Charles University in Prague,
Faculty of Mathematics and Physics,
Department of Theoretical Computer Science and
Mathematical Logic

Ph.D. Thesis

RNDr. Petr Vilím

Global Constraints in Scheduling

Adviser: **Doc. RNDr. Roman Barták, Ph.D.**

Version: May 25, 2007

Prague 2007

Acknowledgments

In this place, I would like to say my best thanks to everyone who supported me in my work and helped me along my whole Ph.D. study.

First of all I owe a major debt to my supervisor Doc. RNDr. Roman Barták, Ph.D. for his patient leading, collaboration, support and help; especially for guiding my research to this topic, providing important information from CP field, helping me publish my papers and reviewing them (most of them several times).

Next I want to thank to Doc. RNDr. Ondřej Čepek, Ph.D. I wrote with him and Roman Barták a paper to CP 2004 which was very successful. They both contributed to this success.

I also wish to express my gratitude to the whole CP community. To anonymous reviewers for providing me valuable feedback, for pointing out flaws in my papers and helping me to fix them. To my mentors on CP conferences who gave me a lot of helpful advices concerning my research and this a Ph.D. Thesis. I'm also very thankful to all the people with whom I could discuss during several conferences – they often asked very good questions; and good questions are often one half of whole research. My special thanks belongs to Philippe Baptiste, Mats Carlsson, Narendra Jussien, Philippe Laborie, Francois Laburthe, Narendra Jussien, Claude Le Pape, Jean-Charles Régin, Jerome Rôgerie, Francesca Rossi and Armin Wolf.

Last but certainly not least I would like to thanks my parents for encouraging me in this work, helping me and for their countenance during the whole quest. My mother wished so much to see this book (even tough she couldn't understand English). And my father gave me the last push to “finally finish it”.

Contents

Chapter 1

Introduction

1.1 Constraint Programming

Constraint Programming (CP) is becoming a popular tool for solving large combinatorial problems. It provides natural modeling capabilities to describe many real-life problems via domain variables and constraints among these variables. There are generic techniques for constraint satisfaction usually based on integration of search (enumeration, labeling) with constraint propagation (domain filtering). Moreover, the CP framework allows integration of problem-dependent filtering algorithms into the constraint solver.

The idea of CP is to describe the problem declaratively by using constraints over variables. Every variable has a set of possible values – the domain of the variable. In this book all domains are finite, usually they are sets of integers. A constraint determines which combinations of value assignments are acceptable and which are not. A solution of such a problem is an assignment of the variables satisfying all constraints.

Mathematically speaking:

Definition 1 *A Constraint Satisfaction Problem (CSP) is a triple $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ where $\mathcal{X} = \{x_1, \ldots, x_n\}$ is a finite set of variables, $\mathcal{D} = \{D_1, \ldots, D_n\}$ is a set of finite domains (sets of possible values) associated with these variables and $\mathcal{C} = \{C_1, \ldots, C_k\}$ is a finite set of constraints. Each constraint is a relation on the subset of domains $C_i \subseteq D_{i_1} \times \cdots \times D_{i_l}$ which defines assignments of values of the variables $x_{i_1}, \ldots, x_{i_l}$ acceptable by the constraint C_i.*

Solution of the CSP is a tuple $\langle v_1, \ldots, v_n \rangle$ of values assigned to the variables $x_1, \ldots, x_n$ such that each variable x_i has the value from its domain D_i and all constraints $\mathcal{C}$ are satisfied.

To take a simple example of a CSP, consider a problem with two variables

x and y with domains $x \in \{1, 2, 3, 4\}$, $y \in \{2, 3, 4, 5\}$ and constraints $x > y$ and $x + y \geq 7$.

Constraint Propagation

A typical way of solving a CSP is constraint propagation combined with some kind of search. The following paragraphs very informally describe this approach.

One of the key principles of constraint propagation is a notion of *support*. We say that an assignment of a value v to a variable x has a support with respect to a constraint C if there exists an assignment of remaining variables to values from their domains such that the constraint C is satisfied. Naturally if a value has no support with respect to some constraint then it can be removed from the domain – we say that the value is *inconsistent*. The algorithm which determines which values are inconsistent with respect to a constraints is called *filtering algorithm* for the constraint. Usually the filtering algorithms for constraints take turns until no one is able to prune a domain any more (we call that state a *fixpoint*) or one domain becomes empty (i.e., the problem has no solution). This iterative process is called *constraint propagation*.

The result of the propagation is a CSP which is equivalent with the original (in terms of solutions) but it is somehow more consistent. Depending on the level of propagation we distinguish several types of consistencies. For example *arc consistency* (AC) [1] can be achieved by removing all unsupported values from all domains. Although AC is very widely used it is not the strongest level of propagation which can be achieved. In particular, when searching for inconsistent values, we can take into account more than one constraint. This is for example idea of *path consistency* (PC). However such stronger consistency techniques are also more time consuming therefore they are not used very much. For more information about this area of constraint programming see for example [15].

Let us continue with the example above to show how to achieve arc consistency: the propagation of the constraint $x > y$ will deduce that it is not possible that $x = 1$ or $x = 2$ because the lowest possible value of y is 2. Therefore the new domain of x will be $\{3, 4\}$. Then the constraint $x + y \geq 7$ will deduce that $y \geq 3$ therefore the new domain of y will be $\{3, 4, 5\}$. Finally the constraint $x > y$ removes the value $x = 3$, therefore the only possible value for x is 4. This is the end of the constraint propagation phase because no constraint is able to change any domain any more – it is a fixpoint.

[1] Traditionally, AC is defined only for binary constraints. However the informal definition of support given here takes into account also non-binary constraints therefore we should better speak about *generalized arc consistency* (GAC) what is an extension of AC for non-binary constraints.

Sometimes maintaining arc consistency is too expensive. The CSP problem itself is NP-complete [18] and detection of unsupported values for complex constraints can be NP-complete too. In this case we often resign to AC and remove only inconsistent values which can be found quickly. For example we can require only *bound consistency* (BC) which says that only the minimum and maximum value in each domain cannot be unsupported. However in scheduling even bound consistency is usually too expensive.

Propagation itself is able to solve only relatively simple classes of problems. Therefore it is usually combined with some search algorithm (for example backtracking). And here comes another key principle of constraint programming: clearly divide propagation, search and heuristics into different modules. Filtering algorithms for constraints can be also seen as “sub-modules” of the propagation. Figure 1.1 shows how these modules cooperate during the process of finding a solution.

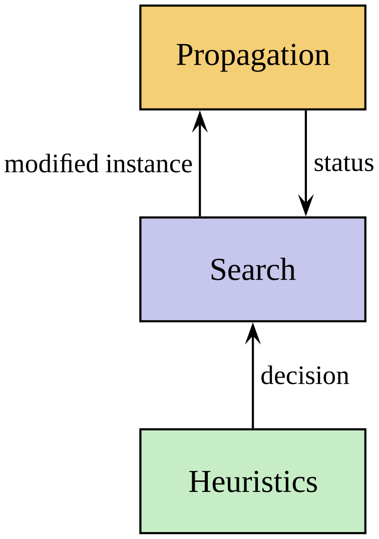

Figure 1.1: Modular CP framework

The purpose of constraint propagation is to narrow down the current problem and to decide whether the problem:

- is solved,
- is infeasible,
- or may be feasible.

This information (status) is used by the search module to decide how to continue:

- When the problem is solved, report the solution to the user. If it is an optimization problem then continue by searching for a better solution.

- When the problem may be feasible then do a “step forward”, i.e., call heuristics to suggest a decision and modify the current problem accordingly. Then continue by constraint propagation on the modified problem.

- When the problem is infeasible then do a “step(s) back”, i.e., return to some decision taken earlier and undo it. If there is no decision to step back then report infeasibility to the user.

There are great benefits coming from this modularity. Constraint solvers such as ILOG Solver, CHIP, SICStus Prolog, CHOCO etc. often offer predefined algorithms for different modules, for example:

- A lot of predefined constraints with variants of filtering algorithms. Sometimes users can even chose which filtering algorithm they want to use for a constraint. And if if there is some aspect of the problem for which there is no predefined constraint matching the needs it is always possible to implement a new one.

- The framework may come with several search algorithms, for example: simple backtracking, branch and bound, limited discrepancy search (LDS) [38], dynamic backtracking [19], or even some local search methods [20].

- And finally there may be also predefined heuristics for commonly used problems. In case of scheduling it may be for example texture-based heuristics [7] or minimum critical set based search [22].

The result is that the user is often able to solve the problem just by picking up the right modules.

For more detailed introduction to Constraint Programming see for example [21, 29, 2, 15].

Global Constraints

A good way how to strengthen constraint propagation is to use one constraint (a so-called *global constraint*) instead of several simple constraints. By using semantic information a global constraint is able to remove more inconsistencies than elementary constraints or it can do it more efficiently. Still filtering algorithm for a global constraints must keep reasonable (polynomial) time and space complexity.

A very good example of the benefit of global constraints is the *all different* constraint. This constraint says that no value can be assigned twice in the given set of variables. Figure 1.2 shows an example of such a problem with four variables u, x, y and z. It is possible to describe the problem by using six binary constraints $u \neq x$, $y \neq x$ etc. However filtering algorithms of these binary constraints has only limited view on the problem. In particular they cannot deduce that $x = 4$ like the all different constraint does (notice that the values 1, 2 and 3 will be taken by variables u, y and z, therefore the only possible value for variable x is 4).

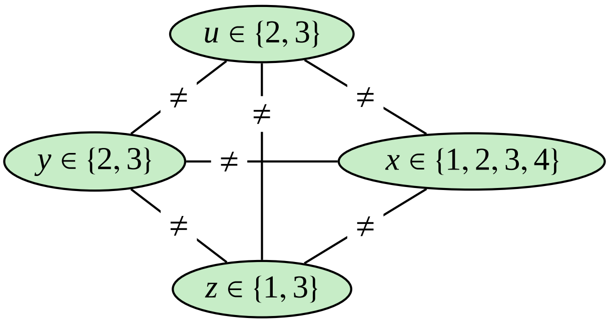

Figure 1.2: Global constraint alldiff ensures that variables u, x, y and z have different values. In this case it is an encapsulation of 6 binary "not-equal" constraints.

Many global constraints have been designed for problems of various classes. Famous constraints are *all different* [26, 30] or *cardinality* constraint [27]. In scheduling there is *unary resource constraint* [11, 24, 4, 25, 33] for modeling simple machines, *cumulative resource constraint* [4, 14, 3, 25] for modeling for example a pool of workers, and *reservoir* [23] for modeling a renewable sources of power.

This book is dedicated to global constraints for scheduling, namely unary resource, unary resource with optional activities and batch processing with sequence dependent setup times. The subjects of search and heuristics is almost leaved out.

1.2 Scheduling

Scheduling is one of the most successful application areas of constraint programming. Behind this success there are powerful global constraints which are able to model resource restrictions. Let us introduce scheduling and the way it is modeled as a CSP on an example of typical academic scheduling problem – *jobshop*.

In jobshop there is a set of *m resources* which we can be used to complete *n jobs*. Each job consists of a sequence of *m* activities[2] each of them requires a different resource. The activities in a job must be processed in the given order. A resource can be used only by one activity in the same time and during this work it cannot be interrupted by another activity. Exact time needed to execute each activity is known in advance. Delays between activities can be arbitrary. The task

[2] When speaking of jobshop, resources are traditionally called machines and activities are called operations. We use terms resource and activity to be consistent within the whole book.

is to find a schedule with the minimal *makespan*, i.e., the minimal completion time of all jobs.

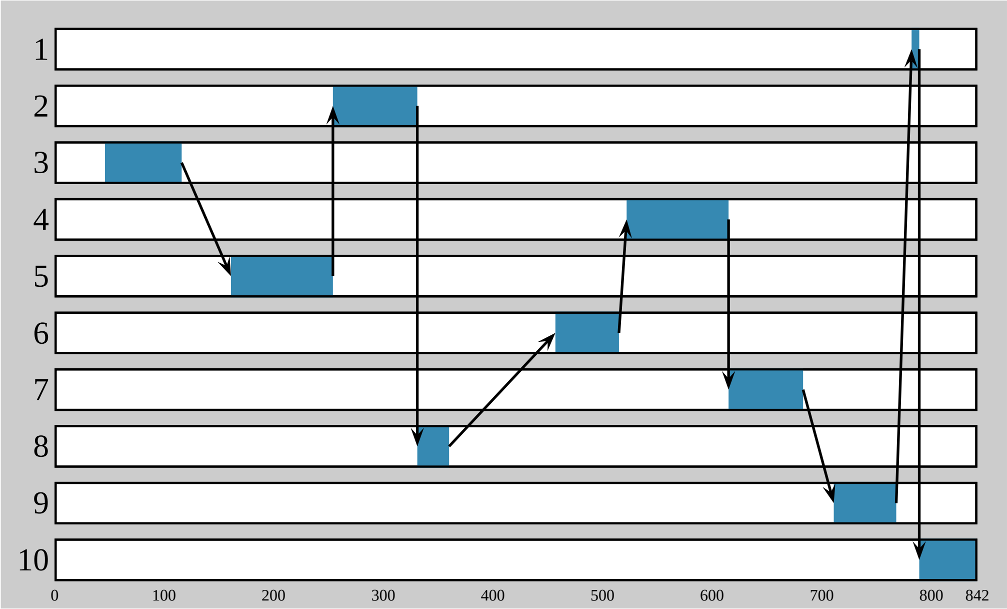

Figure 1.3: A job in the jobshop problem.

Figures 1.3 and 1.4 illustrate a jobshop problem with $n = 10$ jobs and $m = 10$ resources. The first figure shows one of the jobs as it is scheduled in the optimal solution. Precedences between the activities from the job are indicated by arrows. The second figure shows the whole optimal solution (without the arrows to mark precedences). Each job is indicated by a different color. The problem in the figures (LA19) was taken from a common benchmark set for scheduling – OR Library [1].

Another often used benchmark problem is *openshop*. It is similar to jobshop but activities in a job can be processed in arbitrary order.

To model such scheduling problems as a CSP we usually use unary resource constraints. A unary resource constraint takes a set of activities which must be processed on the resource and ensures that the resource is not used by two activities in the same time. Another constraint we will use is a binary precedence constraint $j \ll i$ which expresses that the activity j must finish before the activity i can start.

The jobshop problem LA19 in a CP pseudo-language is provided on example 1.1.

```
{ Declare activities and their durations: }
$j_{1,1}$ := activity(44);
$j_{1,2}$ := activity(5);
$j_{1,3}$ := activity(58);
...
$j_{10,10}$ := activity(26);

{ Post precedence constraints: }
$j_{1,1} \ll j_{1,2} \ll j_{1,3} \ll j_{1,4} \ll j_{1,5} \ll j_{1,6} \ll j_{1,7} \ll j_{1,8} \ll j_{1,9} \ll j_{1,10}$ ;
$j_{2,1} \ll j_{2,2} \ll j_{2,3} \ll j_{2,4} \ll j_{2,5} \ll j_{2,6} \ll j_{2,7} \ll j_{2,8} \ll j_{2,9} \ll j_{2,10}$ ;
$j_{3,1} \ll j_{3,2} \ll j_{3,3} \ll j_{3,4} \ll j_{3,5} \ll j_{3,6} \ll j_{3,7} \ll j_{3,8} \ll j_{3,9} \ll j_{3,10}$ ;
$j_{4,1} \ll j_{4,2} \ll j_{4,3} \ll j_{4,4} \ll j_{4,5} \ll j_{4,6} \ll j_{4,7} \ll j_{4,8} \ll j_{4,9} \ll j_{4,10}$ ;
$j_{5,1} \ll j_{5,2} \ll j_{5,3} \ll j_{5,4} \ll j_{5,5} \ll j_{5,6} \ll j_{5,7} \ll j_{5,8} \ll j_{5,9} \ll j_{5,10}$ ;
$j_{6,1} \ll j_{6,2} \ll j_{6,3} \ll j_{6,4} \ll j_{6,5} \ll j_{6,6} \ll j_{6,7} \ll j_{6,8} \ll j_{6,9} \ll j_{6,10}$ ;
$j_{7,1} \ll j_{7,2} \ll j_{7,3} \ll j_{7,4} \ll j_{7,5} \ll j_{7,6} \ll j_{7,7} \ll j_{7,8} \ll j_{7,9} \ll j_{7,10}$ ;
$j_{8,1} \ll j_{8,2} \ll j_{8,3} \ll j_{8,4} \ll j_{8,5} \ll j_{8,6} \ll j_{8,7} \ll j_{8,8} \ll j_{8,9} \ll j_{8,10}$ ;
$j_{9,1} \ll j_{9,2} \ll j_{9,3} \ll j_{9,4} \ll j_{9,5} \ll j_{9,6} \ll j_{9,7} \ll j_{9,8} \ll j_{9,9} \ll j_{9,10}$ ;
$j_{10,1} \ll j_{10,2} \ll j_{10,3} \ll j_{10,4} \ll j_{10,5} \ll j_{10,6} \ll j_{10,7} \ll j_{10,8} \ll j_{10,9} \ll j_{10,10}$ ;

{ Post unary resource constraints: }
unary_resource($j_{1,8}, j_{2,9}, j_{3,1}, j_{4,9}, j_{5,9}, j_{6,9}, j_{7,9}, j_{8,4}, j_{9,8}, j_{10,1}$);
unary_resource($j_{1,7}, j_{2,4}, j_{3,7}, j_{4,5}, j_{5,6}, j_{6,3}, j_{7,8}, j_{8,3}, j_{9,7}, j_{10,7}$);
unary_resource($j_{1,6}, j_{2,2}, j_{3,9}, j_{4,3}, j_{5,8}, j_{6,1}, j_{7,7}, j_{8,8}, j_{9,6}, j_{10,4}$);
unary_resource($j_{1,10}, j_{2,10}, j_{3,2}, j_{4,8}, j_{5,1}, j_{6,6}, j_{7,1}, j_{8,6}, j_{9,4}, j_{10,3}$);
unary_resource($j_{1,3}, j_{2,8}, j_{3,10}, j_{4,4}, j_{5,10}, j_{6,2}, j_{7,4}, j_{8,2}, j_{9,1}, j_{10,8}$);
unary_resource($j_{1,4}, j_{2,1}, j_{3,3}, j_{4,6}, j_{5,7}, j_{6,5}, j_{7,3}, j_{8,7}, j_{9,5}, j_{10,2}$);
unary_resource($j_{1,2}, j_{2,6}, j_{3,4}, j_{4,7}, j_{5,3}, j_{6,7}, j_{7,6}, j_{8,5}, j_{9,3}, j_{10,9}$);
unary_resource($j_{1,1}, j_{2,7}, j_{3,8}, j_{4,2}, j_{5,5}, j_{6,4}, j_{7,5}, j_{8,9}, j_{9,2}, j_{10,6}$);
unary_resource($j_{1,9}, j_{2,3}, j_{3,5}, j_{4,1}, j_{5,2}, j_{6,8}, j_{7,2}, j_{8,10}, j_{9,9}, j_{10,10}$);
unary_resource($j_{1,5}, j_{2,5}, j_{3,6}, j_{4,10}, j_{5,4}, j_{6,10}, j_{7,10}, j_{8,1}, j_{9,10}, j_{10,5}$);

{ Post objective: }
minimize(makespan);
```

Example 1.1: Jobshop LA19 in a CP pseudo-code

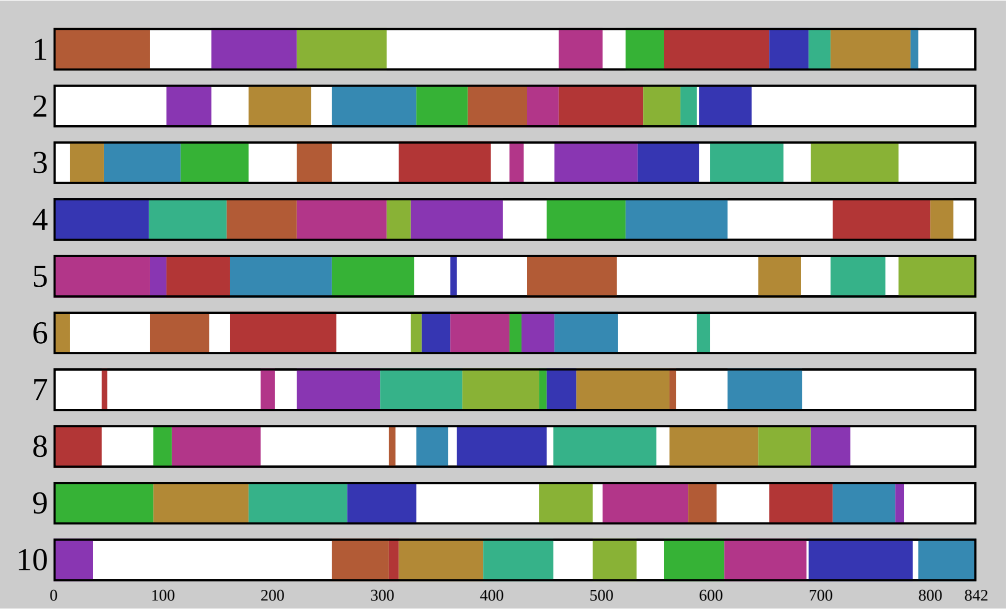

Figure 1.4: An optimal solution of the LA19 jobshop problem

1.3 Structure of this book

The major part of this book is about propagation of various resource constraints: the basic unary resource constraint (chapter 2), unary resource constraint with optional activities (chapter 3) and batch processing with sequence dependent setup times (chapter 4). The book does not touch the topic of cumulative resources even tough I believe that some of the concepts provided here would be useful for this kind of resource too.

The chapter 2 starts by introduction of basic unary resource constraint and historical background of the area. Two major parts of this chapter are Propagation Rules – a theoretical part which investigates principles behind the unary resource constraint from the mathematical point of view, and Filtering Algorithms – a more practical part which uses acquired knowledge to design efficient propagation algorithms.

The chapter 3 is dedicated to optional activities. It is an increasingly studied area of scheduling and mixed planning/scheduling. The task is not only to assign exact dates to activities but also to decide among several processing alternatives. This leads to the concept of optional activities which may but do not have to be in the final schedule.

Finally the chapter 4 studies scheduling on a batching resource – a resource which can process several activities together under the condition that they are “compatible” and their processing starts together and ends together. Furthermore the resource may need some setup time to switch from one kind of processing to another one.

Unless stated otherwise all the propositions and algorithms in this book are my own results or I played a major role in their invention. With the help of Roman Barták and Ondřej Čepek I already published most of these results in the following papers: [34, 32, 33, 36, 37]. Comparing with these papers this book does not have any page limit therefore I could explain some topics more formally and in more depths, namely a concept of fixpoint (chapter 2.3) and a transitive closure of precedences (chapter 2.4.7). There are also new results concerning equivalence of propagation rules for optional activities (chapter 3) which were not published yet, namely the propositions 8 and 9.

Chapter 2

Unary Resource Constraint

2.1 Definition

In scheduling, a *unary resource* is an often used generalization of a machine (or a job in openshop), as was demonstrated at chapter 1.2. A unary resource models a set of non-interruptible *activities* T which must not overlap in time – once a resource starts process an activity it cannot stop or change the activity until processing of the activity is finished.

Each activity $i \in T$ can be restricted by the following limits:

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

A (sub)problem is to find a schedule satisfying all these requirements. This problem is long known to be computationally difficult [18][1].

In constraint programming we associate a *unary resource constraint* with each unary resource. The purpose of this constraint is to reduce the search space by elimination of infeasible values (e.g., start times). This process is called *propagation*, an actual propagation algorithm is often called a *filtering algorithm*.

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

The first simplification (in sake of speed) is the consideration of a set of possible start times of an activity i to be an interval $\langle \mathrm{est}_i, \mathrm{lct}_i - \mathrm{p}_i \rangle$. Our only concern is

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

to contract this interval by tightening the bounds est_i and lct_i, we are not interested in holes in this intervals. Thus the role of a resource constraint can be seen as a process of tightening of time windows $\langle \mathrm{est}_i, \mathrm{lct}_i \rangle$.

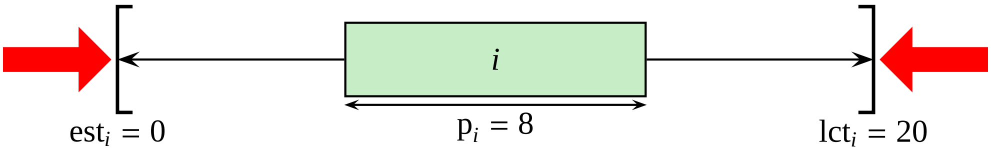

Figure 2.1: An activity i with the earliest starting time 0, the latest completion time 20 and the processing time 8. Red arrows show the intention of the filtering algorithms (tightening of the time interval $\langle \mathrm{est}_i,\ \mathrm{lct}_i \rangle$).

2.2 Existing Filtering Algorithms

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.

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 [11]. Their algorithm has time complexity $\mathcal{O}(n \log n)$, but it is quite complicated to implement.

Later other two versions of this algorithm were developed by Paul Martin with David B. Shmoys [24] and Wim Nuijten [25]. These versions have time complexity $\mathcal{O}(n^2)$, but are much easier to implement and therefore they are widely used today.

This book describes a new version of this algorithm with time complexity $\mathcal{O}(n \log n)$, which is not so hard to implement as the algorithm in [11]. Together with Roman Barták and Ondřej Čepek I published the algorithm in [36] and [37].

Not-First/Not-Last

The first Not-First/Not-Last algorithm was introduced by Philippe Baptiste and Claude Le Pape in [4], its time complexity is $\mathcal{O}(n^2)$.

Later Philippe Torres and Pierre Lopez designed a simpler and faster version of this algorithm [28]. Nevertheless, the worst-case time complexity remains the same – $O(n^2)$. To achieve the same filtering as the original algorithm [4], more iterations of the algorithm may be needed. Nevertheless this algorithm is faster than the previous one.

In this book I describe a new version of this algorithm with worst-case time complexity $O(n \log n)$. I already published the algorithm in [33] and [37]. Like the algorithm [28], more iterations may be needed in order to achieve the filtering of the original algorithm [4], but the algorithm is faster than [28].

Detectable Precedences

This is a new algorithm with worst-case time complexity $O(n \log n)$ which I introduced in [33]. The algorithm was also published in [37].

Precedence Graph

Propagation based on precedence graph was mentioned by several authors, e.g., by Brucker in [10] or by Focacci, Laborie and Nuijten in [16]. In this book I present a simple $O(n^2)$ algorithm which was also published in [33]. This algorithm is tightly related to Detectable Precedences algorithm.

Overload Checking

Overload Checking was originally part of the Edge Finding [11, 24, 25]. 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.

Sweeping

Sweeping is another propagation technique for unary resource constraint. Sweeping algorithm proposed by Wolf in [39] has the same propagation power as algorithms Edge Finding and Not-First/Not-Last together. The worst-case time complexity of this algorithm is $O(n^2)$.

This book does not include this algorithm.

Task Intervals

Propagation using task intervals is presented in [12, 13]. 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 $\mathcal{O}(n^3)$ however in practice it should be considerably faster than that.

The filtering power of this algorithm is not better than combination of Overload Checking, Edge Finding and Not-First/Not-Last. Therefore this algorithm is also not included in this book.

2.3 Fixpoint

With the exception of the Overload Checking, all the algorithms mentioned in the previous section 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 the algorithm 2.1 and the figure 2.2.

Algorithm 2.1: Computation of the fixpoint

```
repeat
  repeat
    repeat
      repeat
        if not Overload_Checking then
          fail;
        Detectable_Precedences;
      until no more propagation;
      Not_First/Not_Last;
    until no more propagation;
    Edge_Finding;
  until no more propagation;
  Precedence_Graph;
until no more propagation;
```

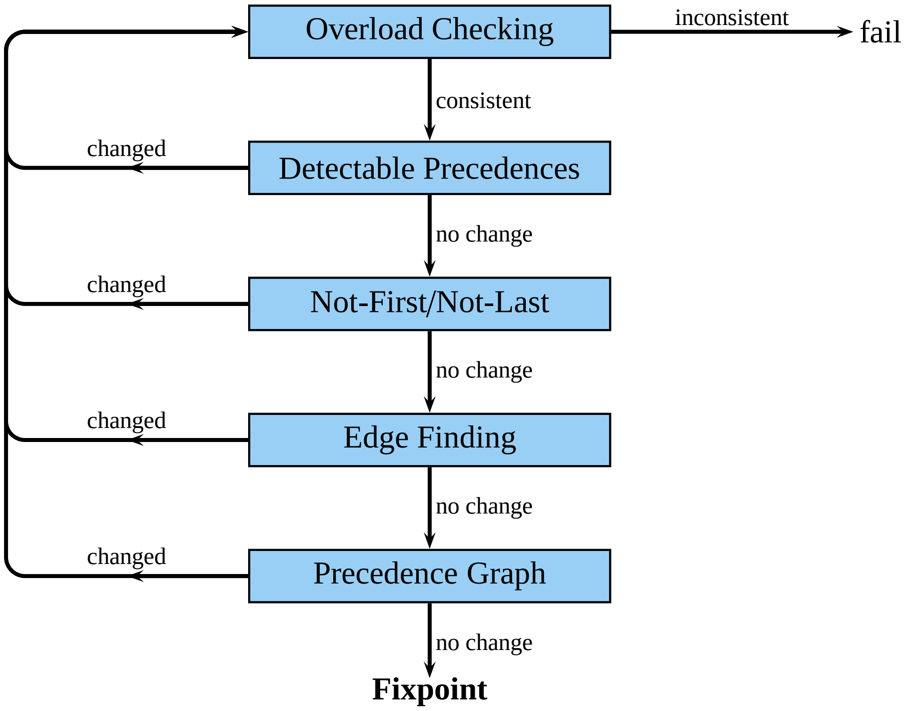

Figure 2.2: Computation of the fixpoint

As Krzysztof Apt proved by Domain Reduction Theorem [2], if all propagation rules are monotonic 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. According to my experimental results the sequence presented in figure 2.2 seems to be the best.

2.3.1 Monotonicity

Definition 2 *Let $\langle \mathcal{X}, D, C\rangle$ be a CSP, $\mathcal{D} = \langle D_1, \ldots, D_n\rangle$ and let f be a filtering function:*

$$f : \mathcal{P}(D_1) \times \ldots \times \mathcal{P}(D_n) \to \mathcal{P}(D_1) \times \ldots \times \mathcal{P}(D_n)$$

Let $\langle d_1, \ldots, d_n\rangle$ and $\langle e_1, \ldots, e_n\rangle$ be two states of domains (i.e., $d_1, e_1 \subseteq D_1, d_2, e_2 \subseteq D_2 \ldots, d_n, e_n \subseteq D_n$) and let:

$$f(\langle d_1, \ldots, d_n\rangle) = \langle d'_1, \ldots, d'_n\rangle$$
$$f(\langle e_1, \ldots, e_n\rangle) = \langle e'_1, \ldots, e'_n\rangle$$

The filtering function f is monotonic *if:*

$$d_1 \subseteq e_1, \ldots, d_n \subseteq e_n \;\Rightarrow\; d'_1 \subseteq e'_1, \ldots, d'_n \subseteq e'_n$$

In other words: considering two instances of the same CSP problem a monotonic filtering function cannot yield worse pruning for the more specific instance.

All the filtering algorithms from this book are monotonic. This property can be easily confirmed by close inspection of each rule. We will not emphasize this property for each rule again.

2.4 Propagation Rules

In this chapter we define rules which are the roots of propagation algorithms for the unary resource in this book. 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 Graph).

The algorithms themselves will be presented later in the chapter 2.5. We will not specifically prove the equivalence of presented algorithms with other versions of these algorithm. The reason is simple – since both algorithms propagate according to the same rules and both algorithms make full propagation the result has to be the same.

2.4.1 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 in chapter 2.1.

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:

$$\mathrm{ect}_\Omega = \max\left\{\mathrm{est}_{\Omega'} + \mathrm{p}_{\Omega'},\ \Omega' \subseteq \Omega\right\} \tag{2.1}$$

Symmetrically we define estimation of the latest start time:

$$\mathrm{lst}_\Omega = \min\left\{\mathrm{lct}_{\Omega'} - \mathrm{p}_{\Omega'},\ \Omega' \subseteq \Omega\right\} \tag{2.2}$$

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

$$\begin{aligned}
\mathrm{est}_\emptyset &= -\infty \\
\mathrm{lct}_\emptyset &= \infty \\
\mathrm{p}_\emptyset &= 0 \\
\mathrm{ect}_\emptyset &= -\infty \\
\mathrm{lst}_\emptyset &= \infty
\end{aligned}$$

2.4.2 Binary Precedence Constraint

Together with the unary resource constraint binary precedence constraints are used to model shop scheduling problems. A precedence constraint $i \ll j$ ensures that the activity i finishes before the activity j starts.

Propagation of the precedence constraint $i \ll j$ is simple: whenever est_i is increased the constraint propagates the change to est_j:

$$i \ll j \quad \Rightarrow \quad \mathrm{est}_j := \max\{\mathrm{est}_j,\ \mathrm{est}_i + \mathrm{p}_i\} \tag{2.3}$$

Similarly when lct_j is decreased, lct_i can be adjusted:

$$i \ll j \quad \Rightarrow \quad \mathrm{lct}_i := \min\{\mathrm{lct}_i,\ \mathrm{lct}_j - \mathrm{p}_j\} \tag{2.4}$$

Note that the rules are symmetrical. This is typical for all filtering rules in scheduling – there is a rule for updating est_i and a symmetric rule for updating lct_i. Since propagation algorithms for symmetric rules are also symmetric we will always consider only one symmetric rule, usually the one which updates est_i.

Adjustments according to the rules (2.3) and (2.4) can be done in $\mathcal{O}(1)$. In the section 2.4.7 we will see that if the activities i and j are on the same resource then it may be better to propagate the precedence together with other precedences using a precedence graph.

2.4.3 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. [40]) says that if the set Ω cannot be processed within its bounds then no solution exists:

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

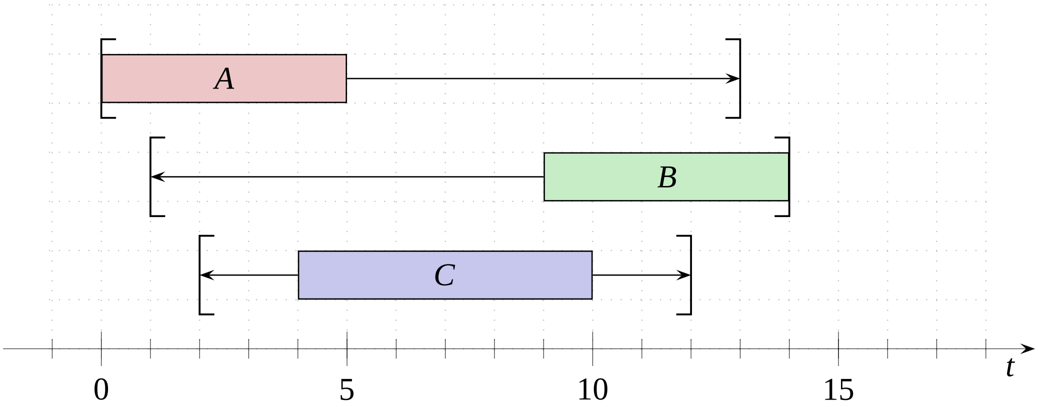

Figure 2.3: A sample problem for Overload Checking

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

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

$$\mathrm{LCut}(T, j) = \left\{k,\ k \in T\ \&\ \mathrm{lct}_k \leq \mathrm{lct}_j\right\} \tag{2.5}$$

The new rule is:

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

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

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

1. Let the rule (OL) detects inconsistency. Let j be such an activity from the set Ω that $\mathrm{lct}_j = \mathrm{lct}_\Omega$. Clearly $\mathrm{lct}_j = \mathrm{lct}_{\mathrm{LCut}(T,j)}$. Moreover $\Omega \subseteq \mathrm{LCut}(T, j)$

and therefore $\mathrm{est}_\Omega + \mathrm{p}_\Omega \leq \mathrm{ect}_{\mathrm{LCut}(T,j)}$. So we have:

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

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

2. Let the rule (OL') detects inconsistency. We define Ω to be such a subset of $\mathrm{LCut}(T, j)$, that $\mathrm{est}_\Omega + \mathrm{p}_\Omega = \mathrm{ect}_{\mathrm{LCut}(T,j)}$ (thanks to the definition (2.1) such a set must exists). Than:

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

and so the rule (OL) also detects inconsistency.

□

2.4.4 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., [4]).

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 the figure 2.4):

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

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

$$\Omega \ll i \quad \Rightarrow \quad \mathrm{est}_i := \max\{\mathrm{est}_i,\ \mathrm{ect}_\Omega\} \tag{2.6}$$

The whole rule is:

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

There is also a symmetric rule, which adjust lct_i:

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

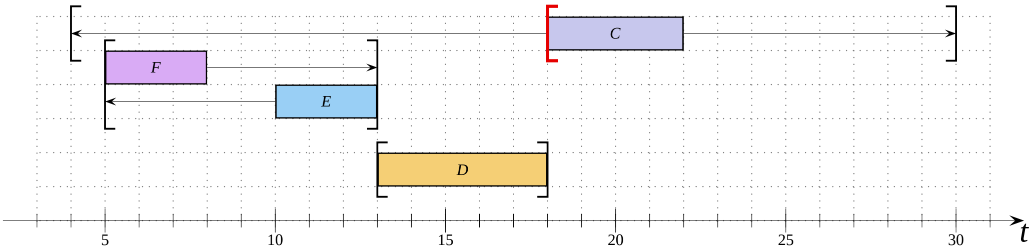

Figure 2.4: A sample problem for Edge Finding: est_C can be changed to 18.

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 chapter 2.5.7:

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

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

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

   And also:

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

   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 prove 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:

   $$\mathrm{ect}_{\mathrm{LCut}(T,j)\cup\{i\}} = \mathrm{est}_{\Omega'} + \mathrm{p}_{\Omega'} \tag{2.7}$$

   Note that thanks to the definition (2.1) of ect such a set Ω' must exist.

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

   $$\mathrm{est}_{\Omega'} + \mathrm{p}_{\Omega'} \overset{(2.7)}{=} \mathrm{ect}_{\mathrm{LCut}(T,j)\cup\{i\}} \overset{(EF')}{>} \mathrm{lct}_j \geq \mathrm{lct}_{\Omega'}$$

   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 $\mathrm{est}_i \geq \mathrm{ect}_{\mathrm{LCut}(T,j)}$ and the rule (EF') achieves nothing. For this set Ω we have:

   $$\min\{\mathrm{est}_\Omega,\ \mathrm{est}_i\} + \mathrm{p}_\Omega + \mathrm{p}_i = \mathrm{est}_{\Omega'} + \mathrm{p}_{\Omega'} \overset{(2.7)}{=} \mathrm{ect}_{\mathrm{LCut}(T,j)\cup\{i\}} \overset{(EF')}{>} \mathrm{lct}_j \geq \mathrm{lct}_\Omega$$

   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., $\mathrm{ect}_\Omega = \mathrm{ect}_{\mathrm{LCut}(T,j)}$. We already know that $\mathrm{ect}_\Omega \leq \mathrm{ect}_{\mathrm{LCut}(T,j)}$ because $\Omega \subseteq \mathrm{LCut}(T, j)$. Suppose now for a contradiction that:

   $$\mathrm{ect}_\Omega < \mathrm{ect}_{\mathrm{LCut}(T,j)} \tag{2.8}$$

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

   $$\mathrm{ect}_{\mathrm{LCut}(T,j)} = \mathrm{est}_\Phi + \mathrm{p}_\Phi \tag{2.9}$$

   Therefore:

   $$\mathrm{est}_\Omega + \mathrm{p}_\Omega \leq \mathrm{ect}_\Omega \overset{(2.8)}{<} \mathrm{ect}_{\mathrm{LCut}(T,j)} \overset{(2.9)}{=} \mathrm{est}_\Phi + \mathrm{p}_\Phi \tag{2.10}$$

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

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

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

$$\forall k \in \mathrm{LCut}(T, j): \quad \mathrm{est}_k \geq \mathrm{est}_{\Omega'} \;\Rightarrow\; k \in \Omega \tag{2.11}$$

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

$$\forall k \in \mathrm{LCut}(T, j): \quad \mathrm{est}_k \geq \mathrm{est}_{\Phi} \;\Rightarrow\; k \in \Phi \tag{2.12}$$

Combining properties (2.11) and (2.12) together we have that either $\Omega \subseteq \Phi$ (if $\mathrm{est}_{\Omega'} \geq \mathrm{est}_{\Phi}$) or $\Phi \subseteq \Omega$ (if $\mathrm{est}_{\Omega'} \leq \mathrm{est}_{\Phi}$). However, $\Phi \subseteq \Omega$ is not possible, because in this case $\mathrm{est}_{\Phi} + \mathrm{p}_{\Phi} \leq \mathrm{ect}_{\Omega}$ which contradicts the inequality (2.10). The result is that $\Omega \subsetneq \Phi$, and so $\mathrm{p}_{\Omega} < \mathrm{p}_{\Phi}$.

Now we are ready to prove the contradiction:

$$\begin{aligned}
&\mathrm{ect}_{\mathrm{LCut}(T,j)\cup\{i\}} \overset{(2.7)}{=} \\
&\quad = \mathrm{est}_{\Omega'} + \mathrm{p}_{\Omega'} \\
&\quad = \min\{\mathrm{est}_{\Omega},\ \mathrm{est}_i\} + \mathrm{p}_{\Omega} + \mathrm{p}_i && \text{because } \Omega = \Omega' \setminus \{i\} \\
&\quad = \min\{\mathrm{est}_{\Omega} + \mathrm{p}_{\Omega} + \mathrm{p}_i,\ \mathrm{est}_i + \mathrm{p}_{\Omega} + \mathrm{p}_i\} \\
&\quad < \min\{\mathrm{est}_{\Phi} + \mathrm{p}_{\Phi} + \mathrm{p}_i,\ \mathrm{est}_i + \mathrm{p}_{\Phi} + \mathrm{p}_i\} && \text{by (2.10) and } \mathrm{p}_{\Omega} < \mathrm{p}_{\Phi} \\
&\quad \leq \mathrm{ect}_{\mathrm{LCut}(T,j)\cup\{i\}} && \text{because } \Phi \subseteq \mathrm{LCut}(T, j)
\end{aligned}$$

□

The rule (EF') has a very useful property, which we will use 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 $\mathrm{lct}_{j_1} \leq \mathrm{lct}_{j_2}$. Then $\mathrm{LCut}(T, j_1) \subseteq \mathrm{LCut}(T, j_2)$ and so $\mathrm{ect}_{\mathrm{LCut}(T,j_1)} \leq \mathrm{ect}_{\mathrm{LCut}(T,j_2)}$. Therefore j_2 yields better propagation than j_1. □

2.4.5 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:

$$\mathrm{est}_{\Omega} + \mathrm{p}_{\Omega} > \mathrm{lct}_i - \mathrm{p}_i \tag{2.13}$$

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

$$\mathrm{lct}_i := \min\left\{\mathrm{lct}_i,\ \max\left\{\mathrm{lct}_j - \mathrm{p}_j,\ j \in \Omega\right\}\right\} \tag{2.14}$$

The full Not-Last rule is:

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

For a demonstration, see figure 2.5.

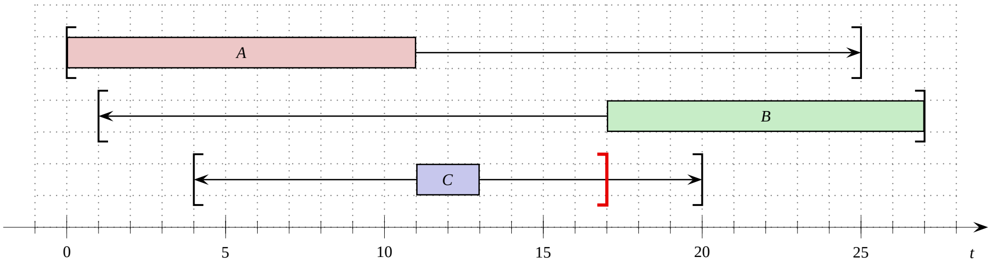

Figure 2.5: A sample problem for Not-Last: lct_C can be changed to 17.

The only algorithm which is based directly on this rule is by Baptiste and Le Pape [4]. 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 [28] modified this rule in the following way. In order to achieve the fixpoint the algorithm must be iterated (see chapter 2.3). 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:

$$\mathrm{NLSet}(T, i) = \left\{j,\ j \in T\ \&\ \mathrm{lct}_j - \mathrm{p}_j < \mathrm{lct}_i\ \&\ j \neq i\right\} \tag{2.15}$$

Because otherwise $\max\left\{\mathrm{lct}_j - \mathrm{p}_j,\ j \in \Omega\right\} \geq \mathrm{lct}_i$. 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) is valid, 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 $\mathrm{NLSet}(T, i)$:

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

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:

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

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:

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

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

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

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 number of iterations can differ, the maximum number of $n - 1$ iterations is not guaranteed in this case.

2.4.6 Detectable Precedences

The idea of detectable precedences was introduced in [32] for a batch resource with sequence dependent setup times and then simplified for the unary resource in [33]. The figure 2.6 provides an example when neither Edge Finding nor Not-First/Not-Last algorithm is able change any bound.

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 $\mathrm{est}_C \geq \mathrm{est}_A + \mathrm{p}_A + \mathrm{p}_B = 10$. This is exactly what the Detectable Precedences algorithm does.

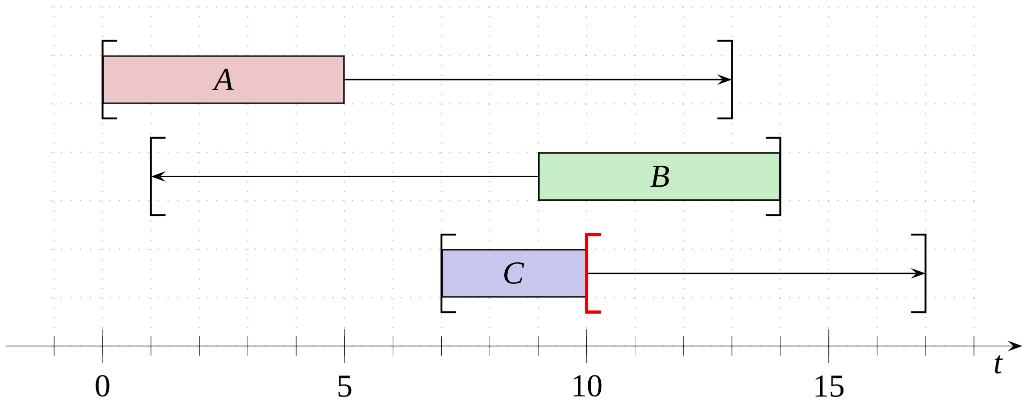

Figure 2.6: A sample problem for Detectable Precedences. est_C can be changed to 10.

Definition 3 *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:*

$$\mathrm{est}_i + \mathrm{p}_i > \mathrm{lct}_j - \mathrm{p}_j \quad \Rightarrow \quad j \ll i \tag{2.16}$$

Notice that in figure 2.6 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:

$$\mathrm{DPrec}(T, i) = \left\{ j,\ j \in T \ \&\ \mathrm{est}_i + \mathrm{p}_i > \mathrm{lct}_j - \mathrm{p}_j \ \ \&\ j \neq i \right\} \tag{2.17}$$

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 (2.6)):

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

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

2.4.7 Precedence Graph

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 (will be defined later).

Taking all these precedences into account, we can define a set of all predecessors of an activity i on a resource as:

$$\mathrm{Prec}(i) = \{j,\ j \in T\ \&\ j \ll i\} \tag{2.18}$$

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

The propagation rule is (notice the similarity with the rules (2.6) and (DP)):

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

There is also a symmetric version for adjustment of lct_i.

The main difficulty is to find all these precedences. Once we know them, propagation according to this rule is quite easy, the algorithm is presented in the chapter 2.5.1.

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

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 in the whole search subtree.*

Proof: 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 a little bit later somewhere in the search subtree[2].

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

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

In the subtree value est_i can only increase and the value lct_j can only decrease:

$$\mathrm{est}'_i \geq \mathrm{est}_i$$
$$\mathrm{lct}'_i \leq \mathrm{lct}_i$$

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

Thus:

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

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

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:

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

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:

$$\mathrm{est}_\Omega > \mathrm{lct}_\Omega - \mathrm{p}_\Omega - \mathrm{p}_i \tag{2.19}$$

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

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

In this inequality, est_Ω can be replaced by the right side of the inequality (2.19):

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

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

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

So the condition (2.16) 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 32 is only a subset of the item 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 Graph. In all these cases the adjustment is done according to the same idea:

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

For assurance see the rules (DP), (EF), (PG). The rule for binary constraint (2.3) is a special case of the rule above for $\Omega = \{j\}$. This allows to define a propagated precedence as follows.

Definition 4 *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:*

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

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.

Transitive closure of precedences

First let us define a transitive closure:

Definition 5 *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 [10] or [16]. 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 a 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:

   $$\mathrm{est}_c \geq \mathrm{est}_b + \mathrm{p}_b$$

   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:

   $$\mathrm{lct}_b - \mathrm{p}_b \geq \mathrm{lct}_a$$

   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 $i_1, i_2, \dots, i_n \in T$ and let $i_1 \ll i_2 \ll \cdots \ll i_n$ be propagated precedences and at least one of them is detectable. Then the precedence $i_1 \ll i_n$ is detectable.*

Proof: We can prove that the precedence $i_1 \ll i_n$ is detectable by repeated application of the previous proposition. For a demonstration, see figure 2.7.

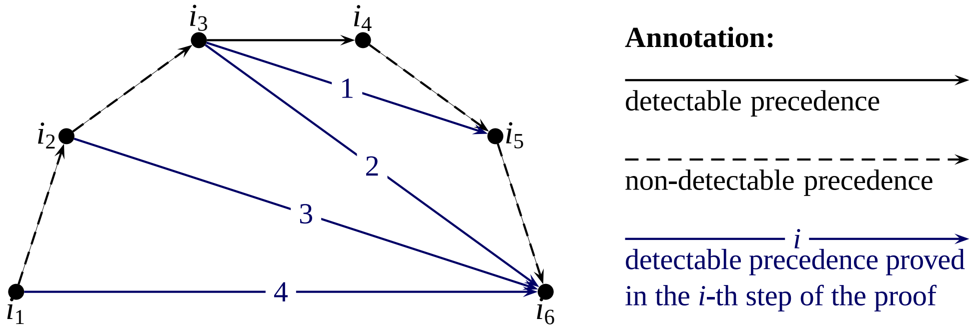
i_3
i_4
i_2
i_5
i_1
i_6
1
2
3
4
Annotation:
detectable precedence
non-detectable precedence
i
detectable precedence proved
in the i-th step of the proof

Figure 2.7: Computation of transitive closure.

On this picture there is a sequence of propagated precedences $i_1 \ll i_2 \ll i_3 \ll i_4 \ll i_5 \ll i_6$ and only one of them ($i_3 \ll i_4$) is detectable. First we prove that $i_3 \ll i_5$ is detectable using lemma 1 (because $i_3 \ll i_4$ is detectable and $i_4 \ll i_5$ is propagated). Next we prove that $i_3 \ll i_6$ is also detectable (because $i_3 \ll i_5$ is detectable and $i_5 \ll i_6$ is propagated). And so on we can prove that $i_2 \ll i_6$ and finally $i_1 \ll i_6$. □

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.

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

2.5 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.

2.5.1 Precedence Graph

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

The algorithm 2.2 requires a transitive closure of non-detectable precedences and its reparation whenever new non-detectable precedence (search decision) is added into the system (as was described in the section 2.4.7).

The idea of the algorithm follows. For each activity i we compute value $\mathrm{ect}_{\mathrm{Prec}(i)}$ 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). Worst-case time complexity of the algorithm is $\mathcal{O}(n^2)$.

Algorithm 2.2: Precedence Graph Based Filtering

```
1  for $i \in T$ do begin
2     $m := -\infty$;
3     for $j \in T$ in non-decreasing order of $\mathrm{est}_j$ do
4        if $j \ll i$ then
5           $m := \max\left\{m, \mathrm{est}_j\right\} + \mathrm{p}_j$;
6     $\mathrm{est}_i := \max\{m, \mathrm{est}_i\}$;
7  end;
```

A symmetric algorithm adjusts lct_i.

2.5.2 Θ-Tree

One of the main complexities of the filtering algorithms for a 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-tree was first introduced in [33] and then slightly modified in [37]. In the following, we will use this modified version of Θ-tree.

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 the Θ-tree see figure 2.8.

[3] This is the main difference from [33]. The tree is deeper by one level, however a simpler computation of ect and ΣP compensates that.

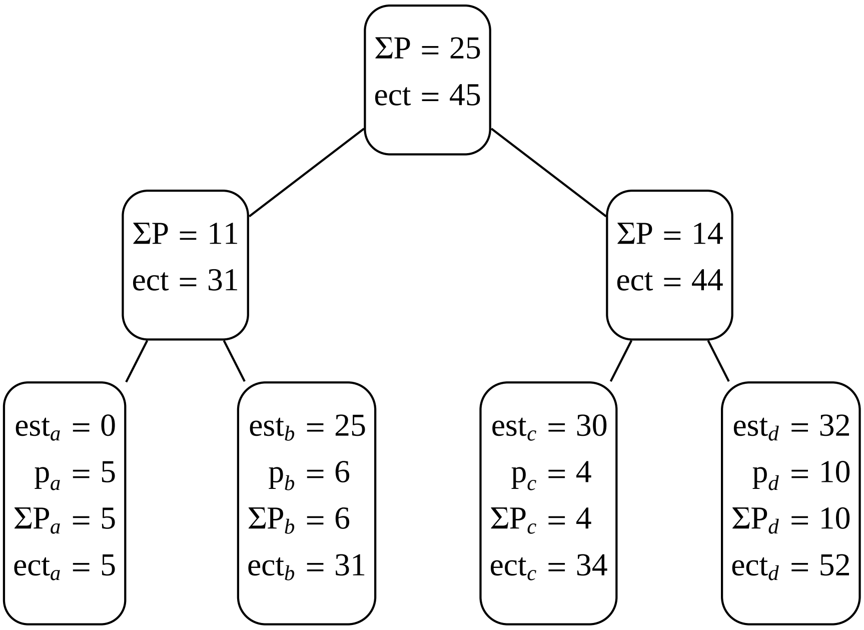

Figure 2.8: An example of a Θ-tree for $\Theta = \{a, b, c, d\}$.

Let v be an arbitrary node of the Θ-tree (an internal node or a leaf). We define 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 left(v) and right(v):

$$\Sigma\mathrm{P}_v = \Sigma\mathrm{P}_{\mathrm{left}(v)} + \Sigma\mathrm{P}_{\mathrm{right}(v)} \tag{2.20}$$

In order to compute also ect_v recursively, the activities cannot be stored in the leaves randomly, but in the non-decreasing order by est from left to right. In particular for any two activities $i, j \in \Theta$, if $\mathrm{est}_i < \mathrm{est}_j$ then the activity i is stored on the left from the activity j. Thanks to this property the following inequality holds:

$$\forall i \in \mathrm{Left}(v), \forall j \in \mathrm{Right}(v) : \quad \mathrm{est}_i \le \mathrm{est}_j \tag{2.21}$$

where Left(v) is a shortcut for Leaves(left(v)), similarly Right(v).

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

$$\mathrm{ect}_v = \max\left\{\mathrm{ect}_{\mathrm{right}(v)},\ \mathrm{ect}_{\mathrm{left}(v)} + \Sigma\mathrm{P}_{\mathrm{right}(v)}\right\} \tag{2.22}$$

Proof: From the definition (2.1), the value ect_v is:

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

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

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

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

2. Left(v) $\cap\, \Theta' \neq \emptyset$. Then $\mathrm{est}_{\Theta'} = \mathrm{est}_{\Theta' \cap \mathrm{Left}(v)}$ because of the property (2.21). Let S be the set of all possible Θ' considered in this part of the proof:

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

Then:

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

We used the fact that the maximum is achieved only by such a set Ω' for which Right(v) $\subsetneq \Omega'$.

Therefore the formula (2.22) is correct. □

Thanks to the recursive formulas (2.20) and (2.22), the values ect_v and $\Sigma\mathrm{P}_v$ can be easily computed within usual operations with a balanced binary tree without changing their time complexities. Time complexities of operations with Θ-tree are summarized in the table 2.1.

Notice that Θ-tree 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 fastest way to implement Θ-tree is to make the shape of the tree fixed during the whole computation. I.e., we start with a perfectly balanced tree that represents all activities on the resource. To indicate that an activity i is not in the set Θ it is enough to set $\Sigma\mathrm{P}_i = 0$ and $\mathrm{ect}_i = -\infty$. Clearly, these additional "empty" leaves will not interfere with the formulas (2.20) and (2.22).

Operation	**Time Complexity**
$\Theta := \emptyset$	$\mathcal{O}(1)$ or $\mathcal{O}(n \log n)$
$\Theta := \Theta \cup \{i\}$	$\mathcal{O}(\log n)$
$\Theta := \Theta \setminus \{i\}$	$\mathcal{O}(\log n)$
ect_Θ	$\mathcal{O}(1)$

Table 2.1: Worst-case time complexities of operations on Θ-tree.

2.5.3 Overload Checking

The algorithm is based on the following observation:

Observation 1 *Let* $T = \{j_1, j_2, \ldots, j_n\}$ *and* $\mathrm{lct}_{j_1} \leq \cdots \leq \mathrm{lct}_{j_n}$. *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 $\Theta = \mathrm{LCut}(T, j)$. Thanks to the ordering of the activities the set $\mathrm{LCut}(T, j)$ can be quickly recomputed from the previous set.

Algorithm 2.3: Overload Checking

```
Θ := ∅;
for $j \in T$ in non-decreasing order of $\mathrm{lct}_j$ do begin
  Θ := $\Theta \cup \{j\}$;
  if $\mathrm{ect}_\Theta > \mathrm{lct}_j$ then
    fail;  {No solution exists}
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 $\mathcal{O}(n \log n)$ – the activities have to be sorted and n-times an activity is inserted into the set Θ.

2.5.4 Detectable Precedences

Let us recall the definition of the detectable predecessors of the activity i (2.17):

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

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* $\mathrm{est}_1 + \mathrm{p}_1 \le \mathrm{est}_2 + \mathrm{p}_2 \le \cdots \le \mathrm{est}_n + \mathrm{p}_n$. *Then* $\mathrm{DPrec}'(T, i_1) \subseteq \mathrm{DPrec}'(T, i_2) \subseteq \cdots \subseteq \mathrm{DPrec}'(T, i_n)$.

The algorithm 2.4 is based on 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 $O(n \log n)$.

Algorithm 2.4: Detectable Precedences

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

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 2.3). 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 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.

2.5.5 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* $\mathrm{lct}_1 \leq \mathrm{lct}_2 \leq \cdots \leq \mathrm{lct}_n$. *Then:*

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

The idea is to compute the sets $\mathrm{NLSet}'(T, i)$ incrementally:

Algorithm 2.5: Not-Last

```
$\Theta$ := $\emptyset$;
Q := queue of all activities $j \in T$ in non-decreasing order of $\mathrm{lct}_j - \mathrm{p}_j$;
for $i \in T$ in non-decreasing order of $\mathrm{lct}_i$ do begin
  while $\mathrm{lct}_i > \mathrm{lct}_{\mathrm{Q.first}} - \mathrm{p}_{\mathrm{Q.first}}$ do begin
    $j$ := Q.first;
    $\Theta$ := $\Theta \cup \{j\}$;
    Q.dequeue;
  end;
  if $\mathrm{ect}_{\Theta \setminus \{i\}} > \mathrm{lct}_i - \mathrm{p}_i$ then
    $\mathrm{lct}'_i$ := $\min\left\{\mathrm{lct}_j - \mathrm{p}_j,\ \mathrm{lct}'_i\right\}$;
end;
```

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 $\mathcal{O}(\log n)$. Therefore the worst-case time complexity of the whole algorithm is $\mathcal{O}(n \log n)$.

Without changing the time complexity the algorithm can be slightly improved: the not-last rule can be also checked for the activity Q.first just before the insertion of the activity Q.first into the set Θ (i.e., after the line 6):

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

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

2.5.6 Θ-Λ-tree

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'):

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

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-tree. 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.

This is $\mathcal{O}(\log n)$ 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 is to compute the following value:

$$\overline{\mathrm{ect}}(\Theta, \Lambda) = \max\left(\{\mathrm{ect}_\Theta\} \cup \{\mathrm{ect}_{\Theta \cup \{i\}},\ i \in \Lambda\}\right) \tag{2.23}$$

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:

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

$\overline{\Sigma\mathrm{P}}$ is the maximum processing time of 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 the Θ-Λ-tree see figure 2.9.

The idea how to compute values $\overline{\Sigma\mathrm{P}}_v$ and $\overline{\mathrm{ect}}_v$ for internal node v follows. A gray activity can be used only once: in the left subtree of v or in the right subtree of v. Note that the gray activity used for $\overline{\Sigma\mathrm{P}}_v$ can be different from the gray activity used for $\overline{\mathrm{ect}}_v$. The formulas (2.20) and (2.22) can be modified to handle gray nodes:

$$\begin{aligned}\overline{\Sigma\mathrm{P}}_v &= \max\left\{\overline{\Sigma\mathrm{P}}_{\mathrm{left}(v)} + \Sigma\mathrm{P}_{\mathrm{right}(v)},\ \Sigma\mathrm{P}_{\mathrm{left}(v)} + \overline{\Sigma\mathrm{P}}_{\mathrm{right}(v)}\right\}\\ \overline{\mathrm{ect}}_v &= \max\{\,\overline{\mathrm{ect}}_{\mathrm{right}(v)}, && \text{(a)}\\ &\qquad \mathrm{ect}_{\mathrm{left}(v)} + \overline{\Sigma\mathrm{P}}_{\mathrm{right}(v)},\ \overline{\mathrm{ect}}_{\mathrm{left}(v)} + \Sigma\mathrm{P}_{\mathrm{right}(v)}\,\} && \text{(b)}\end{aligned}$$

The line (a) considers all sets Θ' such that $\Theta' \cap \mathrm{Left}(v) = \emptyset$ (see the definition (2.1) of ect on page 22). Line (b) considers all sets Θ' such that $\Theta' \cap \mathrm{Left}(v) \neq \emptyset$.

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:

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

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

And for internal node v:

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

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

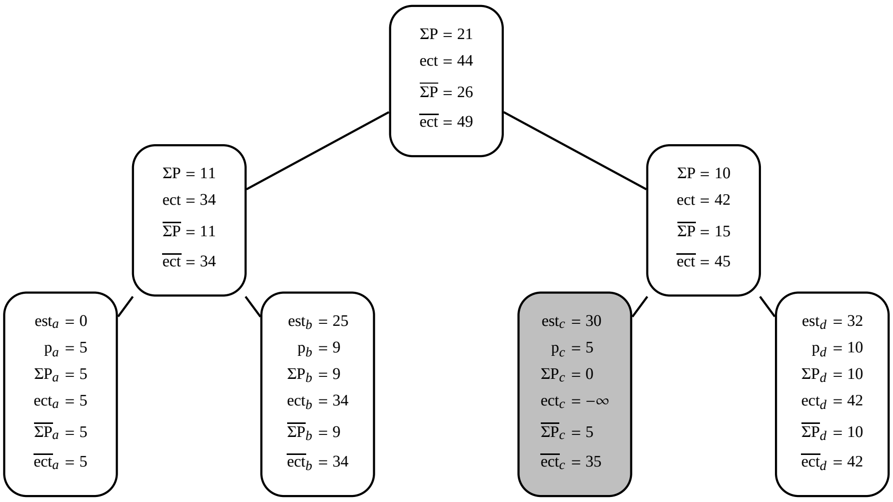

Figure 2.9: An example of a Θ-Λ-tree for $\Theta = \{a, b, d\}$ and $\Lambda = \{c\}$.

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

Table 2.2: Worst-case time complexities of operations on Θ-Λ-tree.

Thanks to these recursive formulas all these computations can be done within usual operations with balanced binary trees without changing their time complexities. Table 2.2 shows time complexities of selected operations on Θ-Λ-tree.

2.5.7 Edge Finding

By the observation 1 (page 40) we known that the sets LCut(T, j) are nested and by the property 1 (page 28) we know that for each activity i we are looking for the biggest 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 starts with $\Theta = \mathrm{LCut}(T, j_n) = T$ 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. Thanks to the property 1 the activity i cannot be updated better, therefore it can be removed from the set Λ.

Algorithm 2.6: Edge Finding

```
 1  $(\Theta, \Lambda) := (T, \emptyset)$;
 2  Q := queue of all activities $j \in T$ in non-increasing order of $\text{lct}_j$;
 3  $j$ := Q.first;
 4  while Q.size > 1 do begin
 5     if $\text{ect}_\Theta > \text{lct}_j$ then
 6        fail;  {Resource is overloaded}
 7     $(\Theta, \Lambda) := (\Theta \setminus \{j\}, \Lambda \cup \{j\})$;
 8     Q.dequeue;
 9     $j$ := Q.first;
10     while $\overline{\text{ect}}(\Theta, \Lambda) > \text{lct}_j$ do begin
11        $i$ := gray activity responsible for $\overline{\text{ect}}(\Theta, \Lambda)$;
12        $\text{est}_i := \max\{\text{est}_i, \text{ect}_\Theta\}$;
13        $\Lambda := \Lambda \setminus \{i\}$;
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. The lines 5 and 6 make the same check as 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 the table 2.2 time complexity of each single line within the loops is $O(\log n)$ maximum. 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.

2.6 Experimental Results

For our tests we took several common jobshop instances from the OR library [1]. The benchmark problem is to compute a destructive lower bound using a shaving technique [24]. A destructive lower bound is a minimum length of the schedule, for which we are not able to proof infeasibility. It is a good benchmark problem because there is no influence of a search heuristic.

Shaving is similar to the proof by a contradiction. We choose an activity i, limit its est_i or lct_i and propagate. If an infeasibility is found then the limitation was invalid and so we can decrease lct_i or increase est_i. To limit CPU time, shaving was used for each activity only once. For more details about shaving see [24].

Table 2.3 shows the results. We measured the CPU[4] time (in seconds) needed to prove the lower bound. In other words the propagation is done twice: with the upper bound LB and LB-1. Column T1 shows total running times when presented $\mathcal{O}(n \log n)$ filtering algorithms are used (overload checking, detectable precedences, not-first/not-last and edge-finding). Column T2 shows total running times when quadratic algorithms are used: quadratic overload checking, not-first/not-last from [28], edge-finding from [25] and $\mathcal{O}(n \log n)$ detectable precedences.

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

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

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

Table 2.3: Destructive Lower Bounds

Chapter 3

Optional Activities

3.1 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 a unary resource with optional activities.

3.2 History

Traditionally, resource constraints are not designed to handle optional activities properly. There are only limited ways how to model them:

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., resource constraint is not dynamic [5]). 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 added on the resource. When a fail is found then the optional activity is rejected. Otherwise bounds of the optional activity can be adjusted. [6] introduces so called PEX-Edge Finding with time complexity $\mathcal{O}(n^3)$. This is a pretty strong propagation, however rather time consuming.

Modified filtering algorithms. Regular and optional activities are treated differently: optional activities do not influence any other activity on the resource, while regular activities influence other regular activities and also optional activities [17]. Most of the filtering algorithms can be modified this way without changing their time complexities. However, this approach is a little bit weaker than the previous one, because the previous approach also checks whether the addition of a optional activity would not cause an immediate fail.

Cumulative resources. If we have a set of similar alternative machines, this set can be modeled as a cumulative resource. This additional (redundant) constraint can improve the propagation before activities are distributed among machines. There is also a special filtering algorithm [40] designed to handle this type of alternatives.

3.3 New Approach

The approach described in this chapter was published in [36, 37]. The idea is to use Θ-Λ-tree data structure to model optional activities.

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 more simple, 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, 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 over-

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

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

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

2. If the addition of an optional activity i does not result in overloading, what is the earliest possible start time and the latest 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 we add an optional activity i, will the first run of a filtering algorithm result in a fail? For example the algorithm Detectable Precedences can increase est_k of some activity k so much that $\mathrm{est}_k + \mathrm{p}_k > \mathrm{lct}_k$. In that case we can also propagate $\mathrm{exists}_i = \mathrm{false}$.

The rest of this chapter shows how to extend filtering algorithms based on Θ-tree to handle optional activities - namely Overload Checking, Not-First/Not-Last and Detectable Precedences. Efficient algorithm for Edge Finding with optional activities is a part of my future work.

3.4 Overload Checking with Optional Activities

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

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

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

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

The proposition 1 (page 24) proves 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 (2.5) from the page 24):

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

Let us say that the activity $j \in T$ is fixed and we want to find all activities o for which the rule (OL'_{oe}) propagates. Clearly $\mathrm{lct}_o \leq \mathrm{lct}_j$, 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:

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

With all these possible activities o what is the maximum value of $\mathrm{ect}_{\mathrm{LCut}(R\cup\{o\},j)}$? Recall definition (2.23) from page 43:

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

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.

The following algorithm 3.1 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 the observation 1 about nesting of the sets LCut.

Note that it is possible that at the line 11 $o = i$. In this case at the next run of the while cycle the value $\overline{\mathrm{ect}}(\Theta,\ \Lambda)$ 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 $\overline{\mathrm{ect}}(\Theta,\ \Lambda) > \mathrm{lct}_i$ 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 $\mathcal{O}(n \log n)$. The inner while loop is repeated n times maximum because each time an activity is removed from the set Λ. The outer for loop has also n iterations, time complexity of each single line is $\mathcal{O}(\log n)$ maximum (see the table 2.2).

3.5 Detectable Precedences with Optional Activities

Let us recall the original propagation rule (DP) for detectable precedences from the page 31. The rule is designed only for regular activities therefore we have to replace the set T by R:

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

Algorithm 3.1: Overload Checking with Optional Activities

```
$(\Theta,\ \Lambda)\ :=\ (\emptyset,\ \emptyset)$;
for $i \in T$ in non-decreasing order of $\mathrm{lct}_i$ do begin
  if $i$ is a optional activity then
    $\Lambda\ :=\ \Lambda \cup \{i\}$;
  else begin
    $\Theta\ :=\ \Theta \cup \{i\}$;
    if $\mathrm{ect}_\Theta > \mathrm{lct}_i$ then
      fail; {No solution exists}
  end;
  while $\overline{\mathrm{ect}}(\Theta,\ \Lambda) > \mathrm{lct}_i$ do begin
    $o$ := optional activity responsible for $\overline{\mathrm{ect}}(\Theta,\ \Lambda)$;
    $\mathrm{exists}_o$ := false;
    $\Lambda\ :=\ \Lambda \setminus \{o\}$;
  end;
end;
```

where

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

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

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

And a symmetric rule:

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

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:

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

The algorithm for the rules (DP_o) and (DP_{oe}) can be found in [36, 37]. However the rule (DP_{oe}) is not really necessary as we will see later in the proposition 8. But first let us prove the following lemma:

Lemma 2 *Let $i \in R$ be a regular activity, $o \in O$ 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:

$$\text{est}_o \geq \text{est}_i + \text{p}_i$$

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

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

Combining these two inequalities we get:

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

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 a moment when all propagation using the rules (OL_o), (DP_o) and ($\overline{\text{DP}}_o$) is done, i.e., the rule (OL_o) did not detect fail and the rules (DP_o) and ($\overline{\text{DP}}_o$) 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 on 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 $\text{DPrec}(R \cup \{o\}, i)$ that:

$$\text{est}_\Psi + \text{p}_\Psi = \text{ect}_{\text{DPrec}(R\cup\{o\},i)} \tag{3.1}$$

Note that thanks 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:

   $$\begin{aligned} \mathrm{ect}_{\mathrm{DPrec}(R\cup\{o\},i)} + \mathrm{p}_i &> \mathrm{lct}_i & \\ \mathrm{est}_o + \mathrm{p}_o + \mathrm{p}_i &> \mathrm{lct}_i & \text{by (3.1) and } \Psi = \{o\} \end{aligned}$$

   Thus the precedence $i \ll o$ is detectable. The precedence $o \ll i$ is detectable too because $\{o\} = \Psi \subseteq \mathrm{DPrec}(R \cup \{o\}, i)$. Using the 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 4) with the exception of the precedence $o \ll i$ which is propagated only on the activity o:

   $$\forall j \in \Psi : \quad \mathrm{lct}_j \leq \mathrm{lct}_i - \mathrm{p}_i \tag{3.2}$$

   $$\forall j \in \Psi \setminus \{o\} : \quad \mathrm{est}_i \geq \mathrm{est}_j + \mathrm{p}_j \tag{3.3}$$

   Therefore:

   $$\mathrm{lct}_{\Psi\cup\{i\}} = \mathrm{lct}_i \tag{3.4}$$

   $$\mathrm{est}_{\Psi\cup\{i\}} = \mathrm{est}_\Psi \tag{3.5}$$

   For the second equality we use the fact that $\Psi \neq \{o\}$ therefore there is a regular activity $j \in \Psi$ for which $\mathrm{est}_i \geq \mathrm{est}_j + \mathrm{p}_j$ by (3.3).

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

   $$\begin{aligned} \mathrm{ect}_{\mathrm{DPrec}(R\cup\{o\},i)} + \mathrm{p}_i &> \mathrm{lct}_i & \\ \mathrm{est}_\Psi + \mathrm{p}_\Psi + \mathrm{p}_i &> \mathrm{lct}_i & \text{by (3.1)} \\ \mathrm{est}_{\Psi\cup\{i\}} + \mathrm{p}_{\Psi\cup\{i\}} &> \mathrm{lct}_{\Psi\cup\{i\}} & \text{by (3.4) and (3.5)} \end{aligned}$$

   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.

□

The algorithm 3.2 is a slightly modified algorithm 2.4 to handle also optional activities using the rule (DP_o). The worst-case time complexity of the algorithm remains the same: $\mathcal{O}(n \log n)$.

Algorithm 3.2: Detectable Precedences with Optional Activities

```
$\Theta$ := $\emptyset$;
Q := queue of all regular activities $j \in R$ in non-decreasing order of $\mathrm{lct}_j - \mathrm{p}_j$;
for $i \in T$ in non-decreasing order of $\mathrm{est}_i + \mathrm{p}_i$ do begin
   while $\mathrm{est}_i + \mathrm{p}_i > \mathrm{lct}_{\mathrm{Q.first}} - \mathrm{p}_{\mathrm{Q.first}}$ do begin
      $\Theta$ := $\Theta \cup \{\mathrm{Q.first}\}$;
      Q.dequeue;
   end;
   $\mathrm{est}'_i$ := $\max\{\mathrm{est}_i,\ \mathrm{ect}_{\Theta\setminus\{i\}}\}$;
end;
for $i \in T$ do
   $\mathrm{est}_i$ := $\mathrm{est}'_i$;
```

3.6 Not-First/Not-Last with Optional Activities

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

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

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

1. Regular activities also influence optional activities:

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

2. If an optional activity o becomes a regular activity then the rule not-last may result in immediate fail 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:

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

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 a moment 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 moment 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:

   $$\begin{aligned} \text{est}_o + \text{p}_o &> \text{lct}_i - \text{p}_i \\ \text{lct}_o - \text{p}_o &< \text{est}_i + \text{p}_i \end{aligned}$$

   Therefore there are detectable precedences $i \ll o$ and $o \ll i$ and by the 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:

   $$\forall j \in \Omega : \quad \text{lct}_j \leq \text{lct}_i - \text{p}_i \tag{3.6}$$
   $$\forall j \in \Omega \setminus \{o\} : \quad \text{est}_i \geq \text{est}_j + \text{p}_j \tag{3.7}$$

   Therefore:

   $$\text{lct}_{\Omega\cup\{i\}} = \text{lct}_i \tag{3.8}$$
   $$\text{est}_{\Omega\cup\{i\}} = \text{est}_\Omega \tag{3.9}$$

   For the second equality we used the fact that $\Omega \neq \{o\}$ thus there is regular activity $j \in \Omega$ such that $\text{est}_i \geq \text{est}_j + \text{p}_j$ by 3.7.

   The rule (NL$_{oe}$) propagates, therefore:

   $$\begin{aligned} \text{est}_\Omega + \text{p}_{\Omega\cup\{i\}} &\geq \text{lct}_i \\ \text{est}_{\Omega\cup\{i\}} + \text{p}_{\Omega\cup\{i\}} &\geq \text{lct}_{\Omega\cup\{i\}} && \text{by (3.9) and (3.8)} \end{aligned}$$

   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'):

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

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 proof of proposition 3. □

The paper [37] presents an algorithm for propagation 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 3.3 which propagates only according to the rule (NL$_{oe}$). It is an easy modification of the algorithm 2.5. Time worst-case time complexity remains $\mathcal{O}(n \log n)$.

Algorithm 3.3: Not-Last with Optional Activities

```
Θ := ∅;
Q := queue of all activities $j \in R$ in non-decreasing order of $\mathrm{lct}_j - \mathrm{p}_j$;
for $i \in T$ in non-decreasing order of $\mathrm{lct}_i$ do begin
  while $\mathrm{lct}_i > \mathrm{lct}_{\mathrm{Q.first}} - \mathrm{p}_{\mathrm{Q.first}}$ do begin
    j := Q.first;
    Θ := $\Theta \cup \{j\}$;
    Q.dequeue;
  end;
  if $\mathrm{ect}_{\Theta \setminus \{i\}} > \mathrm{lct}_i - \mathrm{p}_i$ then
    $\mathrm{lct}'_i$ := $\min\left\{\mathrm{lct}_j - \mathrm{p}_j,\ \mathrm{lct}'_i\right\}$;
end;
```

3.7 Experimental Results

Optional activities were tested on modified 10x10 jobshop instances from OR Library [1]. In each job, activities on 5th and 6th place were taken as alternatives. Therefore in each problem there are 20 optional activities and 80 regular activities. Table 3.1 shows the results. Column LB is the destructive lower bound computed by propagation only, column Opt is the optimal makespan. Column CH is the number of choicepoints needed to find the optimal solution and prove the optimality (i.e., optimal makespan used as the initial upper bound). Finally the column T is the CPU[1] time in seconds.

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

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

Table 3.1: Experimental results for alternative activities

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

Chapter 4

Batch processing with Setup Times

4.1 Introduction

Unary resources are often not sufficient to model real-life problems. In this chapter we focus on a kind of resource which differs from the unary resource in two aspects:

1. The resource is able to process more than one activity at a time. However in this case the overlapping activities form a batch - they start and complete together.
2. Once a batch is completed the resource needs a setup before the next batch can start. The duration of the setup may depend on the type of activities in the two adjacent batches.

Such environment is frequent in real-life scheduling problems where batches describe a collection of activities processed together, e.g., in the pool, and the setup times describe the time necessary to prepare the resource for the next batch, e.g., to clean up the pool. Batch processing and setup times impose additional constraints to the problem and these constraints can be used to prune the search space.

In some sense batch processing can be seen as a mixture of cumulative and disjunctive scheduling. The filtering algorithms for cumulative scheduling are weak when applied to batch processing because they do not use information about batches. The stronger filtering algorithms for disjunctive scheduling cannot be applied directly to batch scheduling because of parallel activities in the batch. Therefore we extended existing algorithms for disjunctive scheduling and design new ones, namely:

- Edge Finding
- Not-First/Not-Last
- Not-Before/Not-After

- Precedence Graph
- Sequence Composition

All these algorithms are presented in this chapter. The exception is the algorithm Sequence Composition which can be found in [35].

4.2 Basic Notions

The following definition of batch processing is close to p-batching in traditional scheduling [8], i.e., two overlapping activities in the resource must start at the same time and they must finish at the same time. The activities processed together form a batch.

We can restrict the number of activities processed together in a single batch using capacity and compatibility restrictions (Figure 4.1). Let T be a set of all activities on the resource. Each activity $i \in T$ has two attributes: c_i indicating the capacity consumed by the activity and f_i indicating the type of the activity (family). The compatibility restriction requires that only activities of the same family can be processed together. The capacity restriction limits the total capacity of each batch by a constant C – the capacity of the resource[1].

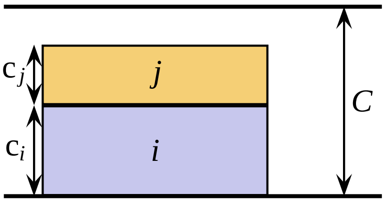

Figure 4.1: Activities of the same family ($\mathrm{f}_i = \mathrm{f}_j$) can be processed simultaneously.

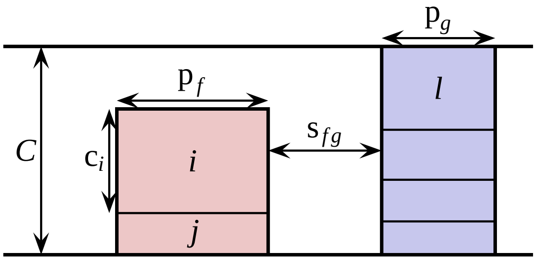

Figure 4.2: Example of two consecutive batches with families f and g.

[1] Renewable resource is assumed, i.e., the capacity is consumed only during the processing of the activity.

Because all activities processed in a batch start and complete together, they have also identical duration. Thus we can assign the duration attribute to the family rather than to a particular activity. Formally let F denotes the set of all families – $F = \{\mathrm{f}_i,\ i \in T\}$. Then p_f denotes the duration (processing time) of every activity of family f.

A setup time must be inserted between each pair of consecutive batches. Because the setup time depends on types of both batches we say that the setup time is sequence dependent. Formally s_{fg} denotes the setup time between batches of family f and family g. No setup is assumed between batches of the same family:

$$\forall f \in F : \quad \mathrm{s}_{ff} = 0 \tag{4.1}$$

Alike [9] we assume that the setup time satisfies the triangular inequality:

$$\forall f, g, h \in F : \quad \mathrm{s}_{fh} \leq \mathrm{s}_{fg} + \mathrm{s}_{gh} \tag{4.2}$$

Not all resources fulfill this inequality. In this case we would not be able to estimate setup times for a set of activities so accurately and the following algorithms would be weaker.

Finally we extend the notation of the processing time to activities as well:

$$\mathrm{p}_i = \mathrm{p}_{\mathrm{f}_i}$$

Earliest starting time est_i and latest completion time lct_i are defined in the same way as for the unary resource.

Let k be the number of families, $k = |F|$. The following filtering algorithms are polynomial in k and n – their time complexity is $\mathcal{O}(kn^2)$. However they require some preprocessing of setup times which has exponential time complexity $\mathcal{O}(k^2 2^k)$. Therefore the key assumption of the following algorithms is that k is much smaller than number of activities $n = |T|$. Note also that the preprocessing can be done before the search starts, it is not necessary to repeat it in each search node.

Setup Time Preprocessing

The previous paragraphs introduced the setup time between two families of activities. Now we extend this notion to bigger sets of families. In particular, let $\phi \subseteq F$ be a set of activity families:

- $\mathrm{s}(\phi)$ denotes the minimal setup time for the activities of families ϕ.
- $\mathrm{s}(f, \phi)$ denotes the same as $\mathrm{s}(\phi)$ but with the additional restriction that the processing starts with the family f.

- s(ϕ, f) likewise, but the processing ends with the family f.

Unlike [9] we propose to compute setup times for different set of families before the scheduling starts. We identify each set of families using a bitmap, i.e., each set is represented by a number. Thus the values of the these attributes can be saved in an array with the access time $O(1)$. We will show that these values can be computed in the time complexity $O(k^2 2^k)$.

Visibly:

$$\forall \phi \subseteq F: \quad \mathrm{s}(\phi) = \min\{\mathrm{s}(f, \phi),\ f \in \phi\} \tag{4.3}$$

Thus once we compute the function $s(f, \phi)$, the function s(ϕ) can be computed using the previous formula in $O(k2^k)$.

For the sets ϕ with just one family, there are no setups used according to (4.1):

$$\forall f \in F: \quad s(f, \{f\}) = 0$$

We can compute the value $s(f, \{f\} \cup \phi)$ from the value s(g, ϕ) using the following recursive formula (note that it requires the triangular inequality (4.2)):

$$\forall \phi \subset F,\ \forall f \in (F \setminus \phi): \quad \mathrm{s}(f, \{f\} \cup \phi) = \min\{\mathrm{s}_{fg} + \mathrm{s}(g, \phi),\ g \in \phi\}$$

The worst-case time complexity of this computation is $O(k^2 2^k)$. The function s(ϕ, f) can be computed symmetrically.

Consolidated Processing Time

In the following algorithms we will often need to compute the lower bound of the processing time needed for a subset of activities. Let $\Omega \subseteq T$ be a set of activities, c(Ω, f) denotes the total capacity of the activities from Ω of the family f:

$$\mathrm{c}(\Omega, f) = \sum_{\substack{i \in \Omega \\ \mathrm{f}_i = f}} \mathrm{c}_i$$

Let u(Ω, f) denotes the minimal time needed for processing of all activities of the family f from Ω. This value is computed as a multiplication of the minimal number of required batches and the processing time of the family f:

$$\mathrm{u}(\Omega, f) = \left\lceil \frac{\mathrm{c}(\Omega, f)}{C} \right\rceil \mathrm{p}_f$$

Let F_Ω be the set of all families in the set Ω – $F_\Omega = \{\mathrm{f}_i,\ i \in \Omega\}$. The total processing time u(Ω) of the set Ω without setups is:

$$\mathrm{u}(\Omega) = \sum_{f \in F_\Omega} \mathrm{u}(\Omega, f)$$

Finally the consolidated processing time $p(\Omega)$ for the set of activities Ω consists of the pure processing time and the setup times:

$$\mathrm{p}(\Omega) = \mathrm{s}(F_\Omega) + \mathrm{u}(\Omega)$$

If we know that the processing of the set Ω has to start or to end with one particular activity $j \in \Omega$ we can estimate the processing time more precisely:

$$\begin{aligned} \mathrm{p}(j, \Omega) &= \mathrm{s}(\mathrm{f}_j, F_\Omega) + \mathrm{u}(\Omega) \\ \mathrm{p}(\Omega, j) &= \mathrm{s}(F_\Omega, \mathrm{f}_j) + \mathrm{u}(\Omega) \end{aligned}$$

Once again, the earliest starting time est_Ω and the latest completion time lct_Ω of the set Ω is defined the same way as on unary resource:

$$\begin{aligned} \mathrm{lct}_\Omega &= \max\{\mathrm{lct}_i,\ i \in \Omega\} \\ \mathrm{est}_\Omega &= \min\{\mathrm{est}_i,\ i \in \Omega\} \end{aligned}$$

4.3 Filtering Algorithms

This section presents several filtering algorithms for batch processing with sequence dependent setup times. As usual each algorithm removes a different type of inconsistent values therefore they can be used together to achieve better pruning.

4.3.1 Overload Checking

The overload rule for the batch processing stays exactly the same as the rule (OL) for the unary resource constraint. The only difference is in the computation of the value p(Ω).

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

The rule says that the problem has no solution if there is a set of activities Ω which cannot be processed within its bounds.

Fortunately it is not necessary to check this rule for every possible subset Ω of T. It is enough to perform the check only for those sets Ω which form so-called task intervals:

Definition 6 *Let $i, j \in T$ be two activities such that* $\mathrm{est}_i \le \mathrm{est}_j$ *and* $\mathrm{lct}_i \le \mathrm{lct}_j$*. Case $i = j$ is also possible. A* task interval $[i, j]$ *is the following set of activities:*

$$[i, j] = \left\{k,\ k \in T\ \&\ \mathrm{est}_k \ge \mathrm{est}_i\ \&\ \mathrm{lct}_k \le \mathrm{lct}_j\right\}$$

The notation $[i, j]$ is adopted from [12, 13, 14].

We will show that in the rule (OL_s) it is necessary to consider only sets Ω which are time intervals:

$$\forall i, j \in T : \quad \mathrm{lct}_{[i,j]} - \mathrm{est}_{[i,j]} < \mathrm{p}([i, j]) \Rightarrow \mathrm{fail} \qquad (\mathrm{OL}'_s)$$

Proposition 11 *The rules (OL_s) and (OL'_s) are equivalent.*

Proof: It is necessary to prove that by considering smaller number of sets Ω we do not lose any filtering, i.e., if there is a set Ω such that the rule (OL_s) detects infeasibility then there is also some time interval $[i, j]$ such that the new rule (OL'_s) does the same.

Let the rule (OL_s) hold for a set $\Omega \subseteq T$. Consider a set Ψ defined the following way:

$$\Psi = \{k,\ k \in T \ \&\ \mathrm{est}_k \geq \mathrm{est}_\Omega \ \&\ \mathrm{lct}_k \leq \mathrm{lct}_\Omega\}$$

Clearly Ψ is a task interval. Moreover:

$$\begin{aligned} \mathrm{est}_\Psi &= \mathrm{est}_\Omega \\ \mathrm{lct}_\Psi &= \mathrm{lct}_\Omega \\ \Psi &\supseteq \Omega \\ \mathrm{u}(\Psi) &\geq \mathrm{u}(\Omega) \\ F_\Psi &\supseteq F_\Omega \\ \mathrm{s}(F_\Psi) &\geq \mathrm{s}(F_\Omega) \\ \mathrm{p}(\Psi) &\geq \mathrm{p}(\Omega) \end{aligned}$$

Therefore the rule (OL'_s) holds for the set Ψ. □

The algorithm 4.1 checks the rule (OL'_s) for all task intervals. Note that at on the line 9 the set Ω is a task interval only when $\mathrm{est}_i \leq \mathrm{est}_j$. However in any case $\mathrm{est}_\Omega = \mathrm{est}_i$ and $\mathrm{lct}_\Omega \leq \mathrm{lct}_j$ therefore these additional checks at the line 9 make no harm[2]. This is a reoccurring pattern which we will see again in the following algorithms. But we will not draw attention to it anymore.

The worst-case time complexity of the algorithm is $O(n^2)$.

4.3.2 Not-Before/Not-After

Consider an arbitrary set $\Omega \subset T$ and an activity $i \notin \Omega$. If we schedule the activity i before the activities Ω, then processing of the set Ω can start at first in the time

[2] Note that we cannot just ignore activities i with $\mathrm{est}_i > \mathrm{est}_j$ because they must be also included in the set Ω (if $\mathrm{lct}_i \leq \mathrm{lct}_j$). In practice the algorithm can be implemented more effectively but that would unnecessarily complicate the presentation of the main idea.

Algorithm 4.1: Overload Checking for Batch Processing

```
for $j \in T$ do begin
  $\Omega := \emptyset$;
  for $i \in T$ in non-increasing order of $\mathrm{est}_i$ do begin
    if $\mathrm{lct}_i > \mathrm{lct}_j$ then
      continue;
    $\Omega := \Omega \cup \{i\}$;
    if $\mathrm{lct}_j - \mathrm{est}_i < \mathrm{p}(\Omega)$ then fail;
  end;
end;
```

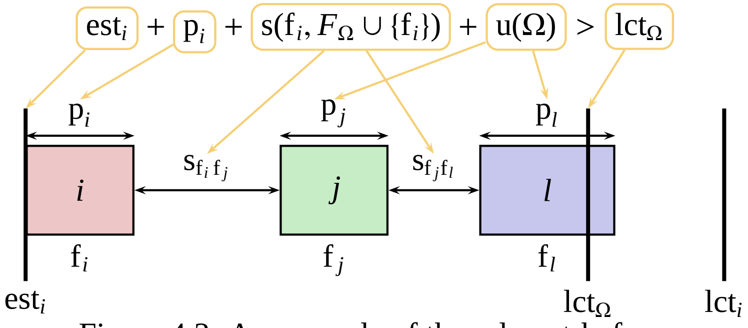
est_i + p_i + $\mathrm{s}(\mathrm{f}_i, F_\Omega \cup \{\mathrm{f}_i\})$ + $\mathrm{u}(\Omega)$ > lct_Ω
p_i
p_j
p_l
$\mathrm{s}_{\mathrm{f}_i \mathrm{f}_j}$
$\mathrm{s}_{\mathrm{f}_j \mathrm{f}_l}$
i
j
l
f_i
f_j
f_l
est_i
lct_Ω
lct_i

Figure 4.3: An example of the rule not-before.

$\mathrm{est}_i + \mathrm{p}_i$. Processing of activities in Ω needs time $\mathrm{u}(\Omega) + \mathrm{s}(\mathrm{f}_i, F_\Omega \cup \{\mathrm{f}_i\})$ because the resource is already adjusted for processing of family f_i. If such a schedule is not possible (i.e. processing of Ω ends after lct_Ω) then the activity i cannot be scheduled before Ω:

$$\forall \Omega \subset T, \forall i \in (T \setminus \Omega) : \quad \mathrm{est}_i + \mathrm{p}_i + \mathrm{u}(\Omega) + \mathrm{s}(\mathrm{f}_i, F_\Omega \cup \{\mathrm{f}_i\}) > \mathrm{lct}_\Omega \quad \Rightarrow \quad i \not\ll \Omega$$

When the activity i cannot be processed before the activities in Ω then processing of the activity i can start at first together with the first activity from Ω[3]:

$$i \not\ll \Omega \quad \Rightarrow \quad \mathrm{est}_i := \max\{\mathrm{est}_i, \mathrm{est}_\Omega\}$$

The whole rule not-before is:

$$\forall \Omega \subset T, \forall i \in (T \setminus \Omega) :$$
$$\mathrm{est}_i + \mathrm{p}_i + \mathrm{u}(\Omega) + \mathrm{s}(\mathrm{f}_i, F_\Omega \cup \{\mathrm{f}_i\}) > \mathrm{lct}_\Omega \quad \Rightarrow \quad \mathrm{est}_i := \max\{\mathrm{est}_i, \mathrm{est}_\Omega\} \qquad \text{(NB)}$$

[3] The new value of est_i could be set even better, for example in the case when there is no activity in Ω with the family with f_i. However such situation is detected by the rule not-first (NF_s) which will be given on page 74. Therefore such improvement of the rule not-before can save some time but would not improve the resulting fixpoint.

There is also a symmetric rule not-after:

$$\begin{aligned}&\forall \Omega \subset T, \forall i \in (T \setminus \Omega) :\\ &\qquad \mathrm{lct}_i - \mathrm{p}_i - \mathrm{u}(\Omega) - \mathrm{s}(F_\Omega \cup \{\mathrm{f}_i\}, \mathrm{f}_i) < \mathrm{est}_\Omega \quad \Rightarrow \quad \mathrm{lct}_i := \min\{\mathrm{lct}_i, \mathrm{lct}_\Omega\} \quad \text{(NA)}\end{aligned}$$

The idea behind both rules is the same therefore in the following we will deal only with the rule (NB).

Again it is enough to check the rule only for task intervals:

$$\begin{aligned}&\forall j, l \in T, \forall i \in (T \setminus [j, l]) :\\ &\qquad \mathrm{est}_i + \mathrm{p}_i + \mathrm{u}([j, l]) + \mathrm{s}(\mathrm{f}_i, F_{[j,l]} \cup \{\mathrm{f}_i\}) > \mathrm{lct}_{[j,l]}\\ &\qquad\qquad \Rightarrow \quad \mathrm{est}_i := \max\left\{\mathrm{est}_i, \mathrm{est}_j\right\} \quad \text{(NB}'\text{)}\end{aligned}$$

Proposition 12 *The rules (NB) and (NB′) are equivalent.*

Proof: Analogous to the proof of proposition 11. Let the rule (NB) holds for a set $\Omega \subseteq T$ and an activity $i \in T \setminus \Omega$. Let the set Ψ be defined the following way:

$$\Psi = \{k,\ k \in T \ \&\ \mathrm{est}_k \geq \mathrm{est}_\Omega \ \&\ \mathrm{lct}_k \leq \mathrm{lct}_\Omega\}$$

Note that $i \notin \Psi$ because otherwise $\mathrm{est}_i \geq \mathrm{est}_\Omega$ and the rule (NB) achieves nothing.

The set Ψ is a task interval, the rule (NB′) holds for it and propagates at least the same as (NB) for the set Ω. □

Moreover we do not have to check the condition of the not-before rule (NB′) for each task interval separately. The following proposition gives a hint how to do it more efficiently.

Observation 4 *Let i be an activity of the family g and j be another arbitrary activity. The rule (NB′) sets* est_i *to* est_j *if and only if:*

$$\mathrm{est}_i > \min\left\{\mathrm{lct}_{[j,l]} - \mathrm{s}(g, F_{[j,l]} \cup \{g\}) - \mathrm{u}([j, l]) - \mathrm{p}_g,\ l \in T\right\} \tag{4.4}$$

The reason is that the inequality (4.4) holds if and only if there is a task interval $[j, l]$ such that (NB′) holds.

Notice that the right side of the inequality (4.4) is independent of a particular activity i. It depends only on the family g of the activity i so we can compute the value of the right side just once and then use the result for all activities of the family g. The algorithm 4.2 does exactly that. The worst-case time complexity of this algorithm is $O(kn^2)$.

Algorithm 4.2: Not-Before/Not-After

```
for $g \in F$ do begin
  for $j \in T$ do begin
    m := $\infty$;
    $\Omega$ := $\emptyset$;
    for $l \in T$ in non-decreasing order of $\mathrm{lct}_l$ do begin
      if $\mathrm{est}_l < \mathrm{est}_j$ then
        continue;
      $\Omega$ := $\Omega \cup \{l\}$;
      m := $\min\{\mathrm{lct}_l - \mathrm{s}(g, F_\Omega \cup \{g\}) - \mathrm{u}(\Omega) - \mathrm{p}_g, m\}$;
    end;
    for $i \in T$ such that $\mathrm{f}_i = g$ and $m < \mathrm{est}_i < \mathrm{est}_j$ do
      $\mathrm{est}_i$ := $\mathrm{est}_j$;
  end;
end;
```

4.3.3 Edge Finding

Like the rules not-before/not-after, the edge finding rule is trying to find a relative position of the activity i in respect to the set of activities Ω. Now we are asking whether the activity i can be processed together with the activities in Ω. If the answer is no then i must be processed either before or after all the activities in Ω. Consider again an arbitrary set $\Omega \subset T$ and an activity $i \notin \Omega$. If there is not enough time for processing activities $\Omega \cup \{i\}$ in the interval $\langle \mathrm{est}_\Omega, \mathrm{lct}_\Omega \rangle$, then the activity i has to be scheduled before or after all the activities from Ω:

$$\forall \Omega \subset T, \forall i \in (T \setminus \Omega) : \quad \mathrm{lct}_\Omega - \mathrm{est}_\Omega < \mathrm{p}(\Omega \cup \{i\}) \Rightarrow (\Omega \ll i \text{ or } i \ll \Omega) \quad (4.5)$$

Now we need to decide if the activity i has to be scheduled before Ω or after it. For that decision we can use the not-before and not-after rules. If the not-before rule (NB) holds then $\Omega \ll i$ and est_i can be changed:

$$\Omega \ll i \quad \Rightarrow$$
$$\mathrm{est}_i := \max\{\mathrm{est}_i,\ \max\{\mathrm{est}_{\Omega'} + \mathrm{u}(\Omega') + \mathrm{s}(F_{\Omega'} \cup \{\mathrm{f}_i\}, \mathrm{f}_i),\ \Omega' \subseteq \Omega\}\} \quad (4.6)$$

The whole edge finding rule is:

$$\forall \Omega \subset T, \forall i \in (T \setminus \Omega) :$$
$$\mathrm{lct}_\Omega - \mathrm{est}_\Omega < \mathrm{p}(\Omega \cup \{i\})$$
$$\& \quad \mathrm{est}_i + \mathrm{p}_i + \mathrm{u}(\Omega) + \mathrm{s}(\mathrm{f}_i, F_\Omega \cup \{\mathrm{f}_i\}) > \mathrm{lct}_\Omega$$
$$\Rightarrow \mathrm{est}_i := \max\{\mathrm{est}_i,\ \max\{\mathrm{est}_{\Omega'} + \mathrm{u}(\Omega') + \mathrm{s}(F_{\Omega'} \cup \{\mathrm{f}_i\}, \mathrm{f}_i),\ \Omega' \subseteq \Omega\}\} \quad (\mathrm{EF}_s)$$

There is also a symmetric rule updating lct_i.

Note that the rule (EF_s) is stronger than the following simple rewrite of the rule (EF) for unary resources:

$$\mathrm{lct}_\Omega - \mathrm{est}_{\Omega\cup\{i\}} < \mathrm{p}(\Omega \cup \{i\}) \quad\Rightarrow\quad \Omega \ll i$$

The rule (NB) prevents the activity i to form a batch with another activity from the set Ω. That is not the case for the simpler rule above because it allows such a batch even before est_Ω what is obviously not possible.

Once again it is enough to consider only sets Ω in the form of task intervals without lost any filtering power:

$$\begin{aligned}
&\forall j, l \in T, \forall i \in (T \setminus [j,l]) : \\
&\qquad\qquad \mathrm{lct}_{[j,l]} - \mathrm{est}_{[j,l]} < \mathrm{p}([j,l] \cup \{i\}) \\
&\qquad\quad \&\ \ \mathrm{est}_i + \mathrm{p}_i + \mathrm{u}([j,l]) + \mathrm{s}(\mathrm{f}_i, F_{[j,l]} \cup \{\mathrm{f}_i\}) > \mathrm{lct}_{[j,l]} \\
&\Rightarrow\ \ \mathrm{est}_i := \max\left\{\mathrm{est}_i,\ \max\left\{\mathrm{est}_{\Omega'} + \mathrm{u}(\Omega') + \mathrm{s}(F_{\Omega'} \cup \{\mathrm{f}_i\}, \mathrm{f}_i),\ \Omega' \subseteq [j,l]\right\}\right\} \quad (\mathrm{EF}'_s)
\end{aligned}$$

Proposition 13 *If the resource is not overloaded then the rules (EF_s) and (EF'_s) are equivalent.*

Proof: Analogous to the proof of propositions 11 and 12. Let the rule (EF_s) holds for an activity $i \in T$ and a set $\Omega \subseteq T \setminus \{i\}$. And let the set Ψ be defined as:

$$\Psi = \{k,\ k \in T\ \&\ \mathrm{est}_k \geq \mathrm{est}_\Omega\ \&\ \mathrm{lct}_k \leq \mathrm{lct}_\Omega\}$$

Note that $i \notin \Psi$ because otherwise the resource would be overloaded.

The set Ψ is a task interval, the rule (EF'_s) holds for it and propagates at least the same as (EF_s) for the set Ω. □

The algorithm for the rule (EF'_s) follows. It is a modification of the Edge Finding algorithm for the unary resource from [24].

Let us choose an activity l and a family g. We would like to check the rule (EF'_s) for all activities i of the family g (i.e., $\mathrm{f}_i = g$) and all task intervals $[j,l]$.

For simplicity, let us assume that the there are not duplicities in earliest starting times est_j. Let $\Omega_0 \subsetneq \Omega_1 \subsetneq \cdots \subsetneq \Omega_x$ be a queue of all the task intervals $[j,l]$ such that $\mathrm{est}_{\Omega_0} > \mathrm{est}_{\Omega_1} > \cdots > \mathrm{est}_{\Omega_x}$. In other words $\Omega_0 = [j_0,l], \Omega_1 = [j_1,l], \ldots, \Omega_x = [j_x,l]$ where $\mathrm{est}_{j_0} > \mathrm{est}_{j_1} > \cdots > \mathrm{est}_{j_x}$. Let $i_0, i_1, \ldots i_y$ denotes a queue of activities of the family g sorted in non-decreasing order by capacity c_{i_k}. In each step of the algorithm we consider an application of the edge finding rule to the pair Ω_x, i_y from the top of these queues. One of the following four cases must happen:

1. **$\mathrm{lct}_{i_y} \leq \mathrm{lct}_{\Omega_x}$.** In this case we will show that if the rule (EF_s) holds then $i \ll \Omega$ and the symmetrical algorithm edge finding together with overload checking (OL_s) deduce fail. Therefore we can ignore this case[4].

   Assume that the rule (EF'_s) holds for i_y and Ω_x. Therefore also (4.5) holds. Together with $\mathrm{lct}_{i_y} \leq \mathrm{lct}_{\Omega_x}$ it implies:

$$\mathrm{lct}_{i_y} - \mathrm{est}_{\Omega_x} \leq \mathrm{lct}_{\Omega_x} - \mathrm{est}_{\Omega_x} < \mathrm{p}(\Omega_x \cup \{i_y\})$$

   It is obvious that:

$$\mathrm{p}(\Omega_x \cup \{i_y\}) \leq \mathrm{u}(\Omega_x) + \mathrm{p}_g + \mathrm{s}(F_{\Omega_x} \cup \{g\}, g)$$

   Therefore:

$$\mathrm{lct}_{i_y} - \mathrm{est}_{\Omega_x} < \mathrm{u}(\Omega_x) + \mathrm{p}_g + \mathrm{s}(F_{\Omega_x} \cup \{g\}, g)$$

   Therefore the rule not-after (NA) holds. Together with (4.5) we get $i \ll \Omega$. This is detected by the symmetrical algorithm edge finding and lct_i is updated to:

$$\mathrm{lct}_{i_y} := \min\left\{\mathrm{lct}_{i_y},\ \mathrm{lct}_{\Omega_x} - \mathrm{u}(\Omega_x) - \mathrm{s}(g, F_{\Omega_x} \cup \{g\})\right\}$$

   Thus:

$$\mathrm{lct}_{i_y} \leq \mathrm{lct}_{\Omega_x} - \mathrm{u}(\Omega_x) - \mathrm{s}(g, F_{\Omega_x} \cup \{g\})$$

   The not-before rule (which is a part of (EF'_s)) still holds for i_y and Ω_x therefore:

$$\mathrm{est}_{i_y} > \mathrm{lct}_{\Omega_x} - \mathrm{p}_g - \mathrm{u}(\Omega_x) - \mathrm{s}(g, F_{\Omega_x} \cup \{g\})$$

   Now consider the set $\Psi = \{i_y\}$:

$$\begin{aligned}\mathrm{lct}_\Psi - \mathrm{est}_\Psi &= \mathrm{lct}_{i_y} - \mathrm{est}_{i_y} \leq \\ &\leq (\mathrm{lct}_{\Omega_x} - \mathrm{u}\,(\Omega_x) - \mathrm{s}\,(g, F_{\Omega_x} \cup \{g\})) - \mathrm{est}_{i_y} < \\ &< (\mathrm{lct}_{\Omega_x} - \mathrm{u}\,(\Omega_x) - \mathrm{s}\,(g, F_{\Omega_x} \cup \{g\})) - \\ &\quad \left(\mathrm{lct}_{\Omega_x} - \mathrm{p}_g - \mathrm{u}\,(\Omega_x) - \mathrm{s}\,(g, F_{\Omega_x} \cup \{g\})\right) = \\ &= \mathrm{p}_g = \mathrm{p}_\Psi\end{aligned}$$

   Hence $\mathrm{lct}_\Psi - \mathrm{est}_\Psi < \mathrm{p}_\Psi$ and the rule (OL_s) deduces fail.

   We have shown that in the case $\mathrm{lct}_{i_y} \leq \mathrm{lct}_{\Omega_x}$ the rule (EF'_s) can be ignored. And because $\mathrm{lct}_{\Omega_0} = \mathrm{lct}_{\Omega_1} = \cdots = \mathrm{lct}_{\Omega_x}$ the rule (EF'_s) can be ignored also

[4] Similarly the symmetrical algorithm for $j \ll \Omega$ ignores the case when $\mathrm{est}_{i_y} \geq \mathrm{est}_{\Omega_x}$. So the pair Ω_x and i_y is ignored by both sides of the Edge Finding algorithm when $\mathrm{lct}_{i_y} \leq \mathrm{lct}_{\Omega_x}$ and $\mathrm{est}_{i_y} \geq \mathrm{est}_{\Omega_x}$. And in this case case we cannot apply the rule (EF'_s) at all because $i_y \in \Omega_x$.

for the activity i_y and all sets $\Omega_0, \Omega_1, \ldots, \Omega_x$. Hence we can remove i_y from the queue $i_0, i_1, \ldots, i_y$.

In the following three cases we expect that $\mathrm{lct}_{i_y} > \mathrm{lct}_{\Omega_x}$ and therefore $i_y \notin \Omega_x$.

2. **The set Ω_x and the activity i_y do not satisfy the condition (4.5)** therefore the activity i_y can be processed together with Ω_x. Any activity $i_0, i_1, \ldots, i_{y-1}$ can be processed together with Ω_x as well because it requires the same or less capacity than the activity i_y. Hence we can remove the set Ω_x from the queue $\Omega_0, \Omega_1, \ldots, \Omega_x$.

3. **The set Ω_x and the activity i_y do not satisfy the condition (NB)** therefore the activity i_y can be processed before the Ω_x. In this case the activity i_y can be processed before any set $\Omega_0, \Omega_1, \ldots, \Omega_{x-1}$. Hence we can remove the activity i_y from the queue $i_0, i_1, \ldots i_y$.

4. **The set Ω_x and the activity i_y satisfy both conditions (NB) and (4.5).** Then we can change value est_{i_y} using the rule (EF'_s). This new value cannot be further improved by any other set $\Omega_0, \Omega_1, \ldots, \Omega_{x-1}$ because of the form of the rule (EF'_s). Hence we can remove i_y from the queue $i_0, i_1, \ldots, i_y$.

The above analysis explains what pairs of activity and task interval needs to be checked. We have shown that only the case 4 above contributes to change of the value est_{i_y} using the rule (EF'_s). When we want to change the value est_{i_y} according to the rule (EF'_s) then we need to know the following value z_x:

$$z_x = \max\{\mathrm{est}_{\Omega'} + \mathrm{u}(\Omega') + \mathrm{s}(F_{\Omega'} \cup \{g\}, g),\ \Omega' \subseteq \Omega_x\}$$

Now we will show how to compute the value z_x efficiently.

Consider an arbitrary set $\Omega' \subseteq \Omega_x$ and the set $\Phi = \{k,\ k \in \Omega_x\ \&\ \mathrm{est}_k \geq \mathrm{est}_{\Omega'}\}$. It is obvious that:

$$\mathrm{est}_{\Omega'} + \mathrm{u}(\Omega') + \mathrm{s}(F_{\Omega'} \cup \{g\}, g) \leq \mathrm{est}_{\Phi} + \mathrm{u}(\Phi) + \mathrm{s}(F_{\Phi} \cup \{g\}, g)$$

So we can choose only the sets Ω' in the form of a task interval $[j, l]$ such that $\mathrm{lct}_l = \mathrm{lct}_{\Omega_x}$. These task intervals are exactly the sets $\Omega_0, \Omega_1, \ldots, \Omega_x$. Hence:

$$z_x = \max\{\mathrm{est}_{\Omega'} + \mathrm{u}(\Omega') + \mathrm{s}(F_{\Omega'} \cup \{g\}, g),\ \Omega' \in \{\Omega_0, \Omega_1, \ldots, \Omega_x\}\}$$

Therefore we can compute z_x recursively using the following formula:

$$\begin{aligned} z_0 &= -\infty \\ z_x &= \max\{z_{x-1},\ \mathrm{est}_{\Omega_x} + \mathrm{u}(\Omega_x) + \mathrm{s}(F_{\Omega_x} \cup \{g\}, g)\} \end{aligned}$$

The above analysis was used for the algorithm 4.3. The algorithm assumes that activities are sorted in non-increasing order by est_j and also by values of c_j, i.e., two sorted lists of activities are prepared in advance. The worst-case time complexity of the algorithm is $O(kn^2)$.

Algorithm 4.3: Edge Finding for Batch Processing

```
for $g \in F$ do begin
  for $l \in T$ do begin
    $\Omega_{-1}$ := $\emptyset$;
    $z_{-1}$ := $-\infty$;
    x := 0;
    for $j \in T$ in the non-increasing order of $\mathrm{est}_j$ do begin
      if $\mathrm{lct}_j > \mathrm{lct}_l$ then
        continue;
      $\Omega_x$ := $\Omega_x \cup \{j\}$;
      $z_x$ := max($z_{x-1}$, $\mathrm{est}_j + \mathrm{u}(\Omega_x) + \mathrm{s}(F_{\Omega_x} \cup \{g\})$);
      x := x+1;
    end;
    $Q\Omega$ := queue of time intervals $[j, l]$ sorted by decreasing $\mathrm{est}_j$;
    $Qi$ := queue of activities with family $g$ sorted by non-increasing $\mathrm{c}_i$;
    while $Q\Omega$ is not empty and $Qi$ is not empty do begin
      y := $Qi$.top;
      $\Omega_x$ := $Q\Omega$.top;
      if $\mathrm{lct}_y \leq \mathrm{lct}_l$ then
        // case 1
        $Qi$.dequeue;
      else if $\mathrm{lct}_l - \mathrm{est}_{\Omega_x} \geq \mathrm{p}(\Omega_x \cup \{y\})$ then
        // case 2
        $Q\Omega$.dequeue;
      else if $\mathrm{lct}_l - \mathrm{est}_y \geq \mathrm{s}(g, F_{\Omega_x} \cup \{g\}) + \mathrm{u}(\Omega_x) + \mathrm{p}_g$ then
        // case 3
        $Qi$.dequeue;
      else begin
        // case 4
        $\mathrm{est}_y$ := max($\mathrm{est}_y$, $z_x$);
        $Qi$.dequeue;
      end;
    end;
  end;
end;
```

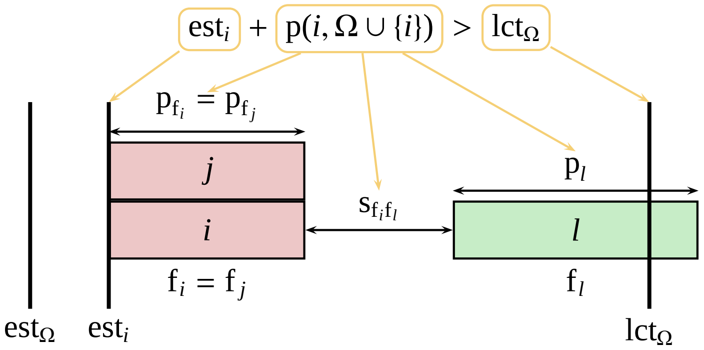
est_i + $\mathrm{p}(i, \Omega \cup \{i\})$ > lct_Ω
$\mathrm{p}_{\mathrm{f}_i} = \mathrm{p}_{\mathrm{f}_j}$
j
i
$\mathrm{s}_{\mathrm{f}_i \mathrm{f}_l}$
p_l
l
$\mathrm{f}_i = \mathrm{f}_j$
f_l
est_Ω
est_i
lct_Ω

Figure 4.4: An example of the rule not-first

4.3.4 Not-First/Not-Last

The rule not-first is based on the following idea. Consider an arbitrary set $\Omega \subsetneq T$ and an activity $i \in T \setminus \Omega$. Let us suppose that the activity i is processed before the set Ω or in the first batch of the set Ω. Then the processing of the activities $\Omega \cup \{i\}$ can start at first at est_i and has to be completed at latest at lct_Ω. If there is not enough time for such processing then the assumption was wrong and the activity i cannot be processed before the first batch of the set Ω completes:

$$\mathrm{est}_i + \mathrm{p}(i, \Omega \cup \{i\}) > \mathrm{lct}_\Omega \;\Rightarrow\; i \nless \Omega \tag{4.7}$$

The fact that $i \nless \Omega$ allows to adjust est_i:

$$i \nless \Omega \;\Rightarrow\; \mathrm{est}_i := \max\left\{\mathrm{est}_i,\ \min\left\{\mathrm{est}_j + \mathrm{p}_j + \mathrm{s}_{\mathrm{f}_j \mathrm{f}_i},\ j \in \Omega\right\}\right\} \tag{4.8}$$

The full not-first rule is:

$$\begin{aligned}&\mathrm{est}_i + \mathrm{p}(i, \Omega \cup \{i\}) > \mathrm{lct}_\Omega \\ &\qquad\qquad \Rightarrow\; \mathrm{est}_i := \max\left\{\mathrm{est}_i,\ \min\left\{\mathrm{est}_j + \mathrm{p}_j + \mathrm{s}_{\mathrm{f}_j \mathrm{f}_i},\ j \in \Omega\right\}\right\}\end{aligned} \tag{NF$_s$}$$

The following paragraphs describe the not-first algorithm from [32] which is based on [28]. There is also another algorithm in [34] however this one is stronger.

Let us fix for a while an activity i which we will try to update. It is clear that if there is some activity $j \in \Omega$ such that $\mathrm{est}_j + \mathrm{p}_j + \mathrm{s}_{\mathrm{f}_j \mathrm{f}_i} \leq \mathrm{est}_i$ then the set Ω cannot change est_i. The only activities which can possibly change est_i are:

$$\Psi = \{j,\ j \in T \ \&\ j \neq i \ \&\ \mathrm{est}_j + \mathrm{p}_j + \mathrm{s}_{\mathrm{f}_j \mathrm{f}_i} > \mathrm{est}_i\}$$

Therefore we will consider only sets $\Omega \subseteq \Psi$. The algorithm 4.4 constructs the set Ψ by adding one activity after another in non-decreasing order of lct_j. The following two values are recomputed after each addition:

$$\text{eftMin} = \min\left\{\text{est}_j + \text{p}_j + \text{s}_{\text{f}_j \text{f}_i},\ j \in \Psi\right\}$$
$$\text{lst} = \min\{\text{lct}_\Omega - \text{p}(i, \Omega \cup \{i\}),\ \Omega \subseteq \Psi\}$$

If lst $<$ est_i then there is a set $\Omega \subseteq \Psi$ for which the rule (4.7) holds. But we do not know the set Ω. However, we can use the rule (4.8) for the set Ψ instead – the drawback is that the resulting increase of est_i to eftMin will not be as good as it could be using the right set Ω. But if we repeat the algorithm several times then we get the same result (this is a resemblance to proposition 3).

Algorithm 4.4: Not-First/Not-Last for Batch Processing

```
for $i \in T$ do begin
   eftMin := $\infty$;
   lst := $\infty$;
   $\Psi$ := $\emptyset$;
   for $j \in T$ in non-decreasing order of $\text{lct}_j$ do
      if $i \neq j$ and $\text{est}_j + \text{p}_j + \text{s}_{\text{f}_j \text{f}_i} > \text{est}_i$ then begin
         $\Psi$ := $\Psi \cup \{j\}$;
         eftMin := $\min(\text{eftMin}, \text{est}_j + \text{p}_j + \text{s}_{\text{f}_j \text{f}_i})$;
         lst := $\min(\text{lst}, \text{lct}_\Psi - \text{p}(i, \Psi))$;
         if lst < $\text{est}_i$ then begin
            $\text{est}_i$ := eftMin;
            break;
         end;
      end;
end;
```

The worst-case time complexity of the algorithm is $O(n^2)$.

4.3.5 Precedence Graph

In case of batch processing we do not design a separate algorithm for detectable precedences and for a precedence graph. The results presented in this chapter are a generalization of the chapters 2.4.6 and 2.4.7.

First we would like to generalize the concept of detectable precedences to batch processing. The activities *i* and *j* can be processed together if:

$$\text{f}_j = \text{f}_i \ \ \&\ \ \text{c}_j + \text{c}_i \leq C \ \&\ \min\left\{\text{lct}_j, \text{lct}_i\right\} - \max\left\{\text{est}_j, \text{est}_i\right\} \geq \text{p}_j \qquad (4.9)$$

And the activity i can be processed before the activity j if:

$$\mathrm{est}_i + \mathrm{p}_i + \mathrm{s}_{\mathrm{f}_i \mathrm{f}_j} \le \mathrm{lct}_j - \mathrm{p}_j \qquad (4.10)$$

If non of the conditions (4.9) and (4.10) holds then $j \ll i$.

Definition 7 *The precedence $j \ll i$ is called detectable if non of the conditions (4.9) and (4.10) holds.*

Filtering based on a precedence graph is simple. Take an activity i and build the set Prec(i) of all its predecessors:

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

Then we propagate using the rule (4.6):

$$\mathrm{est}_i := \max\{\mathrm{est}_i,\ \max\{\mathrm{est}_{\Omega'} + \mathrm{u}(\Omega') + \mathrm{s}(F_{\Omega'} \cup \{\mathrm{f}_i\}, \mathrm{f}_i),\ \Omega' \subseteq \mathrm{Prec}(i)\}\} \qquad (\mathrm{PG}_s)$$

The algorithm 4.5 is based on this rule it has the worst-case time complexity $O(n^2)$.

Algorithm 4.5: Precedence Graph for Batch Processing

```
for $i \in T$ do begin
   $\Omega := \emptyset$;
   $m := -\infty$;
   for $j \in T$ in non-increasing order of $\mathrm{est}_j$ do
      if $j \ll i$ then begin
         $\Omega := \Omega \cup \{j\}$;
         $m := \max(m,\ \mathrm{est}_j + \mathrm{u}(\Omega) + \mathrm{s}(F_\Omega \cup \{\mathrm{f}_i\},\ \mathrm{f}_i))$;
      end;
   $\mathrm{est}_i := \max(m, \mathrm{est}_i)$;
end;
```

Once again the biggest problem of the algorithm is to find all precedences. We would like to consider the following types of precedences:

1. Precedences from the original problem.
2. Precedences added during the search as search decisions.
3. Detectable precedences.
4. Precedences found by Edge Finding.
5. Transitive closure of all precedences mentioned above.

In the rest of this section we will prove the same results for batch processing that we proved a for the unary resource. In particular:

- Precedences found by Edge Finding become detectable therefore the item 4 is a subset of the item 3.
- Detectable precedences are not "forgotten" during propagation – once a precedence is detectable it stays detectable even after arbitrary propagation.
- In item 5 it is enough to compute transitive closure of non-detectable precedences, i.e., the precedences from items 1 and 2. Since the set of precedences from the item 1 is static it is enough to recompute transitive closure after addition of a new search decision.

The following proposition is a generalization of the proposition 5. It shows that all precedences propagated by Edge Finding become detectable.

Proposition 14 *When Edge Finding is unable to find further bound adjustments then all precedences which the Edge Finding found are detectable.*

Before proving the proposition we prove the following lemma:

Lemma 3 *Let $i, j \in T$ be two different activities and let* $\mathrm{est}_i + \mathrm{p}_i > \mathrm{lct}_j - \mathrm{s}_{\mathrm{f}_i \mathrm{f}_j}$. *Then $j \ll i$ is detectable precedence.*

Proof: We have to prove that neither of the conditions (4.9) and (4.10) holds. With (4.10) it is easy – assumption of the lemma clearly contradicts it. With (4.9) we distinguish two cases:

a) $f_i \neq f_j$. In that case clearly (4.9) does not hold.

b) $f_i = f_j$. In that case $\mathrm{s}_{\mathrm{f}_i \mathrm{f}_j} = 0$ and therefore:

$$\min\left\{\mathrm{lct}_j, \mathrm{lct}_i\right\} - \max\left\{\mathrm{est}_j, \mathrm{est}_i\right\} \leq \mathrm{lct}_j - \mathrm{est}_i < \mathrm{p}_i$$

And therefore the rule (4.9) does not hold. □

Now back to the proof of the proposition 14:

Proof: Let us suppose that Edge Finding proved that $\Omega \ll i$. In the rest we prove that for an arbitrary activity $j \in \Omega$ Edge Finding made est_i so big that the precedence $j \ll i$ is detectable.

Edge Finding proved $\Omega \ll i$ so the condition (EF_s) holds:

$$\begin{aligned} \mathrm{p}(\Omega \cup \{i\}) &> \mathrm{lct}_\Omega - \mathrm{est}_\Omega \\ \mathrm{u}(\Omega \cup \{i\}) + \mathrm{s}(F_\Omega \cup \{\mathrm{f}_i\}) &> \mathrm{lct}_\Omega - \mathrm{est}_\Omega \\ \mathrm{lct}_\Omega - \mathrm{u}(\Omega \cup \{i\}) - \mathrm{s}(F_\Omega \cup \{\mathrm{f}_i\}) &< \mathrm{est}_\Omega \end{aligned} \quad (4.11)$$

Because Edge Finding is unable to further change the bounds according to (EF_s):

$$\begin{aligned}&\mathrm{est}_i \geq \max\{\mathrm{est}_{\Omega'} + \mathrm{u}(\Omega') + \mathrm{s}(F_{\Omega'} \cup \{\mathrm{f}_i\},\ \mathrm{f}_i),\ \Omega' \subseteq \Omega\}\\ &\mathrm{est}_i \geq \mathrm{est}_\Omega + \mathrm{u}(\Omega) + \mathrm{s}(F_\Omega \cup \{\mathrm{f}_i\}, \mathrm{f}_i)\end{aligned}$$

In this inequality, est_Ω can be replaced by the left side of the inequality (4.11):

$$\mathrm{est}_i > \mathrm{lct}_\Omega - \mathrm{u}(\Omega \cup \{i\}) - \mathrm{s}(F_\Omega \cup \{\mathrm{f}_i\}) + \mathrm{u}(\Omega) + \mathrm{s}(F_\Omega \cup \{\mathrm{f}_i\}, \mathrm{f}_i)$$

And because:

$$\begin{aligned}\mathrm{u}(\Omega) - \mathrm{u}(\Omega \cup \{i\}) &\geq -\mathrm{p}_i\\ \mathrm{s}(F_\Omega \cup \{\mathrm{f}_i\}, \mathrm{f}_i) - \mathrm{s}(F_\Omega \cup \{\mathrm{f}_i\}) &\geq 0\\ \mathrm{lct}_\Omega &\geq \mathrm{lct}_j\end{aligned}$$

we get:

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

Therefore according to lemma 3 the precedence $j \ll i$ is detectable. □

The following proposition is a generalization of the proposition 4 for batch processing:

Proposition 15 *Let $j \ll i$ be a detectable precedence. Then it stays detectable even after arbitrary propagation.*

Proof: Let est_i and lct_j be bounds before the propagation and let est'_i and lct'_j be bounds after the propagation[5].

First we prove that the condition (4.9) does not hold after the propagation. Assume by contradiction that it is valid:

$$\mathrm{f}_j = \mathrm{f}_i\ \ \&\ \ \mathrm{c}_j + \mathrm{c}_i \leq C\ \&\ \ \min\{\mathrm{lct}'_j, \mathrm{lct}'_i\} - \max\{\mathrm{est}'_j, \mathrm{est}'_i\} \geq \mathrm{p}_j$$

Before the propagation the precedence $j \ll i$ is detectable therefore the condition (4.9) did not hold. Since c_i, c_j, f_i, f_j cannot be changed by propagation we get that:

$$\min\{\mathrm{lct}_j, \mathrm{lct}_i\} - \max\{\mathrm{est}_j, \mathrm{est}_i\} < \mathrm{p}_j$$

And because $\mathrm{est}'_i \geq \mathrm{est}_i$, $\mathrm{est}'_j \geq \mathrm{est}_j$, $\mathrm{lct}'_i \leq \mathrm{lct}_i$ and $\mathrm{lct}'_j \leq \mathrm{lct}_j$:

$$\min\{\mathrm{lct}'_j, \mathrm{lct}'_i\} - \max\{\mathrm{est}'_j, \mathrm{est}'_i\} \leq \min\{\mathrm{lct}_j, \mathrm{lct}_i\} - \max\{\mathrm{est}_j, \mathrm{est}_i\} < \mathrm{p}_j$$

[5] Values p_i, p_j, c_i, c_j, f_i, f_j cannot be changed during the propagation because they are constant

what is a contradiction with the assumption:

$$\min\{\mathrm{lct}'_j, \mathrm{lct}'_i\} - \max\{\mathrm{est}'_j, \mathrm{est}'_i\} \geq \mathrm{p}_j$$

Therefore the condition (4.9) does not hold after the propagation.

The condition (4.10) did not hold before the propagation:

$$\mathrm{est}_i + \mathrm{p}_i + \mathrm{s}_{\mathrm{f}_i\,\mathrm{f}_j} > \mathrm{lct}_j - \mathrm{p}_j$$

And because $\mathrm{est}'_i \geq \mathrm{est}_i$ and $\mathrm{lct}'_j \leq \mathrm{lct}_j$:

$$\mathrm{est}'_i + \mathrm{p}_i + \mathrm{s}_{\mathrm{f}_i\,\mathrm{f}_j} > \mathrm{lct}'_j - \mathrm{p}_j$$

I.e., (4.10) does not hold after the propagation either.

Thus non of the conditions (4.9), (4.10) holds after the propagation therefore the precedence $j \ll i$ stays detectable. □

The notion of propagated precedence can be also easily extended for batch processing:

Definition 8 *The precedence i* $\ll$ *j is called* propagated *if:*

$$\mathrm{est}_j \geq \mathrm{est}_i + \mathrm{p}_i + \mathrm{s}_{\mathrm{f}_i\,\mathrm{f}_j}$$
$$\mathrm{lct}_i \leq \mathrm{lct}_j - \mathrm{p}_j - \mathrm{s}_{\mathrm{f}_i\,\mathrm{f}_j}$$

Notice that since both Edge Finding and Precedence Graph are based on the rule (4.6) they make all their precedences propagated.

Now we can prove that also generalization of the lemma 1 is valid for batch processing:

Proposition 16 *Let a* $\ll$ *b, b* $\ll$ *c and one of these precedences be detectable and the second one 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:

   $$\mathrm{est}_c - \mathrm{s}_{\mathrm{f}_b\,\mathrm{f}_c} \geq \mathrm{est}_b + \mathrm{p}_b$$

   The precedence $a \ll b$ is detectable therefore the inequalities (4.9) and (4.10) do not hold:

   $$\mathrm{est}_b + \mathrm{p}_b + \mathrm{s}_{\mathrm{f}_b\,\mathrm{f}_a} > \mathrm{lct}_a - \mathrm{p}_a$$
   $$\mathrm{est}_c - \mathrm{s}_{\mathrm{f}_b\,\mathrm{f}_c} + \mathrm{s}_{\mathrm{f}_b\,\mathrm{f}_a} > \mathrm{lct}_a - \mathrm{p}_a$$

Together with the triangular inequality for the setup times $\mathrm{s}_{\mathrm{f}_b\,\mathrm{f}_c} + \mathrm{s}_{\mathrm{f}_c\,\mathrm{f}_a} \geq \mathrm{s}_{\mathrm{f}_b\,\mathrm{f}_a}$:

$$\mathrm{est}_c + \mathrm{s}_{\mathrm{f}_c\,\mathrm{f}_a} > \mathrm{lct}_a - \mathrm{p}_a$$

Thus according to lemma 3 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:

$$\mathrm{lct}_b - \mathrm{p}_b \geq \mathrm{lct}_a + \mathrm{s}_{\mathrm{f}_a\,\mathrm{f}_b}$$

   The second precedence $b \ll c$ is detectable therefore the inequalities (4.9) and (4.10) do not hold:

$$\mathrm{est}_c + \mathrm{p}_c + \mathrm{s}_{\mathrm{f}_c\,\mathrm{f}_b} > \mathrm{lct}_b - \mathrm{p}_b$$
$$\mathrm{est}_c + \mathrm{p}_c + \mathrm{s}_{\mathrm{f}_c\,\mathrm{f}_b} > \mathrm{lct}_a + \mathrm{s}_{\mathrm{f}_a\,\mathrm{f}_b}$$

   We use the triangular inequality again, this time $\mathrm{s}_{\mathrm{f}_c\,\mathrm{f}_a} + \mathrm{s}_{\mathrm{f}_a\,\mathrm{f}_b} \geq \mathrm{s}_{\mathrm{f}_c\,\mathrm{f}_b}$:

$$\mathrm{est}_c + \mathrm{p}_c > \mathrm{lct}_a - \mathrm{s}_{\mathrm{f}_c\,\mathrm{f}_a}$$

   Once again according to lemma 3 the precedence $i \ll j$ is detectable. □

Finally generalization of the proposition 6:

Proposition 17 *Let $i_1, i_2, \ldots, i_n \in T$ and let $i_1 \ll i_2 \ll \cdots \ll i_n$ be propagated precedences and at least one of them is detectable. Then the precedence $i_1 \ll i_n$ is detectable.*

Proof: The same as for the proposition 6. □

By proving the previous proposition we proved that batching resource with setup times is in the point of view of detectable precedences very similar to the simple unary resource. This allows using the same way to speed up computation of the transitive closure of precedences. If non-detectable precedences (i.e., precedences from the original problem plus search decisions) are kept separate then the transitive closure of all precedences on the resource can be computed in the following two steps:

- Compute the transitive closure of non-detectable precedences using any standard algorithm.
- Add detectable precedences.

Because the set of non-detectable precedences does not change often it is advantageous to recompute transitive closure only when a new non-detectable precedence is introduced.

4.4 Experimental Results

All filtering algorithms described in this chapter were combined into one algorithm as described by algorithm 4.6.

Algorithm 4.6: Computation of fixpoint for batch processing

```
repeat
  repeat
    repeat
      repeat
        Overload_Checking;
        Not_First/Not_Last;
      until no more changes found;
      Precedence_Graph;
    until no more changes found;
    Edge_Finding;
  until no more changes found;
  Not_Before/Not_After;
until no more changes found;
```

The order in which the algorithms are called is not important for the resulting filtering. The selected order is the fastest for the tested problems.

The benchmark set can be found in [31]. Each benchmark problem is randomly generated one resource problem with $n \in \langle 10, 200\rangle$, $k \in \langle 2, 7\rangle$ and resource capacity $C \in \langle 5, 13\rangle$. The search strategy is simple: take the activity which can be scheduled first and schedule it in its earliest starting time. The table 1 shows the results (column new) and comparison with the results of our previous results [34] (column old). Time was measured on processor Intel Pentium Celeron 375MHz.

There are two reasons why the results are different:

- The old version did not use detectable precedences algorithm,
- The old version uses less effective not-first/not-last algorithm.

The table shows that these two improvements reduce the running time by one half for a lot of problems. However number of backtracks decreased only for problems d, p and z the n. There are two reasons for speed-up even without decreasing number of backtracks:

1. The new version of the not-first/not-last algorithm is faster. The number of its repetitions may have increased but this is not a common case.

				new		old	
problem	n	k	solutions	backtracks	time	backtracks	time
a	30	3	88	12	1.24s	12	2.22s
b	25	5	56251	5016	16m 10s	5016	25m 30s
c	25	5	72	0	0.80s	0	1.78s
d	40	6	12	0	0.41s	1	1.02s
e	20	2	28	0	0.13s	0	0.20s
f	50	6	6	2	0.34s	2	0.70s
g	30	3	1690	77	21.96s	77	37.08s
h	75	5	12	2	1.38s	2	2.90s
i	50	5	48	44	3.44s	44	7.59s
j	50	5	10	4	0.69s	4	1.34s
k	50	7	9	0	0.72s	0	1.33s
l	50	5	4	0	0.20s	0	0.51s
m	50	3	3	3	0.16s	3	0.35s
n	30	5	39	13	0.79s	13	1.78s
o	50	5	32	8	1.62s	8	3.60s
p	30	3	270	18	3.53s	24	6.05s
q	50	7	228	0	12.44s	0	28.03s
r	50	7	324	0	22.79s	0	44.94s
s	100	7	50	0	11.31s	0	26.48s
t	200	7	8	0	7.50s	0	18.00s
v	100	7	240	0	54.26s	0	2m 6s
w	50	2	24	4	0.83s	4	1.36s
x	50	2	1368	384	39.57s	384	1m 8s
z	50	4	24	6	1.06s	7	2.14s

Table 4.1: Batch processing with sequence dependent setup times: Experimental results

2. The filtering based on a detectable precedences is very fast. It particular it can make a lot of filtering which was originally done by edge-finding. Hence the slower edge-finding algorithm is not repeated so often.

Chapter 5

Conclusions

This book explored only a few narrow topics within the wide area of global constraints in scheduling. For example I hardly mentioned such typical scheduling constraints as cumulative resources or reservoirs, I completely omitted heuristics and problematics of choosing best branching decisions, and I didn't mention any advanced search method although they play a key role in the process of finding a solution.

But despite its limited scope I believe that this book casts some new light on a few (but important) scheduling constraints from both theoretical and practical point of view. New filtering algorithms for unary resource presented at the beginning of this book move the worst-case complexity from $\mathcal{O}(n^2)$ to $\mathcal{O}(n \log n)$ and are also experimentally confirmed to be faster than the previous ones. A novel approach for optional activities received very good acceptance on major constraint programming conference CP 2004. Finally the chapter about batch processing with sequence-dependent setup times provides efficient filtering algorithms for this kind of resource.

I do not pretend that the work presented in this book is complete. On the contrary there are still of lot of areas to explore further. I have tried to mention some of them throughout this book but let me present few major topics I would like to address in the future:

- An idempotent algorithm for detectable precedences as mentioned on page 41.
- An edge finding algorithm for unary resource with optional activities.
- A new filtering algorithms for cumulative resources inspired by Θ-tree.
- Optional activities on cumulative resources.

Moreover with the success of CP scheduling in practice there is a flow of new scientific challenges coming from real life problems. So happily there is still a lot of work to do.

List of Symbols

$\mathrm{c}(\Omega, f)$	Total capacity of the activities from Ω of the family f
c_i	Capacity required by the activity i on batching resource.
$\mathrm{DPrec}(T, i)$	Set of detectable predecessors of the activity i from the set T
$\mathrm{DSucc}(R, i)$	Set of detectable successors of the activity i from the set R
p_Ω	Duration of the set Ω
p_i	Duration of the activity i
ect_Ω	The earliest completion time of the set of activities Ω
$\overline{\mathrm{ect}}(\Theta, \Lambda)$	Maximum of $\mathrm{ect}_{\Theta \cup \{i\}}$ where $i \in \Lambda$
$\overline{\mathrm{ect}}_v$	Earliest completion time of a subtree of v with at most one optional/gray activity included
est_Ω	Earliest starting time of the set Ω
est_i	Earliest starting time of the activity i
exists_i	Boolean CP variable associated with the activity i which states whether the activity will be in the final resulting or not.
f_i	Family of the activity i (for batching resource)
lct_Ω	Latest completion time of the set Ω
lct_i	Latest completion time of the activity i
$\mathrm{LCut}(T, j)$	Set of all activities from T which cannot complete sooner than activity j
$\mathrm{Leaves}(v)$	Set of leave nodes (activities) of the subtree rooted at the node v

$\mathrm{Left}(v)$	Shortcut for $\mathrm{Leaves}(\mathrm{left}(v))$
$\mathrm{left}(v)$	Left son of the node v in binary tree
lst_Ω	Latest start time of the set Ω
$\mathrm{NLSet}(T, i)$	Set of activities from the set T which can possibly update bounds of the activity i by the rule Not-Last
$\mathrm{Prec}(i)$	Set of all predecessors of the activity i
$\mathrm{responsible}_{\overline{\mathrm{ect}}}(v)$	Optional/gray activity responsible for the value of $\overline{\mathrm{ect}}_v$
$\mathrm{responsible}_{\overline{\Sigma\mathrm{P}}}(v)$	Optional/gray activity responsible for the value of $\overline{\Sigma\mathrm{P}}_v$
$\mathrm{Right}(v)$	Shortcut for $\mathrm{Leaves}(\mathrm{right}(v))$
$\mathrm{right}(v)$	Right son of the node v in binary tree
s_{fg}	Setup time needed by resource between processing of activities with families f and g
$\Sigma\mathrm{P}_v$	Shortcut for $\mathrm{p}_{\mathrm{Leaves}(v)}$
$\overline{\Sigma\mathrm{P}}_v$	Maximum processing time of activities in a subtree of v with at most one optional/gray activity included
$\mathrm{u}(\Omega, f)$	Minimal time needed for processing of all activities of family f from Ω

List of Algorithms

List of Tables

List of Figures

Index

Bibliography

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

[2] Krzysztof R. Apt. The essence of constraint propagation. *Theoretical Computer Science*, 221(1–2):179–210, 1999. URL `citeseer.ist.psu.edu/apt98essence.html`.

[3] 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.

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

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

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

[7] J. Christopher Beck, Andrew J. Davenport, Edward M. Sitarski, and Mark S. Fox. Texture-based heuristics for scheduling revisited. In *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI-97)*, pages 241–248, Providence, Rhode Island, 1997. AAAI Press / MIT Press. ISBN 0-262-51095-2. URL `citeseer.ist.psu.edu/beck97texturebased.html`.

[8] P. Brucker. *Scheduling Algorithms*. Springer-Verlag, 3rd edition, 2001.

[9] P. Brucker and O. Thiele. A branch and bound method for the general shop problem with sequence dependent setup-times. In *OR Spectrum*, pages 145–161. Springer-Verlag, 1996.

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

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

[12] Yves Caseau and Francois Laburthe. Improved CLP Scheduling with Task Intervals. In Pascal van Hentenryck, editor, *Proceedings of the 11th International Conference on Logic Programming, ICLP'94*. The MIT press, 1994.

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

[14] Yves Caseau and Francois Laburthe. Cumulative scheduling with task intervals. In *Joint International Conference and Symposium on Logic Programming*, pages 363–377, 1996.

[15] Rina Dechter. *Constraint Processing (The Morgan Kaufmann Series in Artificial Intelligence)*. Morgan Kaufmann, May 2003. ISBN 1558608907.

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

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

[18] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H.Freeman and Company, San Francisco, 1979.

[19] Matthew L. Ginsberg, James M. Crawford, and David W. Etherington. Dynamic backtracking, 1996. URL `http://citeseer.ist.psu.edu/ginsberg96dynamic.html`.

[20] Pascal Van Hentenryck and Laurent Michel. *Constraint-Based Local Search*. The MIT Press, August 2005. ISBN 0262220776.

[21] Vipin Kumar. Algorithms for constraint-satisfaction problems: A survey. *AI Magazine*, 13(1):32–44, 1992. URL `citeseer.ist.psu.edu/article/kumar92algorithms.html`.

[22] Philippe Laborie. Complete mcs-based search: Application to resource constrained project scheduling. In Leslie Pack Kaelbling and Alessandro Saffiotti, editors, *IJCAI*, pages 181–186. Professional Book Center, 2005. ISBN 0938075934.

[23] Philippe Laborie. Algorithms for propagating resource constraints in ai planning and scheduling: existing approaches and new results. *Artif. Intell.*, 143 (2):151–188, 2003. ISSN 0004-3702. doi: http://dx.doi.org/10.1016/S0004-3702(02)00362-4.

[24] Paul Martin and David B. Shmoys. A new approach to computing optimal schedules for the job-shop scheduling problem. In W. H. Cunningham, S. T. McCormick, and M. Queyranne, editors, *Proceedings of the 5th International Conference on Integer Programming and Combinatorial Optimization, IPCO'96*, pages 389–403, Vancouver, British Columbia, Canada, 1996.

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

[26] Jean-Charles Régin. A filtering algorithm for constraints of difference. In *Proceedings of AAAI-94*, pages 362–367, 1994.

[27] Jean-Charles Régin. Generalized arc consistency for global cardinality constraint. In *Proceedings of AAAI-96*, 1996.

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

[29] C. Tsang. *Foundations of Constraint Satisfaction*. Academic Press, New York, 1993.

[30] W.J. van Hoeve. The alldiff constraint: A survey. In *Proceedings of ERCIM Workshop of Constraints*, 2001.

[31] P. Vilím and R. Barták. A benchmark set for batch processing with sequence dependent setup times, 2002. URL `http://kti.mff.cuni.cz/˜vilim/batch`.

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

[33] Petr Vilím. $O(n \log n)$ filtering algorithms for unary resource constraint. In *Proceedings of CP-AI-OR 2004*. Springer-Verlag, 2004.

[34] Petr Vilím and Roman Barták. Filtering algorithms for batch processing with sequence dependent setup times. In *Proceedings of the 6th International Conference on AI Planning and Scheduling, AIPS'02*. The AAAI Press, 2002.

[35] Petr Vilím and Roman Barták. A filtering algorithm sequence composition for batch processing with sequence dependent setup times. Technical Report 2002/1, Charles University, Faculty of Mathematics and Physics, 2002.

[36] Petr Vilím, Roman Barták, and Ondřej Čepek. Unary resource constraint with optional activities. In *Principles and Pracitce of Constraint Programming - CP 2004, 10th International Convererence, CP 2004, Toronto, Canada*, volume 3258 of *Lecture Notes in Computer Science*, pages 62–76. Springer, 2004. ISBN 3-540-23241-9.

[37] Petr Vilím, Roman Barták, and Ondřej Čepek. Extension of $o(n \log n)$ filtering algorithms for the unary resource constraint to optional activities. *Contraints*, 10(4):403–425, October 2005. ISSN 1383-7133.

[38] Matthew L. Ginsberg William D. Harvey. Limited discrepancy search. In Chris S. Mellish, editor, *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95); Vol. 1*, pages 607–615, Montréal, Québec, Canada, August 20-25 1995. Morgan Kaufmann, 1995. URL `citeseer.ist.psu.edu/harvey95limited.html`.

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

[40] Armin Wolf and Hans Schlenker. Realizing the alternative resources constraint problem with single resource constraints. In *15th International Conference on Applications of Declarative Programming and Knowledge Management - INAP 2004*, Potsdam, Brandenburg, Germany, 2004.