cases} \mathrm{responsible}_{\overline{\Sigma \mathrm{P}}}(\mathrm{left}(v)) & \text{if } \overline{\Sigma \mathrm{P}}(v) = \overline{\Sigma \mathrm{P}}_{\mathrm{left}(v)} + \Sigma \mathrm{P}_{\mathrm{right}(v)} \\ \mathrm{responsible}_{\overline{\Sigma \mathrm{P}}}(\mathrm{right}(v)) & \text{if } \overline{\Sigma \mathrm{P}}(v) = \Sigma \mathrm{P}_{\mathrm{left}(v)} + \overline{\Sigma \mathrm{P}}_{\mathrm{right}(v)} \end{cases}$$

Operation	Time Complexity
$(\Theta,\ \Lambda) := (\emptyset,\ \emptyset)$	$\mathcal{O}(1)$
$(\Theta,\ \Lambda) := (T,\ \emptyset)$	$\mathcal{O}(n \log n)$
$(\Theta,\ \Lambda) := (\Theta \setminus \{i\},\ \Lambda \cup \{i\})$	$\mathcal{O}(\log n)$
$\Theta := \Theta \cup \{i\}$	$\mathcal{O}(\log n)$
$\Lambda := \Lambda \cup \{i\}$	$\mathcal{O}(\log n)$
$\Lambda := \Lambda \setminus \{i\}$	$\mathcal{O}(\log n)$
$\overline{\text{ect}}(\Theta, \Lambda)$	$\mathcal{O}(1)$
ect_Θ	$\mathcal{O}(1)$

Table 2. Worst-case time complexities of operations on Θ-Λ-trees.

$$\text{responsible}_{\overline{\text{ect}}}(v) = \begin{cases} \text{responsible}_{\overline{\text{ect}}}(\text{right}(v)) & \text{if } \overline{\text{ect}}(v) = \overline{\text{ect}}_{\text{right}(v)} \\ \text{responsible}_{\overline{\Sigma\text{P}}}(\text{right}(v)) & \text{if } \overline{\text{ect}}(v) = \text{ect}_{\text{left}(v)} + \overline{\Sigma\text{P}}_{\text{right}(v)} \\ \text{responsible}_{\overline{\text{ect}}}(\text{left}(v)) & \text{if } \overline{\text{ect}}(v) = \overline{\text{ect}}_{\text{left}(v)} + \Sigma\text{P}_{\text{right}(v)} \end{cases}$$

Due to these recursive formulas all these computations can be done with usual operations on balanced binary trees without changing their time complexities. Table 2 shows the time complexities of selected operations on Θ-Λ-trees.

3.7. Edge finding

By Observation 1 (page 182) we known that the sets $\text{LCut}(T, j)$ are nested and by Property 1 (page 170) we know that for each activity i we are looking for the biggest $\text{LCut}(T, j)$ such that the rule (EF') holds.

Let j_n be the activity with the greatest lct_j from the set T. The Algorithm 6 starts with $\Theta = \text{LCut}(T, j_n) = T$ and $\Lambda = \emptyset$. Activities are sequentially (in non-increasing order by lct_j) moved from the set Θ into the set Λ, i.e., white nodes are discolored to gray. This way Θ is always some $\text{LCut}(T, j)$ (with an exception when there are two activities with the same lct) and the set Λ is the set of all activities i for which the rule (EF') was not applied yet. As soon as $\overline{\text{ect}}(\Theta,\ \Lambda) > \text{lct}_\Theta$, a responsible gray activity i is updated. Due to Property 1 the activity i cannot be updated better, therefore it can be removed from the set Λ.

Algorithm 6: Edge Finding

```
1   (Θ, Λ) := (T, Ø);
2   Q := queue of all activities j ∈ T in non-increasing order of lct_j;
3   j := Q.first;
4   while Q.size > 1 do begin
5       if ect_Θ > lct_j then
6           fail; {Resource is overloaded}
7       (Θ, Λ) := (Θ \ {j}, Λ ∪ {j});
8       Q.dequeue;
9       j := Q.first;
10      while ect‾(Θ, Λ) > lct_j do begin
11          i := gray activity responsible for ect‾(Θ, Λ);
12          est_i := max{est_i, ect_Θ};
13          Λ := Λ \ {i};
14      end;
15  end;
```

Note that at line 11 there has to be some gray activity responsible for $\overline{\text{ect}}(\Theta, \Lambda)$ because otherwise we would end up by fail on line 11. Lines 5 and 6 make the same check as the Overload Checking algorithm therefore they are not mandatory and can be removed.

During the entire run of the algorithm, the maximum number of iterations of the inner while loop (lines 10–14) is n, because each iteration removes an activity from the set Λ. Similarly, the number of iterations of the outer loop is n, because each time an activity is removed from the queue Q. According to Table 2 the worst time complexity of each single line within the loops is $O(\log n)$. Therefore the worst-case 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 $\overline{\text{ect}}_k = \text{ect}_k$ and $\overline{\Sigma\text{P}}_k = \Sigma\text{P}_k$ for each node k. Hence we can save some time by building the initial Θ-Λ-tree as a 'normal' Θ-tree.

4. Optional activities

4.1. Motivation

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 in the final schedule or not. Therefore we will call such activities *optional*.

For an optional activity, we would like to speculate what would happen if the activity is processed by the resource. This chapter presents strong filtering algorithms for an unary resource with optional activities.

4.2. History

Traditionally, resource constraints are not designed to handle optional activities properly, and there are only limited ways how to model them. The following paragraphs describes such models.

Dummy activities It is a workaround for constraint solvers which do not allow to add more activities on the resource during a search (i.e., the resource constraint is not dynamic [6]). 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 the minimal possible processing time instead of the 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., the minimal processing time is no longer zero). The main disadvantage of this approach is that the impossibility to use a particular alternative cannot be detected 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 temporarily set as regular activity. When a fail is found then the optional activity is rejected. Otherwise bounds of the optional activity can be adjusted. [7] introduces so called PEX-Edge Finding with time complexity $O(n^3)$. Any existing filtering algorithm for regular unary resource constraint can be extended this way by the cost of increasing its time complexity by factor n.

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 among machines. There is also a special filtering algorithm [29] designed to handle this type of alternatives.

4.3. New approach

The approach described in this chapter was published in [27, 28]. The idea is to modify existing filtering algorithms to treat optional activities differently: regular activities influence all other activities on the resource but optional activities do not. This way we try to achieve the same filtering as was described in the paragraph 'filtering of options' without increasing the time complexity by factor n. This chapter describes the following

filtering algorithms for unary resource with optional activities: Overload Checking, Detectable Precedences and Not-First/Not-Last. For Edge-Finding with optional activities see [16].

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

To make the notation concerning optional activities simpler, let R be the set of all regular activities and O the set of all optional activities, $T = R \cup O$, $R \cap O = \emptyset$.

For optional activities requiring a resource, we would like to consider the following questions:

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? Recalling the rule (OL) the resource is overloaded if there is such a set $\Omega \subseteq R$ that:

   $$\text{est}_\Omega + \text{p}_\Omega > \text{lct}_\Omega$$

   Certainly, if a resource is overloaded then the problem has no solution. Hence if an addition of an optional activity i results in an overloading then we can conclude that $\text{exists}_i = \text{false}$.

2. If the addition of an optional activity i does not result in an overloading, what is the earliest possible start time and the latest completion time of the activity i with respect to the regular activities on the resource? We would like to apply usual filtering algorithms for the activity i, however the activity i cannot cause any change on regular activities.

3. If an optional activity i would become a regular activity, will the first run of a filtering algorithm result in a fail? For example, with i turned into regular activity, Detectable Precedences can increase est_k of another (regular) activity k so much that $\text{est}_k + \text{p}_k > \text{lct}_k$. In that case we can also propagate $\text{exists}_i = \text{false}$.

In the following we will try to extend the algorithms from previous chapters by taking the tree items above into account. Note that it would be possible to go even further. For example, we could take into account two optional activities at the same time and try to reason about what happens if both of them become regular. However, to my best knowledge, there is currently no such algorithm.

4.4. Overload checking with optional activities

This section presents a modified Overload Checking algorithm which can handle optional activities. The original overload rule (OL) remains valid, however we must consider only regular activities R:

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

Now let us take into account an optional activity $o \in O$. If a change this optional activity into regular activity would result in an overloading then the activity can never be processed by the resource:

$$\forall \Omega \subseteq R,\ o \in O: \quad \left(\mathrm{lct}_{\Omega\cup\{o\}} - \mathrm{est}_{\Omega\cup\{o\}} < \mathrm{p}_{\Omega\cup\{o\}} \quad \Rightarrow \quad \mathrm{exists}_o := \mathrm{false}\right) \qquad (\mathrm{OL}_{oe})$$

Proposition 1 (page 166) shows that these two rules are equivalent to:

$$\forall j \in R: \quad \left(\mathrm{ect}_{\mathrm{LCut}(R,j)} > \mathrm{lct}_j \quad \Rightarrow \quad \mathrm{fail}\right) \qquad (\mathrm{OL}'_o)$$

$$\forall j \in R \cup \{o\}: \quad \left(\mathrm{ect}_{\mathrm{LCut}(R\cup\{o\},j)} > \mathrm{lct}_j \quad \Rightarrow \quad \mathrm{exists}_o := \mathrm{false}\right) \qquad (\mathrm{OL}'_{oe})$$

where $\mathrm{LCut}(R, j)$ is (recall the definition (3) from the page 166):

$$\mathrm{LCut}(R, j) = \left\{k,\ k \in R \,\&\, \mathrm{lct}_k \leqslant \mathrm{lct}_j\right\}$$

Let us assume that the activity $j \in T$ is fixed and we want to find all activities o for which the rule (OL'_{oe}) propagates. Clearly $\mathrm{lct}_o \leqslant \mathrm{lct}_j$, otherwise $o \notin \mathrm{LCut}(R\cup\{o\}, j)$ and if the rule (OL'_{oe}) holds than the resource is overloaded even without optional activity o. The set of all applicable activities o is therefore:

$$\mathrm{LCut}(O, j) = \left\{o,\ o \in O \,\&\, \mathrm{lct}_o \leqslant \mathrm{lct}_j\right\}$$

With all these possible activities o what is the maximum value of $\mathrm{ect}_{\mathrm{LCut}(R\cup\{o\},j)}$? Recall definition (21) from page 186:

$$\max\left\{\mathrm{ect}_{\mathrm{LCut}(R\cup\{o\},j)},\ o \in \mathrm{LCut}\,(O, j)\right\} = \overline{\mathrm{ect}}\,(\mathrm{LCut}\,(R, j)\,,\ \mathrm{LCut}\,(O, j))$$

Thus the rule (OL'_{oe}) is applicable if and only if $\overline{\mathrm{ect}}(\mathrm{LCut}(R, j), \mathrm{LCut}(O, j)) > \mathrm{lct}_j$. In this case the activity responsible for $\overline{\mathrm{ect}}(\mathrm{LCut}(R, j), \mathrm{LCut}(O, j))$ can be excluded from the resource.

Algorithm 7 detects overloading, it also deletes all optional activities o such that an addition of this activity o alone causes an overload. Of course, a combination of several optional activities may still cause an overload. The algorithm is based on the idea above and on Observation 1 about nesting of the sets LCut.

Algorithm 7: Overload Checking with Optional Activities

```
1   (Θ, Λ) := (Ø, Ø);
2   for i ∈ T in non-decreasing order of lct_i do begin
3       if i is an optional activity then
4           Λ := Λ ∪ {i};
5       else begin
6           Θ := Θ ∪ {i};
7           if ect_Θ > lct_i then
8               fail; {No solution exists}
9       end;
10      while ect‾(Θ, Λ) > lct_i do begin
11          o := optional activity responsible for ect‾(Θ, Λ);
12          exists_o := false;
13          Λ := Λ \ {o};
14      end;
15  end;
```

Note that it is possible that at line 11 $o = i$ holds. In this case at the next run of the while cycle the value $\overline{\text{ect}}(\Theta, \Lambda)$ is compared with lct_i on the line 10 even thought i is no longer in Θ nor in Λ. However that is not important: in this case $\overline{\text{ect}}(\Theta, \Lambda) > \text{lct}_i$ can no longer hold because that would be already detected in the previous run of the outer for cycle.

The worst-case time complexity of the algorithm is $O(n \log n)$. 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, the time complexity of each single line is $O(\log n)$ at most (see Table 2).

4.5. Detectable precedences with optional activities

Let us recall the original propagation rule (DP) for detectable precedences from the page 173. The rule is designed only for regular activities therefore we have to replace the set T by R:

$$\forall i \in R: \quad \text{est}_i := \max\left\{\text{est}_i,\ \text{ect}_{\text{DPrec}(R,i)}\right\}$$

where

$$\text{DPrec}(R,i) = \left\{j,\ j \in R \ \&\ \text{est}_i + \text{p}_i > \text{lct}_j - \text{p}_j \ \&\ j \neq i\right\}$$

Regular activities also influence optional activities therefore we can extend the rule in the following way:

$$\forall i \in T: \quad \text{est}_i := \max\left\{\text{est}_i,\ \text{ect}_{\text{DPrec}(R,i)}\right\} \qquad (\text{DP}_o)$$

And a symmetric rule:

$$\forall i \in T: \quad \mathrm{lct}_i := \min\left\{\mathrm{lct}_i,\ \mathrm{lst}_{\mathrm{DSucc}(R,i)}\right\} \qquad (\overline{\mathrm{DP}}_o)$$

where

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

The second rule follows. Let $i \in R$ be a regular activity, $o \in O$ an optional activity and let $o \ll i$ be a detectable precedence. If o becomes a regular activity then est_i will be updated by the rule above to $\max\left\{\mathrm{est}_i,\ \mathrm{ect}_{\mathrm{DPrec}(R\cup\{o\},i)}\right\}$. However this change of est_i may be unfeasible (immediate fail) when:

$$\max\left\{\mathrm{est}_i,\ \mathrm{ect}_{\mathrm{DPrec}(R\cup\{o\},i)}\right\} + \mathrm{p}_i > \mathrm{lct}_i$$

In this case o cannot be processed by the resource and we can set exists_o to false. The full rule is:

$$\forall i \in R, \forall o \in O \text{ such that } o \ll i \text{ is detectable}: \quad \left(\mathrm{ect}_{\mathrm{DPrec}(R\cup\{o\},i)} + \mathrm{p}_i > \mathrm{lct}_i \quad \Rightarrow \quad \mathrm{exists}_o := \mathrm{false}\right) \quad (\mathrm{DP}_{oe})$$

The algorithm for the rules (DP_o) and (DP_{oe}) can be found in [27, 28]. However the rule (DP_{oe}) is not really necessary as we will see later in Proposition 8. But first let us prove the following lemma:

Lemma 2 *Let $i \in R$ be a regular activity, $o \in O$ be an optional activity and let $i \ll o$ and $o \ll i$ be detectable precedences. Then full propagation according to the rules (OL_{oe}) and (DP_o) sets* exists_o *to* false.

Proof Let us consider the moment when the rule (DP_o) finishes propagation of the precedence $i \ll o$. The precedence becomes propagated therefore:

$$\mathrm{est}_o \geqslant \mathrm{est}_i + \mathrm{p}_i$$

The precedence $o \ll i$ stays detectable even after propagation (see Proposition 4), therefore:

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

Combining these two inequalities we get:

$$\mathrm{est}_o > \mathrm{lct}_o - \mathrm{p}_o$$

Therefore the rule (OL_{oe}) for the set $\Omega = \emptyset$ sets exists_o to false. □

Proposition 8 *Full propagation using the rules (OL_o), (OL_{oe}), (DP_o) and ($\overline{DP}_o$) makes also all changes resulting from the rule (DP_{oe}).*

Proof Let us consider the state when all propagation using the rules (OL_o), (DP_o) and ($\overline{\mathrm{DP}}_o$) is done, i.e., the rule (OL_o) did not detect fail and the rules (DP_o) and ($\overline{\mathrm{DP}}_o$) are not able to find any change. In the following we prove that in this moment the rule (DP_{oe}) can be substituted by the rule (OL_{oe}). Note that by the Domain Reduction Theorem [2] the resulting fixpoint is independent of the sequence in which the (monotonic) propagation rules are executed.

Let $i \in R$ and $o \in O$ be such activities that the rule (DP_{oe}) sets exists_o to false. We will prove that the same pruning can achieved using the rule (OL_{oe}). Let Ψ be such a subset of $\mathrm{DPrec}(R \cup \{o\}, i)$ that:

$$\mathrm{est}_\Psi + \mathrm{p}_\Psi = \mathrm{ect}_{\mathrm{DPrec}(R \cup \{o\}, i)} \tag{22}$$

Note that due to the definition of ect such set Ψ must exists and $o \in \Psi$ because otherwise the rule (OL_o) would already end the propagation by fail.

We distinguish two cases: $\Psi = \{o\}$ and $\Psi \neq \{o\}$.

1. Case $\Psi = \{o\}$. The rule (DP_{oe}) sets exists_o to false, therefore:

$$\begin{aligned} \mathrm{ect}_{\mathrm{DPrec}(R \cup \{o\}, i)} + \mathrm{p}_i &> \mathrm{lct}_i \\ \mathrm{est}_o + \mathrm{p}_o + \mathrm{p}_i &> \mathrm{lct}_i \qquad \text{by (22) and } \Psi = \{o\} \end{aligned}$$

   Thus the precedence $i \ll o$ is detectable. The precedence $o \ll i$ is detectable too because $\{o\} = \Psi \subseteq \mathrm{DPrec}(R \cup \{o\}, i)$. Applying Lemma 2 the rule (OL_{oe}) sets exists_o to false.

2. Case $\Psi \neq \{o\}$. For all activities $j \in \Psi$ the precedences $j \ll i$ are detectable because $\Psi \subseteq \mathrm{DPrec}(R \cup \{o\}, i)$. By applications of the rules (DP_o) and ($\overline{\mathrm{DP}}_o$) these precedences become propagated (recall Definition 2) with the exception of the precedence $o \ll i$ which is propagated only on the activity o:

$$\forall j \in \Psi: \quad \mathrm{lct}_j \leqslant \mathrm{lct}_i - \mathrm{p}_i \tag{23}$$

$$\forall j \in \Psi \setminus \{o\}: \quad \mathrm{est}_i \geqslant \mathrm{est}_j + \mathrm{p}_j \tag{24}$$

   Therefore:

$$\mathrm{lct}_{\Psi \cup \{i\}} = \mathrm{lct}_i \tag{25}$$

$$\mathrm{est}_{\Psi \cup \{i\}} = \mathrm{est}_\Psi \tag{26}$$

   For (26) we use the fact that $\Psi \neq \{o\}$ therefore there is a regular activity $j \in \Psi$ for which $\mathrm{est}_i \geqslant \mathrm{est}_j + \mathrm{p}_j$ by (24).

   The rule (DP_{oe}) sets exists_o to false, therefore:

$$\mathrm{ect}_{\mathrm{DPrec}(R \cup \{o\}, i)} + \mathrm{p}_i > \mathrm{lct}_i$$

$$\begin{aligned} \mathrm{est}_\Psi + \mathrm{p}_\Psi + \mathrm{p}_i > \mathrm{lct}_i & \qquad \text{by (22)} \\ \mathrm{est}_{\Psi\cup\{i\}} + \mathrm{p}_{\Psi\cup\{i\}} > \mathrm{lct}_{\Psi\cup\{i\}} & \qquad \text{by (25) and (26)} \end{aligned}$$

Thus the rule (OL_{oe}) propagates for the activity o and the set $\Omega = (\Psi \cup \{i\}) \setminus \{o\}$ and sets exists_o to false.

□

Algorithm 8 is a slightly modified version of Algorithm 4 to handle also optional activities using the rule (DP_o). The worst-case time complexity of the algorithm remains the same: $O(n \log n)$.

Algorithm 8: Detectable Precedences with Optional Activities

```
1   Θ := Ø;
2   Q := queue of all regular activities j ∈ R in non-decreasing order of lct_j − p_j;
3   for i ∈ T in non-decreasing 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;
10  for i ∈ T do
11      est_i := est'_i;
```

4.6. Not-first/not-last with optional activities

Let us recall the rule Not-Last (NL):

$$\forall \Omega \subsetneq R,\ \forall i \in (R \setminus \Omega):$$
$$\mathrm{est}_\Omega + \mathrm{p}_\Omega > \mathrm{lct}_i - \mathrm{p}_i \ \Rightarrow\ \mathrm{lct}_i := \min\left\{\mathrm{lct}_i,\ \max\left\{\mathrm{lct}_j - \mathrm{p}_j,\ j \in \Omega\right\}\right\}$$

From this rule we will derive two propagation rules for optional activities:

1. Regular activities also influence optional activities:

$$\forall \Omega \subseteq R,\ \forall i \in (T \setminus \Omega):$$
$$\mathrm{est}_\Omega + \mathrm{p}_\Omega > \mathrm{lct}_i - \mathrm{p}_i \ \Rightarrow\ \mathrm{lct}_i := \min\left\{\mathrm{lct}_i,\ \max\left\{\mathrm{lct}_j - \mathrm{p}_j,\ j \in \Omega\right\}\right\} \quad (\mathrm{NL}_o)$$

2. If an optional activity o becomes a regular activity then the rule not-last may result in an immediate failure by changing lct_i below $\mathrm{est}_i + \mathrm{p}_i$. In this case the activity o cannot be processed by the resource at all:

$$\forall o \in O, \forall \Omega \subsetneq R \cup \{o\}, o \in \Omega, \forall i \in (R \setminus \Omega) :$$
$$\text{est}_\Omega + \text{p}_\Omega > \text{lct}_i - \text{p}_i \;\&\; \max\left\{\text{lct}_j - \text{p}_j,\ j \in \Omega\right\} < \text{est}_i + \text{p}_i$$
$$\Rightarrow \quad \text{exists}_o := \text{false} \quad (\text{NL}_{oe})$$

The following proposition proves that the propagation rule (NL_{oe}) is not necessary.

Proposition 9 *Full propagation according to the rules (OL_o), (OL_{oe}), (DP_o) and ($\overline{\text{DP}}_o$) makes also all changes resulting from the rule (NL_{oe}).*

Proof Let us consider the state when the rules (OL_o), (DP_o) and ($\overline{\text{DP}}_o$) are not able to propagate any more. In the following we will prove that in this state the rule (OL_{oe}) makes also all changes resulting from the rule (NL_{oe}).

Let the rule (NL_{oe}) propagates for $o \in O$, $\Omega \subsetneq R \cup \{o\}$ and $i \in R \setminus \Omega$. We distinguish two cases: $\Omega = \{o\}$ and $\Omega \neq \{o\}$.

1. Case $\Omega = \{o\}$. Because the rule (NL_{oe}) propagates we get:

$$\text{est}_o + \text{p}_o > \text{lct}_i - \text{p}_i$$
$$\text{lct}_o - \text{p}_o < \text{est}_i + \text{p}_i$$

   Thus there are detectable precedences $i \ll o$ and $o \ll i$ and by Lemma 2 the rule (OL_{oe}) sets exists_o to false.

2. Case $\Omega \neq \{o\}$. Because the rule (NL_{oe}) propagates we get:

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

   therefore:

$$\forall j \in \Omega : \quad \text{lct}_j - \text{p}_j < \text{est}_i + \text{p}_i$$

   Thus for all $j \in \Omega$ there is a detectable precedence $j \ll i$. These precedences $j \ll i$ are propagated using the rules (DP_o) and ($\overline{\text{DP}}_o$), with exception of the precedence $o \ll i$ which is propagated only to the activity o:

$$\forall j \in \Omega : \ \text{lct}_j \leqslant \text{lct}_i - \text{p}_i \tag{27}$$
$$\forall j \in \Omega \setminus \{o\} : \ \text{est}_i \geqslant \text{est}_j + \text{p}_j \tag{28}$$

   Therefore:

$$\text{lct}_{\Omega \cup \{i\}} = \text{lct}_i \tag{29}$$
$$\text{est}_{\Omega \cup \{i\}} = \text{est}_\Omega \tag{30}$$

   For (30) we used the fact that $\Omega \neq \{o\}$ thus there is a regular activity $j \in \Omega$ such that $\text{est}_i \geqslant \text{est}_j + \text{p}_j$ by (28).

The rule (NL_{oe}) propagates, therefore:

$$\begin{aligned} \mathrm{est}_{\Omega} + \mathrm{p}_{\Omega\cup\{i\}} &\geqslant \mathrm{lct}_i \\ \mathrm{est}_{\Omega\cup\{i\}} + \mathrm{p}_{\Omega\cup\{i\}} &\geqslant \mathrm{lct}_{\Omega\cup\{i\}} \qquad \text{by (30) and (29)} \end{aligned}$$

Therefore the rule (OL_{oe}) propagates and sets exists_o to false.

□

Let us return to the propagation rule (NL_o). We can modify this rule in a similar way as we modified the rule (NL) to (NL'):

$$\forall i \in T: \quad \mathrm{est}_{\mathrm{NLSet}(R,i)} + \mathrm{p}_{\mathrm{NLSet}(R,i)} > \mathrm{lct}_i - \mathrm{p}_i \quad \Rightarrow$$
$$\mathrm{lct}_i := \min\left\{\mathrm{lct}_i,\ \max\left\{\mathrm{lct}_j - \mathrm{p}_j,\ j \in \mathrm{NLSet}(R,i)\right\}\right\} \quad (\mathrm{NL}'_o)$$

where

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

Proposition 10 *Considering an activity $i \in T$, at most $n-1$ iterative applications of the rule (NL'_o) achieve the same filtering as one pass of the filtering according to the rule (NL_o).*

Proof Analogous to the proof of Proposition 3. □

The paper [28] presents a propagation algorithm according to the rule (NL'_o) and a weaker version of the rule (NL_{oe}). As we proved in Proposition 9 the rule (NL_{oe}) is not really necessary. Therefore we can use a more simple Algorithm 9 which propagates only according to the rule (NL_{oe}). It is a simple modification of Algorithm 5. The difference is that at line 4 we put only regular activities into the queue Q, however at line 5 we iterate over all activities in T including the optional ones. The worst-case time complexity remains to be $O(n \log n)$.

Algorithm 9: Not-Last with Optional Activities

```
1   for i ∈ T do
2       lct'_i := lct_i;
3   Θ := Ø;
4   Q := queue of all activities j ∈ R in non-decreasing order of lct_j − p_j;
5   for i ∈ T in non-decreasing order of lct_i do begin
6       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 := min{lct_j − p_j, lct'_i};
13  end;
14  for i ∈ T do
15      lct_i := lct'_i;
```

5. Conclusions

This article presents propagation algorithms for the unary resource constraint. The main advantages of these algorithms are their speed and propagation power. With the exception of precedence energy all of these algorithms has time complexity $O(n \log n)$ what makes them very fast and scalable. Furthermore they are able to adjust time bounds of optional activities even before it is decided whether they will be part of the schedule or not. This property is very important for dealing with alternative production rules, which are often part of the real-life problems. Variants[4] of the algorithms presented here are implemented and available in the ILOG CP Optimizer 2.0 [1].

Acknowledgments I would like to thank Roman Barták and Ondřej Čepek who helped me to design the presented algorithms. Also I would like to thank Philippe Laborie and Jerome Rôgerie, my colleagues at ILOG, for valuable discussions about the subject.

[4]The article does not describe implementation tricks and optimizations which does not influence worst case time complexities of the presented algorithms. These implementation details are considered to be technical know-how of ILOG.

References

[1] ILOG CP Optimizer.
URL http://www.ilog.fr/products/cpoptimizer/.

[2] K.R. APT: The essence of constraint propagation. *Theoretical Computer Science*, **221**(1-2), (1999), 179-210,
URL http://citeseer.ist.psu.edu/apt98essence.html.

[3] P. BAPTISTE: A theoretical and experimental study of resource constraint propagation. In *PhD thesis*. University of Compiegne, 1998.

[4] P. BAPTISTE, C. LE PAPE and W. NUIJTEN: Satisfiability tests and time-bound adjustments for cumulative scheduling problems. In *Technical Report 98/97*. Universit de Technologie de Compigne, 1998.

[5] P. BAPTISTE and C. LE PAPE: Edge-finding constraint propagation algorithms for disjunctive and cumulative scheduling. *Proc. 15th Workshop of the U.K. Planning Special Interest Group*, (1996).

[6] ROMAN BARTÁK: Dynamic global constraints in backtracking based environments. *Annals of Operations Research*, **118** (2003), 101-119.

[7] J.C. BECK and M.S. FOX: Scheduling alternative activities. *Proc. 16th National Conf. Artificial Intelligence and 11th Conf. Innovative Applications of Artificial Intelligence*, AAAI Press / The MIT Press, (1999), 680-687.

[8] P. BRUCKE: Complex scheduling problems. 1999. URL
http://citeseer.ist.psu.edu/brucker99complex.html.

[9] P. BRUCKER and S. KNUST: An $O(n^2)$-algorithm for the input-or-output test. *Osnabrucker Schriften zur Mathematik, Reihe preprints*, **259** (2005),
URL ftp://ftp.mathematik.uni-osnabrueck.de/pub/osm/preprints/osmp259.pdf.

[10] J. CARLIER and E. PINSON: Adjustments of head and tails for the job-shop problem. *European J. Operational Research*, **78** (1994), 146-161.

[11] Y. CASEAU and F. LABURTHE: Improved CLP Scheduling with Task Intervals. In P. van Hentenryck, (Ed), *Proc. 11th Int. Conf. Logic Programming*, The MIT press, (1994).

[12] Y. CASEAU and F.LABURTHE: Disjunctive scheduling with task intervals. *Technical report, LIENS Technical Report 95-25*. Ecole Normale Suprieure Paris, Franc, 1995.

[13] R. DECHTER: Constraint Processing (The Morgan Kaufmann Series in Artificial Intelligence). Morgan Kaufmann, 2003.

[14] F. FOCACCI, P. LABORIE and W. NUIJTEN: Solving scheduling problems with setup times and alternative resources. In C.A. Knoblock S. Chien, S. Kambhampati, (Ed), *Proc. 5th Int. Conf. Artificial Intelligence Planning and Scheduling*, AAAI Press, (2000), 92-111.

[15] M.R. GAREY and D.S. JOHNSON: Computers and Intractability: A Guide to the Theory of NP-Completeness. W.H. Freeman and Co., San Francisco, 1979.

[16] S. KUHNERT: Efficient edge-finding on unary resources with optional activities. *Proc. 17th Conf. Applications of Declarative Programming and Knowledge Management*, and *21st Workshop on (Constraint) Logic Programming*, Technical Report 434, Universität Würzburg, (2007), 35-46. URL `http://www1.informatik.uni-wuerzburg.de/databases/INAP/2007/TR/report.pdf`.

[17] P. LABORIE: Algorithms for propagating resource constraints in AI planning and scheduling: existing approaches and new results. *Artificial Intelligence*, **143**(2), (2003), 151-188. ISSN 0004-3702. URL `http://dx.doi.org/10.1016/S0004-3702(02)00362-4`.

[18] P. MARTIN and D.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, (Ed), *Proc. 5th Int. Conf. Integer Programming and Combinatorial Optimization*, (1996), 389-403.

[19] L. MERCIER and P. VAN HENTENRYCK: Edge finding for cumulative scheduling. *Informs J. Computing*, (2005).
URL `http://www.cs.brown.edu/ pvh/ef2.pdf`. To appear.

[20] J.-N. MONETTE, Y. DEVILLE and P.E. DUPONT: A position-based propagator for the open-shop problem. In P. Van Hentenryck and L.A. Wolsey, (Ed), *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 4th Int. Conf.*, (2007), volume 4510 of *Lecture Notes in Computer Science*, Springer, 2007, 186-199.

[21] C. LE PAPE: Implementation of resource constraints in ILOG SCHEDULE: a library for the development of constraint-based scheduling systems. *Intelligent Systems Engineering*, **3**(2), (1994), 55-66. URL `www.ilog.com/products/optimization/tech/research/ise94.pdf`.

[22] C. LE PAPE, P. BAPTISTE and W. NUIJTEN: Constraint-based scheduling: Applying constraint programming to scheduling problems. Kluwer Academic Publishers, 2001.

[23] A. SCHUTT, A. WOLF and G. SCHRADER: Not-first and not-last detection for cumulative scheduling in $O(n^3 \log n)$. *Proc. 16th Int. Conf. Applications of Declarative Programming and Knowledge Management*, (2005), 66-80.

[24] P. TORRES and P. LOPEZ: On not-first/not-last conditions in disjunctive scheduling. *European J. Operational Research*, **127**(2), (1999), 332-343.

[25] P. VILÍM: Batch processing with sequence dependent setup times: New results. *Proc. 4th Workshop of Constraint Programming for Decision and Control*, (2002). URL `http://vilim.eu/petr/cpdc2002.pdf`.

[26] P. VILÍM: $O(n \log n)$ filtering algorithms for unary resource constraint. In J.-C. Régin and M. Rueher, (Ed), *Proc. CP-AI-OR 2004*, volume 3011 of *Lecture Notes in Computer Science*, Springer-Verlag, (2004), 335-347. URL `http://vilim.eu/petr/nlogn.pdf`.

[27] P. VILÍM, R. BARTÁK and O. ČEPEK: Unary resource constraint with optional activities. *Principles and Pracitce of Constraint Programming, 10th Int. Conf.*, Toronto, Canada, volume 3258 of *Lecture Notes in Computer Science*, Springer, (2004), 62-76. URL `http://vilim.eu/petr/cp2004.pdf`.

[28] P. VILÍM, R. BARTÁK and O. ČEPEK: Extension of $O(n \log n)$ filtering algorithms for the unary resource constraint to optional activities. *Contraints*, **10**(4), (2005), 403-425. ISSN 1383-7133. URL `http://vilim.eu/petr/constraints2005.pdf`.

[29] A. WOLF and H. SCHLENKER: Realizing the alternative resources constraint problem with single resource constraints. *Proc. 15th Int. Conf. Applications of Declarative Programming and Knowledge Management and the 18th Workshop on Logic Programming*, Technical Report 327, Bayerische Julius-Maximilians-Universität Würzburg, 2004, 280-294.

[30] A. WOLF: Pruning while sweeping over task intervals. In F. Rossi, (Ed), *Principles and Practice of Constraint Programming*, volume 2833 of *Lecture Notes in Computer Science*, Springer, 2003, 739-753.

[31] A. WOLF and G. SCHRADER: $O(n \log n)$ overload checking for the cumulative constraint and its application. *Proc. 16th Int. Conf. on Applications of Declarative Programming and Knowledge Management*, Springer, (2005), 88-101.