Technische Universität Berlin

Fraunhofer Institut
Rechnerarchitektur
und Softwaretechnik

Effiziente Suchraumeinschränkung zur constraintbasierten Planung optionaler Aktivitäten auf exklusiven Ressourcen

Diplomarbeit

zur Erlangung des Grades eines
Diplom-Informatikers

vorgelegt von

Sebastian Kuhnert

Betreuer: Dr. rer. nat. Armin Wolf, Fraunhofer Institut für Rechnerarchitektur und Softwaretechnik
Erstgutachter: Prof. Dr.-Ing. Stefan Jähnichen, Institut für Softwaretechnik und Theoretische Informatik, TU Berlin
Zweitgutachter: Dr.-Ing. Stephan Herrmann, Institut für Softwaretechnik und Theoretische Informatik, TU Berlin

Berlin, im September 2007

Danksagung

Ich danke meinem Betreuer Armin Wolf, der immer ein offenes Ohr für mich hatte und mir während der gesamten Zeit des Schreibens beratend zur Seite stand.

Ebenso danke ich meiner Familie und den Freunden aus meiner Gemeinde, die mich unterstützt und ermutigt haben. Insbesondere möchte ich meinem Vater, meiner Schwester und Markus Heller für die Mühe danken, die sie beim Korrekturlesen auf sich genommen haben.

Schließlich gilt mein Dank auch dem Evangelischen Studienwerk Villigst, das mich während meines Studiums mit einem Stipendium unterstützt und darüberhinaus ein Forum für viele interessante Gespräche geboten hat.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Einleitung

> Der Beginn aller Wissenschaften
> ist das Erstaunen, dass die
> Dinge sind, wie sie sind.
>
> *Aristoteles*

Viele Zeitplanungsprobleme haben gemeinsam, dass mehrere Dinge sich nicht zeitlich überschneiden dürfen: Maschinen können unterschiedliche Werkstücke nur nacheinander bearbeiten, und in einem Raum können nicht mehrere Veranstaltungen gleichzeitig stattfinden. Bei der mathematischen Modellierung wird das Bearbeiten eines Werkstücks ebenso wie das Abhalten einer Veranstaltung als *Aktivität* betrachtet. Eine Menge solcher Aktivitäten benötigt dann dieselbe *Ressource* – eine Maschine oder einen Raum. Da stets nur eine Aktivität gleichzeitig auf eine der hier betrachteten Ressourcen zugreifen kann, werden sie auch als *exklusive Ressourcen* bezeichnet.

Die Tatsache, dass eine Menge von Aktivitäten überlappungsfrei geplant werden muss, kann als eine Einschränkung (englisch *constraint*) für die Lösungssuche aufgefasst werden. Die Constraint-Programmierung geht genau diesen Weg und bietet einen Rahmen zum Formulieren und Lösen von unterschiedlichsten Problemen. Dabei werden die in den Einschränkungen enthaltenen Informationen genutzt, um den Suchraum einzuschränken und so das Finden von Lösungen zu beschleunigen.

Häufig hängt es von externen Faktoren ab, ob eine Aktivität tatsächlich notwendig ist und in einer Lösung enthalten sein muss – so kann für Werkstück eine Auswahl zwischen verschiedenen Maschinen bestehen. Die Aktivitäten, die die Bearbeitung dieses Werkstücks auf den einzelnen Maschinen beschreiben, sind dann *optional* und es muss jeweils eine von ihnen ausgewählt werden. Damit dies mit den Mitteln der Constraint-Programmierung effizient geschehen kann, müssen die verwendeten Verfahren erweitert werden. Ein großer Schritt in diese Richtung wurde von Vilím, Barták und Čepek [VBČ04] unternommen, jedoch blieb ein zentrales Verfahren bisher unberücksichtigt: Es ist noch kein effizienter Edgefinding-Algorithmus bekannt, der Einschränkungen aus optionalen Aktivitäten ableiten kann.

Da Edgefinding einer der erfolgreichsten Ansätze zur Suchraumeinschränkung an exklusiven Ressourcen ist, ist es das Ziel dieser Diplomarbeit, einen entsprechenden Algorithmus zu entwickeln, der bei einer möglichst geringen Laufzeit möglichst viele Einschränkungen finden soll.

Dies schlägt sich auch im Aufbau dieser Arbeit nieder: In Kapitel 1 werden mit einer Einführung in die Constraint-Programmierung die Grundlagen für die Behandlung des Problems gelegt. Anschließend wird in Kapitel 2 speziell auf die constraintbasierte Zeitplanung eingegangen, wobei der Schwerpunkt auf den bereits erwähnten exklusiven Ressourcen liegt, für die auch die am weitesten verbreiteten Verfahren zur Suchraumeinschränkung vorgestellt werden. Außerdem wird diskutiert, welche Auswirkungen die Existenz von optionalen Aktivitäten auf diese Algorithmen hat.

In Kapitel 3 wird dann der neue Edgefinding-Algorithmus vorgestellt. Um ihn zu formulieren, müssen zunächst einige Verfahren zur Berechnung der frühesten Fertigstellungszeit unterschiedlicher Mengen von Aktivitäten entwickelt werden. Anschließend wird der Algorithmus auf Eigenschaften wie Korrektheit, Laufzeit und Optimalität untersucht.

Nach diesen theoretischen Ergebnissen wird der neue Algorithmus in Kapitel 4 anhand von einigen in der Literatur bekannten Standardproblemen mit anderen Edgefinding-Algorithmen verglichen, wobei der Fokus auf die Laufzeit und die Anzahl der gefundenen Einschränkungen gelegt wird.

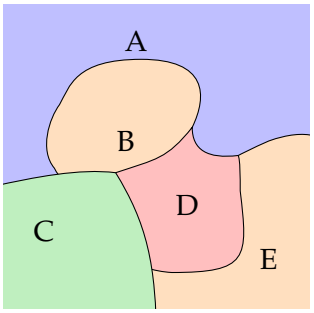
1. Constraint-Programmierung

> Was du auch tust, handle klug
> und wahre die Grenzen!
>
> *Solon, griechischer Lyriker und Politiker*

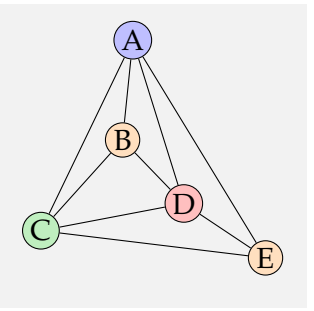
Ein zentrales Anliegen der Informatik ist das Lösen von Problemen: Alltägliche Aufgabenstellungen sollen mithilfe von Computern gelöst werden. Dazu muss (a) eine mathematische Formalisierung gefunden und (b) die formalisierte Probleminstanz durch einen Algorithmus gelöst werden.

Ein anschauliches Beispiel ist das Färben von Landkarten, wobei angrenzende Flächen unterschiedliche Farben erhalten sollen und nur eine begrenzte Anzahl von Farben zur Verfügung steht. Abbildung 1.1(a) zeigt eine Beispiel-Karte, die bereits auf zulässige Weise gefärbt ist.

Zur Modellierung dieses Problems werden häufig Graphen verwendet: Flächen werden zu Knoten, die genau dann durch eine Kante verbunden werden, wenn sie aneinandergrenzen. Der sich aus der Beispielkarte ergebende Graph wird in Abbildung 1.1(b) gezeigt. In dem so erzeugten Graphen muss jedem



(a) Die zu färbende Landkarte, hier bereits mit einer möglichen Lösung



(b) Darstellung als Graph: Flächen werden zu Knoten, Grenzen zu Kanten

$$A, B, C, D, E \in \{\mathrm{r}, \mathrm{g}, \mathrm{b}, \mathrm{o}\},$$

$$\begin{array}{ll} A \neq B, & A \neq C, \\ A \neq D, & A \neq E, \\ B \neq C, & B \neq D, \\ C \neq D, & C \neq E, \\ D \neq E & \end{array}$$

(c) Beschreibung der möglichen Lösungen mit Variablen und Ungleichheiten

Abbildung 1.1: Das Färben von Landkarten mit verschiedenen Modellierungen

Knoten eine Farbe zugeordnet werden, wobei zwei Knoten, die durch eine Kante verbunden werden, nicht dieselbe Farbe haben dürfen. Dieses als *Knotenfärbung* (englisch *vertex coloring*) bekannte Problem ist NP-vollständig. Das gilt auch im Fall von planaren Graphen, wie sie sich aus Landkarten ergeben [GJ79, Seite 191, Problem GT4].[1]

Es ist also sehr zeitaufwendig, Lösungen für große Graphen beziehungsweise Karten zu finden. Andererseits ist es sehr einfach, die möglichen Lösungen mathematisch zu beschreiben. Dies ist sogar ohne Verwendung von Graphen möglich, indem jeder Fläche eine Variable mit dem Wertebereich der möglichen Farben zugeordnet wird und die Ungleichheit der Variablen gefordert wird, die benachbarten Flächen zugeordnet sind. Abbildung 1.1(c) zeigt das Ergebnis dieser Modellierung für die Beispielkarte.

Die Constraint-Programmierung verfolgt als Vertreterin der deklarativen Programmierung genau diesen Ansatz: Ein Problem wird durch eine Menge von Variablen beschrieben, denen jeweils eine Menge möglicher Werte zugeordnet wird und die sogenannten *Constraints* (englisch für Einschränkung, Randbedingung) unterliegen. Im vorgestellten Beispiel werden hierfür nur Ungleichheiten verwendet, es ist aber eine Vielzahl verschiedenartiger Constraints möglich. Tabelle 1.1 gibt eine Übersicht über einige Constraints, die in der bei Fraunhofer FIRST entwickelten Java-Bibliothek `firstcs` [HMS+03] implementiert sind.

Tabelle 1.1: Auswahl von in der Bibliothek `firstcs` implementierten Constrainttypen

Constraint	**Beschreibung**
NotEqual	Zwei Variable dürfen nicht den gleichen Wert annehmen.
Less	Die erste Variable muss einen kleineren Wert annehmen als die zweite.
Product	Drei Variable müssen die Gleichung $a \cdot b = c$ erfüllen.
WeightedSum	Die Variablen einer Liste werden nach Multiplikation mit den ihnen zugeordneten konstanten Faktoren aufaddiert und müssen einen festen Wert ergeben.
AllDifferent	Eine Menge von Variablen muss paarweise unterschiedliche Werte annehmen.

[1] Eine Ausnahme stellen planare Graphen dar, in denen zusätzlich von keinem Knoten mehr als vier Kanten ausgehen: In diesem Fall ist eine Färbung mit der minimalen Anzahl von Farben in Polynomialzeit möglich. Dies ist im Allgemeinen aber nicht der Fall: Deutschland hat beispielsweise neun Nachbarländer.

1.1. Syntax von Constraints

Um diese Vielfalt unterschiedlicher Constraints in einem einheitlichen mathematischen Modell behandeln zu können, wird auf die Prädikatenlogik zurückgegriffen. Ziel ist, Constraints als spezielle logische Formeln aufzufassen. Dazu sind zunächst einige grundlegende Definitionen zur Syntax der Prädikatenlogik nötig:

Definition 1.1 (Logische Signatur[2])**.** Eine *logische Signatur* ist ein Tripel $\Sigma = (S, OP, R)$ aus

(a) einer nichtleeren Menge S von *Sortensymbolen*,

(b) einer Menge OP von *Operationssymbolen*; dabei ist jedem Operationssymbol $f \in OP$ eine *Operationsdeklaration* $f\colon s_1 \dots s_n \to s$ zugeordnet, wobei $n \in \mathbb{N}$ ist und $s_1, \dots, s_n, s \in S$; für $n = 0$ heißt f *Konstantensymbol*, sonst *Funktionssymbol*,

(c) einer Menge R von *Relationssymbolen*; jedem Relationssymbol $r \in R$ ist eine *Relationsdeklaration* $r : \langle s_1 \dots s_n \rangle$ zugeordnet, wobei $n \in \mathbb{N}^+$ ist und $s_1, \dots, s_n \in S$.

Eine *zu Σ passende Familie von Variablenmengen* ist eine Familie $X = (X_s)_{s \in S}$ ausVariablenmengen X_s.

Ausgehend von Signaturen können nun Terme, Formeln und Constraints definiert werden:

Definition 1.2 (Terme[3])**.** Sei $\Sigma = (S, OP, R)$ eine logische Signatur und X eine dazu passende Familie von Variablen.

Die Menge $T_{\Sigma,s}(X)$ der *Terme zur Sorte s* enthält

(a) alle Variablen $x \in X_s$ zur passenden Sorte

(b) alle Konstantensymbole $c \in OP$ mit Deklaration $c\colon \to s$

(c) zusammengesetzte Terme der Form $f(t_1, \dots, t_n)$ für alle $f \in OP$ mit Deklaration $f\colon s_1 \dots s_n \to s$, falls die einzelnen t_i jeweils Terme zur passenden Sorte sind, also $t_1 \in T_{\Sigma,s_1}(X)$, …, $t_n \in T_{\Sigma,s_n}(X)$.

Definition 1.3 (Atomare Formeln[4] und Constraints[5])**.** Sei $\Sigma = (S, OP, R)$ eine logische Signatur und X eine dazu passende Familie von Variablen.

- Die Menge $AForm_\Sigma(X)$ der *atomaren Formeln* enthält
  (a) *Gleichungen* der Form $t_1 = t_2$ zu allen Sorten $s \in S$ zwischen Termen $t_1, t_2 \in T_{\Sigma,s}(X)$ der gleichen Sorte,

[2] Vgl. [EMG+01, Definition 19.2.1].
[3] Vgl. [EMG+01, Definition 19.2.5].
[4] Vgl. [EMG+01, Definition 19.3.1].
[5] Vgl. [HW07, Definition 3.1].

(b) *Prädikationen* der Form $r(t_1, \ldots, t_n)$ für alle $r \in R$ mit Deklaration $r : \langle s_1 \ldots s_n \rangle$, falls die einzelnen t_i jeweils Terme zur passenden Sorte sind, also $t_1 \in T_{\Sigma,s_1}(X)$, $\ldots$, $t_n \in T_{\Sigma,s_n}(X)$; für einstellige Relationssymbole ist auch die Suffixschreibweise $t_1\ r$, für zweistellige die Infixschreibweise $t_1\ r\ t_2$ zulässig,

(c) das *Verum* $\top$ und das *Falsum* $\bot$ als immer wahre bzw. immer falsche Formel.[6]

- Die Menge $\mathcal{C}_\Sigma(X)$ aller *Constraints über* Σ *und* X umfasst genau alle atomaren Formeln.
- Die Menge $vars(C)$ bezeichnet alle an einem Constraint C beteiligten Variablen.

Bei dieser Definition ist zu beachten, dass noch keinerlei Aussage über die Semantik von Sorten-, Funktions- und Relationssymbolen sowie von Variablen gemacht wird: Bei deren Namen handelt es sich um einfache Zeichenketten. Um dennoch bereits auf syntaktischer Ebene den Variablen konkrete Wertebereiche zuordnen zu können, bedient man sich eines kleinen Tricks: Für jeden Wertebereich W, der einer Variable $x \in X_s$ zugewiesen werden soll, wird eine einstellige Relation »$\in W$« mit Deklaration $\in W : \langle s \rangle$ zu R hinzugefügt. Zusammen mit der Postfixnotation für Prädikationen ergibt sich dann die Formel $x \in W$.

Wie das in Abbildung 1.1(a) auf Seite 1 vorgestellte Färbeproblem mit Constraints modelliert werden kann, wird in Abbildung 1.2 gezeigt. Bemerkenswert ist dabei die Ähnlichkeit zwischen Abbildung 1.2(b) und Abbildung 1.1(c), die

sorts :	*value*
opns :	
rels :	$\neq : \langle value\ value \rangle$
	$\in\{\mathrm{r},\mathrm{g},\mathrm{b},\mathrm{o}\} : \langle value \rangle$
	$\cdots$
vars :	$A, B, C, D, E : value$

(a) Signatur mit Sortensymbolen, Operationssymbolen, Relationssymbolen sowie Variablen

$A \in\{\mathrm{r},\mathrm{g},\mathrm{b},\mathrm{o}\}$, $B \in\{\mathrm{r},\mathrm{g},\mathrm{b},\mathrm{o}\}$,
$C \in\{\mathrm{r},\mathrm{g},\mathrm{b},\mathrm{o}\}$, $D \in\{\mathrm{r},\mathrm{g},\mathrm{b},\mathrm{o}\}$,
$E \in\{\mathrm{r},\mathrm{g},\mathrm{b},\mathrm{o}\}$,
$A \neq B$, $A \neq C$, $A \neq D$,
$A \neq E$, $B \neq C$, $B \neq D$,
$C \neq D$, $C \neq E$, $D \neq E$

(b) Constraints

Abbildung 1.2: Einfache Modellierung des Färbeproblems mit Constraints

[6] Hofstedt und Wolf [HW07] behandeln Gleichungen nicht explizit und fordern stattdessen bei der Definition von Constraintsystemen, dass R für jede Sorte $s \in S$ ein Symbol $=_s$ enthält, das in der zugehörigen Struktur als Kongruenzrelation interpretiert werden muss. Vgl. dazu die Definitionen von Struktur (Seite 5) und Constraintsystem (Seite 6).

sich nur in notationellen Details unterscheiden. Dieser Effekt tritt sehr häufig auf, wenn ein Problem mit Constraints modelliert wird und führt die Eleganz deklarativer Programmierung eindrucksvoll vor Augen.

1.2. Semantik von Constraints

Constraints sind (wie logische Formeln) rein syntaktische Konstrukte, denen erst noch eine Bedeutung zugewiesen werden muss. Dazu werden in der Prädikatenlogik *Strukturen* verwendet:

Definition 1.4 (Struktur[7])**.** Sei $\Sigma = (S, OP, R)$ eine logische Signatur.

Eine Σ-*Struktur* $A = \big((A_s)_{s\in S}, (f_A)_{f\in OP}, (r_A)_{r\in R}\big)$ ist ein Tripel von Familien mit

(a) einer nichtleeren *Trägermenge* A_s für jede Sorte $s \in S$,
(b) einer Funktion $f_A\colon A_{s_1} \times \ldots \times A_{s_n} \to A_s$ für jedes Operationssymbol $f \in OP$ mit Deklaration $f\colon s_1 \ \ldots \ s_n \to s$ (für Konstantensymbole $c \in OP$, $c\colon \to s$ wird die nullstellige Funktion als Konstante $c_A \in A_s$ notiert) und
(c) einer Relation $r_A \subseteq A_{s_1} \times \ldots \times A_{s_n}$ für jedes Relationssymbol $r \in R$ mit Deklaration $r : \langle s_1 \ \ldots \ s_n\rangle$.

Die Verbindung zwischen Syntax und Semantik wird durch Variablenbelegungen, Termauswertung und die Gültigkeit von Formeln hergestellt:

Definition 1.5 (Variablenbelegung[8] und Termauswertung[9])**.** Seien eine logische Signatur $\Sigma = (S, OP, R)$, eine dazu passende Familie von Variablenmengen $X = (X_s)_{s\in S}$ und eine Σ-Struktur A gegeben.

- Eine *Variablenbelegung* $\beta\colon X \to A$, $\beta = (\beta_s)_{s\in S}$ ist eine Familie von Abbildungen $\beta_s\colon X_s \to A_s$ für jede Sorte $s \in S$. Jeder Variable wird also in der Struktur ein Wert aus der Trägermenge zur passenden Sorte zugeordnet.
- Die *Termauswertung* $\beta^*\colon T_\Sigma(X) \to A$ erweitert den Definitionsbereich einer Variablenbelegung β von einzelnen Variablen auf beliebige Terme. Sie ist definiert als eine Familie $\beta^* = (\beta^*_s)_{s\in S}$ von Abbildungen $\beta^*_s\colon T_{\Sigma,s}(X) \to A_s$, die Termen der Sorte s ein Element der zugehörigen Trägermenge in der Struktur A zuordnen:
  - $\beta^*_s(x) = \beta_s(x)$ für alle Variablen $x \in X_s$
  - $\beta^*_s(c) = c_A$ für alle Konstantensymbole $c \in OP$ mit Deklaration $c\colon \to s$
  - $\beta^*_s(f(t_1, \ldots, t_n)) = f_A(\beta^*_{s_1}(t_1), \ldots, \beta^*_{s_n}(t_n))$ für alle Funktionssymbole $f \in OP$ mit Deklaration $f : s_1 \ \ldots \ s_n \to s$.

[7] Vgl. [EMG+01, Definition 19.2.3].
[8] Vgl. [EMG+01, Seite 336].
[9] Vgl. [EMG+01, Definition 19.2.5].

Der Index zur Kennzeichnung der Sorte wird häufig weggelassen, wenn er sich aus dem Kontext ergibt, sodass $\beta(x)$ statt $\beta_s(x)$ und $\beta^*(t)$ statt $\beta^*_s(t)$ geschrieben wird.

Ausgehend davon kann nun die Gültigkeit von Formeln und Constraints definiert werden:

Definition 1.6 (Gültigkeit eines Constraints[10])**.** Sei $\Sigma = (S, OP, R)$ eine logische Signatur, $X = (X_s)_{s \in S}$ eine dazu passende Familie von Variablenmengen und A eine Σ-Struktur.

- Ein Constraint $C \in \mathcal{C}_\Sigma(X)$ wird unter einer Variablenbelegung $\beta \colon X \to A$ *bestätigt*, wenn $(A, \beta) \vDash C$ gilt, wobei die Relation $\vDash$ wie folgt definiert ist:

$$\begin{array}{ll} (A,\beta) \vDash t_1 = t_2 & \Leftrightarrow \beta^*(t_1) = \beta^*(t_2) \\ (A,\beta) \vDash r(t_1,\ldots,t_n) & \Leftrightarrow (\beta^*(t_1),\ldots,\beta^*(t_n)) \in r_A \\ (A,\beta) \vDash \top & \text{ist immer erfüllt} \\ (A,\beta) \vDash \bot & \text{ist nie erfüllt} \end{array}$$

- Ein Constraint $C \in \mathcal{C}_\Sigma(X)$ heißt *gültig in* A, geschrieben $A \vDash C$, wenn es für alle Variablenbelegungen $\beta \colon X \to A$ bestätigt wird, also $(A, \beta) \vDash C$ gilt.

Constraints werden folglich als Relationen interpretiert. Diese Relationen schränken den Lösungsraum ein, da für jede Lösung alle Constraints gültig sein müssen. Auf diese Art werden mögliche Lösungen sehr intuitiv durch Constraints beschrieben.

Die prädikatenlogischen Grundlagen lassen sich nun zu Constraintsystemen zusammenfassen:

Definition 1.7 (Constraintsystem)**.** Ein *Constraintsystem* $\mathcal{S} = (\Sigma, X, A, \mathcal{CS})$ ist ein Tupel aus

(a) einer logischen Signatur $\Sigma = (S, OP, R)$,
(b) einer dazu passenden Familie von Variablenmengen $X = (X_s)_{s \in S}$,
(c) einer Σ-Struktur A und
(d) einer Menge $\mathcal{CS}$ von zulässigen Constraints mit $\{\top, \bot\} \subseteq \mathcal{CS} \subseteq \mathcal{C}_\Sigma(X)$.[11]

Es gibt zahlreiche unterschiedliche Constraintsysteme, die jeweils für unterschiedliche Arten von Problemen verwendet werden:

[10] Vgl. die Gültigkeit von Formeln [EMG+01, Definition 19.4.1].

[11] Hofstedt und Wolf [HW07, Definition 3.1] nehmen in das Tupel zusätzlich eine Theorie (also eine Menge von gültigen Formeln) auf. Diese Menge wird für Betrachtungen im Bereich der constraint-logischen Programmierung (CLP) verwendet, ist aber für diese Arbeit nicht von Interesse.

Beispiel (Boolesche Gleichungen)**.** Das vergleichsweise einfache Constraintsystem $\mathcal{S}_{\mathbb{B}} = (\Sigma_{\mathbb{B}}, X_{\mathbb{B}}, A_{\mathbb{B}}, \mathcal{CS}_{\mathbb{B}})$ ermöglicht das Lösen boolescher Gleichungssysteme. Die Signatur $\Sigma_{\mathbb{B}} = (\{bool\}, \{true, false, not, and\}, \emptyset)$ setzt sich aus den Konstantensymbolen $true\colon \rightarrow bool$ und $false\colon \rightarrow bool$ sowie den Funktionssymbolen $not\colon bool \rightarrow$ und $and\colon bool\ bool \rightarrow bool$ zusammen. Da $\{\wedge, \neg\}$ eine Junktorbasis ist [EMG+01, Seite 276], lassen sich alle aussagenlogischen Formeln in einen logisch äquivalenten Term dieser Signatur überführen. Die Menge der Relationssymbole ist leer, da außer der vordefinierten Gleichheit keine weiteren Relationen benötigt werden. Die $\Sigma_{\mathbb{B}}$-Struktur $A_{\mathbb{B}}$ interpretiert die Signatur auf die naheliegende Weise durch $A_{\mathbb{B},bool} = \{0, 1\}$, $true_{A_{\mathbb{B}}} = 1$, $false_{A_{\mathbb{B}}} = 0$, $not_{A_{\mathbb{B}}}(x) = (x + 1 \mod 2)$ und $and_{A_{\mathbb{B}}}(x, y) = x \cdot y$. Es werden alle formulierbaren Constraints zugelassen, also $\mathcal{CS} = \mathcal{C}_{\Sigma_{\mathbb{B}}}(X_{\mathbb{B}})$.

Trotz seiner Einfachheit ermöglicht dieses Constraintsystem das Formulieren anspruchsvoller Probleme: Instanzen des NP-schweren [GJ79, Seite 259] Erfüllbarkeitsproblems für aussagenlogische Formeln lassen sich in Constraints dieses Systems überführen: Die Erfüllbarkeit der Formel $\neg(a \wedge \neg b) \wedge (\neg a \wedge b)$ entspricht dem Constraint $true = (not(a\ and\ (not\ b)))\ and\ ((not\ a)\ and\ b)$.

Ein Constraintsystem kann auch als Beschreibung einer konkreten Implementierung aufgefasst werden. Aus diesem Grund ist $\mathcal{CS}$ häufig nur eine Teilmenge von $\mathcal{C}_{\Sigma}(X)$, weil viele Implementierungen beispielsweise nicht beliebige Terme als Argument für Constraints zulassen.

Beispiel (Ein Constraintsystem zur Beschreibung von `firstcs`)**.** Die Java-Bibliothek `firstcs` ist ein am Fraunhofer Institut für Rechnerarchitektur und Softwaretechnik entwickeltes System zum Lösen von Constraints über endlichen Wertebereichen [HMS+03]. Diese werden auch als *Finite-Domain-Constraints* bezeichnet. Durch die Einschränkung auf endliche Domänen werden etwa Intervall-Constraints über rationalen und reellen Zahlen ausgeschlossen, da Intervalle über diesen Zahlenbereichen abzählbar beziehungsweise überabzählbar unendlich sind. Diese Beschränkung birgt den Vorteil, bessere Aussagen über Terminierung und asymptotische Laufzeit zu ermöglichen. Es ist üblich, $\mathbb{Z}$ oder $\mathbb{N}$ als Trägermenge zu verwenden. In `firstcs` kommt der Java-Typ `int` zum Einsatz, der Werte von -2^{31} bis $2^{31} - 1$ ermöglicht.

Die in `firstcs` möglichen Constraints und ihre Semantik werden durch das Constraintsystem $\mathcal{S}_{\texttt{firstcs}} = (\Sigma_{\texttt{firstcs}}, X_{\texttt{firstcs}}, A_{\texttt{firstcs}}, \mathcal{CS}_{\texttt{firstcs}})$ beschrieben. Aus Gründen der Übersichtlichkeit wird hier nur ein Teil der möglichen Constrainttypen angegeben, nämlich die aus Tabelle 1.1 auf Seite 2. Dabei wird wieder die vereinfachte Notation für Signaturen aus [EMG+01] verwendet, bei der die Menge der Sortensymbole hinter **sorts**, die Operationssymbole zusammen mit ihrer Deklaration hinter **opns** und die Relationsdeklarationen hinter **rels** angegeben werden.

$$
\begin{aligned}
\Sigma_{\texttt{firstcs}}: \quad \textbf{sorts}: \quad & \mathit{int}, \mathit{list} \\
\textbf{opns}: \quad & 0, 1, -1, \ldots :\rightarrow \mathit{int} \\
& \mathit{empty} :\rightarrow \mathit{list} \\
& \mathit{cons} : \mathit{int}\ \mathit{list} \rightarrow \mathit{list} \\
\textbf{rels}: \quad & \in W : \langle \mathit{int} \rangle \qquad \text{für alle } W \subseteq \mathbb{Z} \\
& \mathit{NotEqual} : \langle \mathit{int}\ \mathit{int} \rangle \\
& \mathit{Less} : \langle \mathit{int}\ \mathit{int} \rangle \\
& \mathit{Product} : \langle \mathit{int}\ \mathit{int}\ \mathit{int} \rangle \\
& \mathit{WeightedSum} : \langle \mathit{list}\ \mathit{list}\ \mathit{int} \rangle \\
& \mathit{AllDifferent} : \langle \mathit{list} \rangle \\
& \ldots
\end{aligned}
\tag{1.1}
$$

Die Sorte *list* ist dabei ein Hilfskonstrukt, um die Java-Arrays, die in `firstcs` verwendet werden, als Terme darstellen zu können.

Variable zur Sorte *int* sind als Java-Objekte realisiert, die durch ihre Adresse im Speicher identifiziert werden. Ihnen kann optional eine Zeichenkette als Name zugewiesen werden, um sie besser zuordnen zu können. Diese Namen müssen aber nicht eindeutig sein. Variable zur Sorte *list* sind nicht vorgesehen. Daraus ergibt sich $X_{\texttt{firstcs}} = (X_{\texttt{firstcs},\mathit{int}}, X_{\texttt{firstcs},\mathit{list}})$ mit $X_{\texttt{firstcs},\mathit{int}} \subseteq \mathbb{N}$ für die Speicheradressen der Objekte zur Klasse `Variable` und $X_{\texttt{firstcs},\mathit{list}} = \emptyset$.

Die Semantik der Constraints wird durch die folgende Struktur festgelegt:

$$
\begin{aligned}
A_{\texttt{firstcs}}: \quad & A_{\texttt{firstcs},\mathit{int}} = \mathbb{Z} \\
& A_{\texttt{firstcs},\mathit{list}} = \mathbb{Z}^* \\
& 0_{A_{\texttt{firstcs}}} = 0, 1_{A_{\texttt{firstcs}}} = 1, -1_{A_{\texttt{firstcs}}} = -1, \ldots \\
& \mathit{empty}_{A_{\texttt{firstcs}}} = \lambda \\
& \mathit{cons}_{A_{\texttt{firstcs}}}(a, w) = a.w \\
& \in W_{A_{\texttt{firstcs}}} = W \qquad \text{für alle } W \subseteq \mathbb{Z} \\
& \mathit{NotEqual}_{A_{\texttt{firstcs}}} = \{(a, b) \subseteq \mathbb{Z}^2 \mid a \neq b\} \\
& \mathit{Less}_{A_{\texttt{firstcs}}} = \{(a, b) \subseteq \mathbb{Z}^2 \mid a < b\} \\
& \mathit{Product}_{A_{\texttt{firstcs}}} = \{(a, b, c) \subseteq \mathbb{Z}^3 \mid a \cdot b = c\} \\
& \mathit{WeightedSum}_{A_{\texttt{firstcs}}} = \{(A, L, c) \subseteq \mathbb{Z}^* \times \mathbb{Z}^* \times \mathbb{Z} \mid \\
& \qquad |A| = |L|, \textstyle\sum_{i=1}^{|L|} A[i] \cdot L[i] = c\} \\
& \mathit{AllDifferent}_{A_{\texttt{firstcs}}} = \{L \subseteq \mathbb{Z}^* \mid L[i] \neq L[j]\ \forall i \neq j\} \\
& \ldots
\end{aligned}
\tag{1.2}
$$

In diesem Constraintsystem tritt nun auch der Fall ein, dass $\mathcal{CS}_{\texttt{firstcs}} \subsetneq \mathcal{C}_{\Sigma_{\texttt{firstcs}}}(X_{\texttt{firstcs}})$, da an vielen Stellen nur Variablen, an anderen nur Konstanten zugelassen sind. So müssen in den Constraints $\mathit{NotEqual}(x, y)$, $\mathit{Less}(x, y)$ und $\mathit{Product}(x, y, z)$ jeweils Variable für x, y und z verwendet werden. Das erste

Argument von *WeightedSum* muss eine Liste von Konstanten sein, das zweite eine Liste von Variablen und das dritte wieder eine Konstante. Außerdem wird schon auf syntaktischer Ebene verlangt, dass die Listen in den ersten beiden Argumenten gleich lang sind. Analog muss die Liste im Argument von *AllDifferent* aus Variablen bestehen. Soll ein konstanter Wert $c \in \mathbb{Z}$ verwendet werden, muss stattdessen eine Hilfsvariable x_c eingeführt werden, die zusätzlich mit dem Constraint $x_c \in \{c\}$ belegt wird.

In Abbildung 1.2(b) auf Seite 4 wurde bereits eine Darstellung des Färbeproblems als eine Menge von Constraints angegeben. Die folgende Definition formalisiert, was genau unter einem durch Constraints modellierten Problem verstanden wird:

Definition 1.8 (Constraint-Erfüllungs-Problem[12])**.** Sei $\mathcal{S} = (\Sigma, X, A, \mathcal{CS})$ ein Constraintsystem.

- Ein *Constraint-Erfüllungs-Problem* (englisch *Constraint Satisfaction Problem*, kurz *CSP*) ist eine Menge von Constraints der Form $C = \{C_1, \ldots, C_n, x_1 \in W_1, \ldots, x_m \in W_m\}$. Die jeweiligen W_i werden *Domäne* (englisch *Domain*) oder *Wertebereich* von x_i genannt und auch mit $D(x_i)$ bezeichnet. Folgende Bedingungen müssen erfüllt sein:
  (a) Die Constraints C_i haben nicht die Form $x \in W$ und
  (b) die C_i enthalten nur Variablen, zu denen auch ein Wertebereich angegeben wird, also $vars(\bigcup_{i=1}^{n} C_i) \subseteq \{x_1, \ldots, x_m\}$.

  Die Constraints $\top$ (immer wahr) und $\bot$ (immer falsch) sind ebenfalls CSPs.
- Eine *Lösung* eines CSP C ist eine Variablenbelegung $\beta\colon X \to A$, unter der alle Constraints des CSP bestätigt werden, also $(A, \beta) \vDash C_j$ für alle $C_j \in C$ gilt. Dies wird als $(A, \beta) \vDash C$ notiert.
- Ein CSP heißt *erfüllbar* oder auch *(logisch) konsistent*, falls mindestens eine Lösung existiert. Andernfalls heißt es *unerfüllbar* oder auch *inkonsistent*.

Die Menge der in Abbildung 1.2(b) auf Seite 4 aufgeführten Constraints ist ein CSP, welches das Färbeproblem für die in Abbildung 1.1(a) auf Seite 1 angegebene Karte modelliert. Die dort gewählte Färbung entspricht der Lösung $\beta(A) = b$ wie blau, $\beta(B) = \beta(E) = o$ wie orange, $\beta(C) = g$ wie grün und $\beta(D) = r$ wie rot.

[12] Vgl. [HW07, Definitionen 4.1 und 3.4]. In der Literatur wird dieses Konzept auch in Anlehnung an die graphische Darstellung als *Constraint-Netzwerk* [Dec92] und aus Perspektive der logischen Programmierung als *Constraint-Programm* [FA97; FHK+92] bezeichnet.

1.3. Constraint-Propagation und Lösungssuche

Eine Stärke der Constraint-Programmierung besteht darin, dass allgemeine Verfahren existieren, die für alle CSPs eines Constraintsystems funktionieren und nicht für jedes Problem neu programmiert werden müssen.

Das grundlegendste dieser Verfahren ist das *Filtern* der Wertebereiche. Für jeden Constrainttyp, der in einem Constraintsystem verfügbar ist, muss ein eigener Filteralgorithmus implementiert werden. Diese Algorithmen nutzen die in einem Constraint enthaltenen Informationen, um die Wertebereiche der beteiligten Variablen möglichst stark zu verkleinern, ohne dabei mögliche Lösungen zu verlieren. Dazu wird eine spezielle Form der Konsistenz verwendet:

Definition 1.9 (Lokale Konsistenz[13])**.** Sei $\mathcal{S} = (\Sigma, X, A, \mathcal{CS})$ ein Constraintsystem und $C = \{C_1, \ldots, C_n, x_1 \in W_1, \ldots, x_m \in W_m\}$ ein CSP.

- Ein Constraint $C_i \in C$ heißt *lokal konsistent*, wenn jede an C_i beteiligte Variable auf einen beliebigen Wert ihres Wertebereichs festgelegt werden kann und dann für alle übrigen an C_i beteiligten Variablen Werte aus ihren Wertebereichen existieren, sodass C_i für diese Kombination von Werten erfüllt ist:

$$\forall x \in vars(C_i) : \forall v \in D(x) : \exists \beta \colon X \to A : \\ \beta(x) = v \wedge (A, \beta) \vDash C_i \wedge \forall y \in vars(C_i) : \beta(y) \in D(y)$$

- Das CSP C heißt *lokal konsistent*, wenn jedes Constraint $C_i \in C$ lokal konsistent ist.

Der Filteralgorithmus zu einem Constrainttyp stellt in der Regel lokale Konsistenz für Constraints dieses Typs her. Bei sehr komplexen Constraints kann es allerdings vorkommen, dass der zugehörige Filteralgorithmus nur einen Teil der möglichen Einschränkungen vornimmt, weil sonst unverhältnismäßig viel Zeit benötigt würde. Wenn alle an einem Constraint beteiligten Variablen einelementige Wertebereiche haben, muss ein Filteralgorithmus aber in jedem Fall die Zulässigkeit dieser Lösung überprüfen können.

Eine mögliche Einschränkung der lokalen Konsistenz ist die *Grenzenkonsistenz*:

Definition 1.10 (Grenzenkonsistenz[14])**.** Sei $\mathcal{S} = (\Sigma, X, A, \mathcal{CS})$ ein Constraintsystem, in dem für jede Trägermenge A_s eine Ordnung existiert, und sei weiterhin $C = \{C_1, \ldots, C_n, x_1 \in W_1, \ldots, x_m \in W_m\}$ ein CSP.

[13] Vgl. [HW07, Definition 4.4]. Andere Autoren sprechen bei binären Constraints von *Kantenkonsistenz* (englisch *arc consistency*) und bei Constraints mit beliebig vielen Variablen von *Hyperkantenkonsistenz* (englisch *hyper arc consistency*).

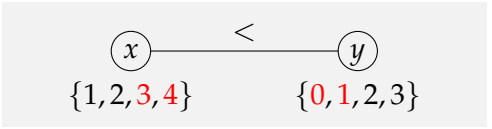
[14] Vgl. [HW07, Definition 4.6].

- Ein Constraint $C_i \in C$ heißt *grenzenkonsistent* (englisch *bounds consistent*), wenn jede an C_i beteiligte Variable auf das Minimum und Maximum ihres Wertebereichs festgelegt werden kann und für alle übrigen an C_i beteiligten Variablen Werte zwischen Minimum und Maximum ihres Wertebereichs existieren, sodass C_i für diese Kombination von Werten erfüllt ist:

$$\forall x \in vars(C_i) : \forall v \in \{\min(x), \max(x)\} : \exists \beta \colon X \to A : \\ \beta(x) = v \wedge (A, \beta) \vDash C_i \wedge \forall y \in vars(C_i) : \beta(y) \in [\min(y), \max(y)]$$

- Das CSP C heißt *grenzenkonsistent*, wenn jedes Constraint $C_i \in C$ grenzenkonsistent ist.

Beispiel (Filteralgorithmus für das *Less*-Constraint)**.** Abbildung 1.3 zeigt, welche Auswirkungen das Herstellen der lokalen Konsistenz für das *Less*-Constraint hat: Für die x-Werte 3 und 4 existieren keine y-Werte, die echt größer sind und damit das Constraint erfüllen würden. Deshalb müssen sie aus dem Wertebereich von x entfernt werden. Für die x-Werte 1 und 2 existiert aber jeweils mindestens ein größerer y-Wert, sodass sie in der Domäne erhalten bleiben müssen. Umgekehrt existiert nur für die y-Werte 2 und 3 (mindestens) ein kleinerer x-Wert, sodass die beiden anderen y-Werte aus der Domäne entfernt werden müssen.



(a) Vor dem Filtern enthalten die Domänen hier noch Werte, welche die lokale Konsistenz verletzen.

(b) Nach dem Filtern sind die betreffenden Werte entfernt und das Constraint ist lokal konsistent.

Abbildung 1.3: Lokale Konsistenz am *Less*-Constraint

Das Verfahren lässt sich zu folgendem Algorithmus verallgemeinern:

Listing 1.1: Filteralgorithmus für $Less(x, y)$

```
1 // entferne aus D(x) alle Werte ohne größeren Wert in D(y):
2 D(x) := {v ∈ D(x) | v < max{D(y)}}
3 // entferne aus D(y) alle Werte ohne kleineren Wert in D(x):
4 D(y) := {v ∈ D(y) | v > min{D(x)}}
```

Um nicht nur ein, sondern alle an einem CSP beteiligten Constraints lokal konsistent zu machen, werden die entsprechenden Filteralgorithmen wiederholt aufgerufen, bis ein Fixpunkt erreicht ist. Für das CSP

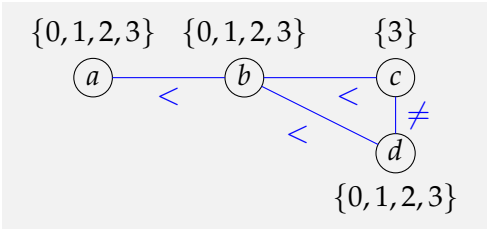
$$C_1 = \{a \in \{0,1,2,3\}, b \in \{0,1,2,3\}, c \in \{3\}, d \in \{0,1,2,3\}, \\ a < b, b < c, b < d, c \neq d\}$$

wird in Abbildung 1.4 gezeigt, wie durch (teilweise wiederholtes) Herstellen der lokalen Konsistenz die Domänen der beteiligten Variablen immer weiter eingeschränkt werden. Weil die aus den Constraints resultierenden Einschränkungen sich dabei im Graphen des CSP ausbreiten, wird dieser Vorgang *Propagation*, also Ausbreiten genannt.

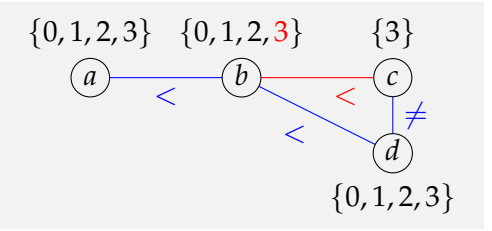
Leider ist die Constraint-Propagation nicht immer so erfolgreich wie in diesem Beispiel. Wenn beispielsweise zu Beginn c nicht durch $c \in \{3\}$ sondern nur durch $c \in \{2,3\}$ eingeschränkt würde, ergäbe sich auch für keine der übrigen Variablen eine einelementige Domäne.[15] Aus diesem Grund ist es notwendig, die Constraint-Propagation mit anderen Verfahren zu kombinieren. Hierzu werden in der Regel modifizierte Algorithmen zur Tiefensuche mit Backtracking verwendet, die in jeder Ebene des Suchbaums eine Variable betrachten und ihr für jeden Teilbaum einen Wert ihrer Domäne zuweisen. Um den exponentiell großen Lösungsraum beherrschbar zu machen, wird jede Entscheidung mit dem zuvor vorgestellten Algorithmus propagiert. Abbildung 1.5 auf Seite 14 veranschaulicht, wie dadurch der Suchraum verkleinert wird.

Das Verfahren zur Tiefensuche hat exponentielle Zeitkomplexität und ist damit schon für relativ kleine Probleme viel zu langsam. Die Effizienz der Constraint-Programmierung rührt daher, dass möglichst viele Teile des Suchbaums vermieden werden. Inwieweit das gelingt, hängt von der Struktur des CSP ab, ist jedoch nicht allgemein beweisbar. Aus diesem Grund liegt das Haupteinsatzgebiet der Constraint-Programmierung in der Lösung von Problemen, für die kein effizienter Lösungsalgorithmus bekannt ist. In der Regel handelt es sich um NP-vollständige Probleme. Für diese Problemklasse stellt die Constraint-Programmierung einen ausgezeichneten Ansatz dar, mit dem in vielen Fällen in vertretbarer Zeit Lösungen gefunden werden können. Polynomielle Laufzeitschranken können aber nicht garantiert werden – sonst wäre ja auch das P = NP-Problem gelöst.

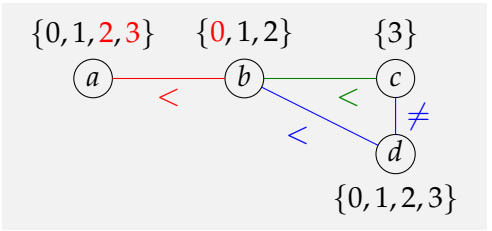
[15] Durch logische Argumentation ließen sich zwar a und b auf einelementige Wertebereiche einschränken, nur mit Mitteln der Constraint-Propagation ist dies aber nicht möglich.



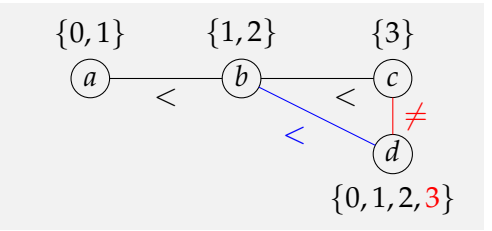
(a) Am Anfang müssen noch alle Constraints betrachtet werden. In realen Implementierungen wird in der Regel eine Warteschlange verwendet, hier wird mehrfach davon abgewichen.



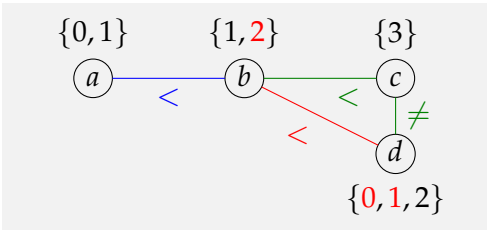
(b) Beim Filtern für $b < c$ kann 3 aus dem Wertebereich von b entfernt werden. Nach Abschluss des Filterns wird $b < c$ aus der Menge der wartenden Constraints entfernt.



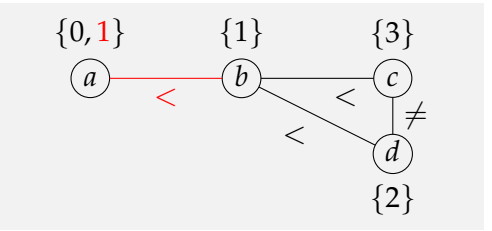
(c) Beim Filtern für $a < b$ werden Werte für a und b entfernt. Wegen der geänderten Domäne von b muss das Constraint $b < c$ erneut betrachtet werden, bringt aber keine Änderung.



(d) Beim Filtern für $c \neq d$ kann 3 aus dem Wertebereich von d entfernt werden. Wegen der Änderung bei d müsste $b < d$ erneut betrachtet werden, ist aber bereits dafür eingeplant.

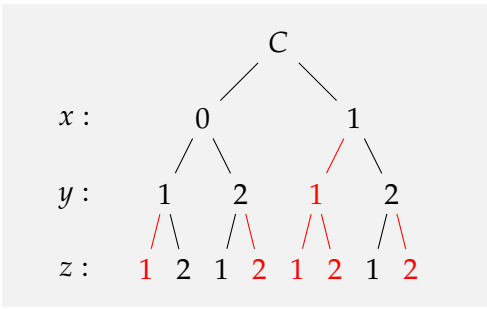


(e) Beim Filtern für $b < d$ werden mehrere Werte für b und d entfernt. Neben $a < b$ müssen auch $b < c$ und $c \neq d$ erneut betrachtet werden, letztere bringen aber keine Änderung.

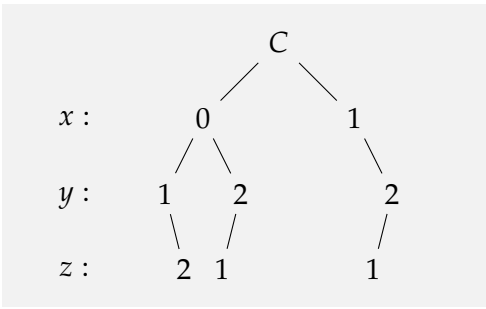


(f) Beim erneuten Filtern für $a < b$ wird schließlich ein Fixpunkt erreicht. Im Allgemeinen sind nicht wie hier alle Domänen einelementig – vgl.Abbildung 1.3(b) auf Seite 11.

Abbildung 1.4: Fixpunktberechnung zur Herstellung der lokalen Konsistenz. Noch nicht (oder noch nicht wieder) betrachtete Constraints sind blau gefärbt.



(a) Bei der einfachen Tiefensuche muss der gesamte Lösungsraum durchsucht werden. Teilbäume mit unzulässigen Lösungen sind rot gekennzeichnet.



(b) Bei der Tiefensuche mit Constraint-Propagation werden alle Teilbäume ausgelassen, in denen die lokale Konsistenz verletzt ist.

Abbildung 1.5: Lösungssuche für das CSP $C = \{x \in \{0,1\}, y \in \{1,2\}, z \in \{1,2\}, x \neq y, y \neq z\}$. In den Knoten der Bäume steht jeweils die getroffene Entscheidung.

Die Verfahren zur Lösung von CSPs lassen sich in *Constraintlösern* zusammenfassen:

Definition 1.11 (Constraintlöser[16])**.** Sei $\mathcal{S} = (\Sigma, X, A, \mathcal{CS})$ ein Constraintsystem. Ein *Constraint-Löser* (englisch *constraint solver*) für $\mathcal{S}$ ist eine Sammlung von Operationen auf Constraints dieses Systems. Diese Operationen dienen unter anderem dazu, CSPs auf Erfüllbarkeit zu untersuchen und gegebenenfalls Lösungen anzugeben.

In der Bibliothek `firstcs` ist ein Constraintlöser implementiert, der zahlreiche Constrainttypen und Suchstrategien zur Verfügung stellt. Neben der oben geschilderten einfachen Tiefensuche mit Backtracking (realisiert in der Klasse `BtLabel`) sind auch spezialisierte Varianten implementiert: So ist etwa das Suchverfahren in `ResourceLabel` auf die in Kapitel 2 beschriebenen Constraints zur Zeitplanung auf exklusiven Ressourcen spezialisiert. Es löst CSPs mit solchen Constraints erheblich schneller als das generische `BtLabel`, weil in den einzelnen Such-Schritten nicht Variablen einem festen Wert zugeordnet werden, sondern vielmehr die Reihenfolge von je zwei Aktivitäten festgelegt wird. Die genaue Vorgehensweise bei dieser Suchstrategie wird in Abschnitt 2.3 beschrieben.

[16] Für eine detailliertere Beschreibung siehe [HW07, Seite 6off.].

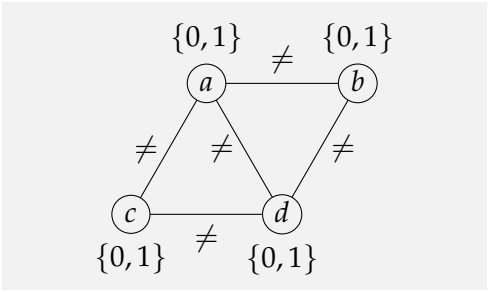
1.4. Globale Constraints

Wie schon erwähnt ist es für die Suche vorteilhaft, wenn Inkonsistenzen möglichst früh erkannt werden. Eines der dabei auftretenden Probleme ist, dass ein lokal konsistentes CSP nicht notwendigerweise erfüllbar sein muss. Abbildung 1.6(a) zeigt eine Variante des Färbeproblems (diesmal mit Zahlen statt Farben). Die angegebenen Domänen sorgen zwar für lokale Konsistenz: Für 0 kann am anderen Ende der Kanten immer 1 stehen und umgekehrt. Allerdings existiert keine Lösung, da a, c und d paarweise verschieden sein müssen.

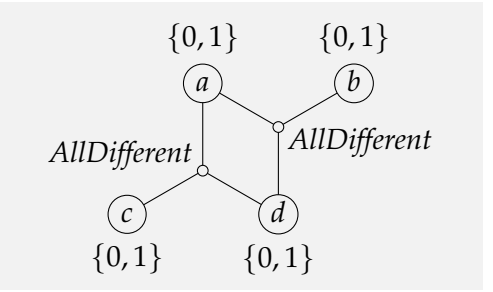
An dieser Stelle wäre es wünschenswert, eine stärkere Form von Konsistenz herstellen zu können, um möglichst alle inkonsistenten Teile des Suchbaums zu vermeiden. Die stärkste Form der Konsistenz ist globale Konsistenz:

Definition 1.12 (Globale Konsistenz[17])**.** Sei $\mathcal{S} = (\Sigma, X, A, \mathcal{CS})$ ein Constraintsystem und $C = \{C_1, \ldots, C_n, x_1 \in W_1, \ldots, x_m \in W_m\}$ ein CSP.

- Für eine Teilmenge der vorkommenden Variablen $X' \subseteq vars(C)$ enthält das *Teil-CSP* $C_{X'}$ genau die Constraints in C, an denen nur Variablen aus X' beteiligt sind, also: $C_{X'} := \{C_i \in C \mid vars(C_i) \subseteq X'\}$. (Das umfasst auch alle Wertebereich-Constraints $x_j \in W_j$ der Variablen $x_j \in X'$.)
- Ein Teil-CSP $C_{X'} \subseteq C$ heißt *global konsistent*, wenn für jede Teillösung $\beta' \colon X' \to A$ mit $(A, \beta') \vDash C_{X'}$ eine Lösung $\beta \colon X \to A$ mit $(A, \beta) \vDash C$ existiert, die für alle $x' \in X'$ mit β' übereinstimmt: $\beta(x') = \beta'(x')$.
- Das CSP C heißt *global konsistent*, wenn für alle $X' \subseteq vars(C)$ das entsprechende Teil-CSP $C_{X'}$ global konsistent ist.



(a) Ein inkonsistentes CSP (a, b und c können nicht paarweise verschieden sein), das dennoch lokal konsistent ist.



(b) Bei Verwendung des globalen *AllDifferent*-Constraints tritt lokale Inkonsistenz früher auf.

Abbildung 1.6: Die Grenzen lokaler Konsistenz

[17] Vgl. [Dec92, Definition 2.2].

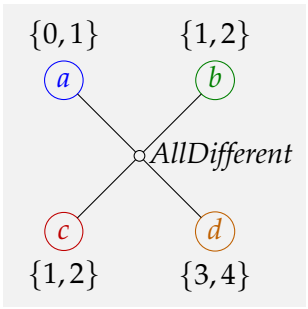
Für die Zwecke der Suche wäre es bereits ausreichend, wenn dies für einelementige X' gelten würde. Aber auch diese abgeschwächte Form der globalen Konsistenz ist im Allgemeinen nur sehr aufwendig herzustellen. Stattdessen wird ein anderer Weg eingeschlagen: Es werden weiterhin nur Algorithmen für lokale Konsistenz, dafür aber *globale Constraints* verwendet. Diese schränken möglichst viele Variablen auf einmal ein. Beispielsweise lässt sich paarweise Ungleichheit von n Variablen wahlweise durch $\frac{1}{2}n(n-1)$ *NotEqual*-Constraints oder durch ein einziges *AllDifferent*-Constraint modellieren. In Abbildung 1.6(b) auf der vorherigen Seite wird gezeigt, wie das Färbeproblem mit globalen Constraints beschrieben werden kann. Durch die stärkere Propagation wird erreicht, dass das so entstandene CSP nicht lokal konsistent ist und der tatsächlich betrachtete Teil des Suchraums verkleinert wird.

Das globale *AllDifferent*-Constraint hat nicht nur den Vorteil, Inkonsistenzen früher zu erkennen. Durch die geringere Anzahl von Constraints erreicht die Propagation mit weniger Iterationen ihren Fixpunkt. Globale Constraints ermöglichen also effizienteres Lösen von CSPs, weswegen sie bei der Modellierung eines Problems in Constraintform bevorzugt verwendet werden.

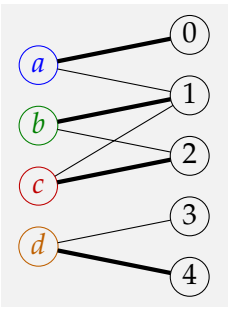
Um globale Constraints verwenden zu können, müssen dafür spezialisierte Filteralgorithmen entwickelt werden. Hierzu ist meist zusätzlicher Aufwand nötig. Hoeve [Hoe01] gibt einen Überblick über die Entwicklungen bei den Filteralgorithmen für das *AllDifferent*-Constraint. Der am weitesten verbreitete stammt von Régin [Rég94] und verwendet einen Algorithmus aus der Graphentheorie, um lokale Konsistenz herzustellen: Zu einem *AllDifferent*-Constraint C wird ein bipartiter Graph erstellt, der links für jede an C beteiligte Variable $x \in vars(C)$ einen Knoten v_x und rechts für jeden Wert $w \in \bigcup_{x \in vars(C)} D(x)$ einen Knoten v_w enthält. Zwischen zwei Knoten v_x und v_w wird genau dann eine Kante eingefügt, wenn $w \in D(x)$. Ein Beispiel hierfür ist in Abbildung 1.7 zu sehen.

Anschließend wird in diesem Graphen nach *maximalen Matchings* gesucht, also möglichst große Mengen von Kanten, die keinen Endknoten gemeinsam haben. Wenn für jeden Variablen-Knoten eine Kante enthalten ist, entspricht das Matching genau einer Variablenbelegung, die das betrachtete Constraint erfüllt. Existiert kein solches Matching, ist das Constraint inkonsistent. Andernfalls können die Wertebereiche gefiltert werden: Es werden jede Werte entfernt, die in keinem der maximalen Matchings mit der entsprechenden Variable verbunden sind, da diese Werte in keiner erfüllenden Variablenbelegung und damit auch in keiner Lösung vorkommen können. Abbildung 1.7(c) zeigt eine solche Einschränkung, die durch paarweise *NotEqual*-Constraints nicht erkannt worden wäre.

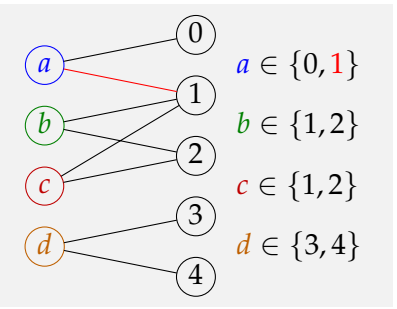
Der skizzierte Filteralgorithmus kann mit Zeitkomplexität $\mathcal{O}(m\sqrt{n})$ reali-



(a) Das zu betrachtende Constraint mit den Wertebereichen seiner Variablen



(b) Der bipartite Graph mit einem maximalen Matching



(c) Einschränkungen durch das Löschen von Kanten, die in keinem maximalen Matching vorkommen

Abbildung 1.7: Zum Filtern am *AllDifferent*-Constraint wird ein bipartiter Graph erstellt und nach maximalen Matchings durchgesucht.

siert werden, wobei $n = |vars(C)|$ die Anzahl der beteiligten Variablen und $m = \sum_{x \in vars(C)} |D(x)|$ die Summe der Wertebereichsgrößen ist [Hoe01]. Dieser Aufwand lohnt sich, da im Vergleich zu paarweisen *NotEqual*-Constraints sehr viel weniger Constraints benötigt und Einschränkungen früher gefunden werden.

1.5. Optimierung mit Constraints

Häufig soll nicht nur eine beliebige, sondern eine möglichst gute Lösung für ein Problem gefunden werden. Das ist auf der Basis von CSPs nicht direkt möglich, da für diese immer nur Erfüllbarkeit entschieden werden kann. Um mit den Mitteln der Constraint-Programmierung optimale Lösungen ermitteln zu können, muss zunächst modelliert werden, worin die Optimalität besteht. Dazu muss eine Zielfunktion angegeben werden, deren Wert minimiert oder maximiert werden soll. Das geschieht in der Regel dadurch, dass eine weitere Variable zu einem CSP hinzugefügt wird, deren Wert durch entsprechende Constraints auf den Wert der Zielfunktion eingeschränkt wird:

Definition 1.13 (Constraint-Optimierungs-Problem[18])**.** Seien ein Constraintsystem $\mathcal{S} = (\Sigma, X, A, \mathcal{CS})$, ein CSP $C \subseteq \mathcal{CS}$ und eine darin vorkommende Variable $x \in vars(C)$ gegeben. Auf dem Wertebereich von x muss eine Ordnung definiert sein. Weiterhin bezeichne $kind \in \{\min, \max\}$ die Art der Optimierung.

[18] Vgl. [HW07, Definition 14.1].

- Ein *Constraint-Optimierungs-Problem* (englisch *Constraint Optimization Problem*, kurz *COP*) ist ein Tripel $(C, x, kind)$, das aus einem Constraint-Erfüllungs-Problem, einer Variable und der Art der Optimierung besteht.
- Eine Lösung $\beta\colon X \to A$ heißt *optimal*, falls $\beta(x)$ minimal (für $kind = \min$) beziehungsweise maximal (für $kind = \max$) in Bezug auf alle Lösungen von C ist.

Wie bei der Lösung von CSPs existieren zur Lösung von COPs allgemeine Algorithmen. Dabei wird die Lösung eines COPs $(C, x, \min)$[19] auf das Lösen von CSPs $C_{bound} := C \cup \{x \leq bound\}$ zurückgeführt, in denen der Wertebereich von x zusätzlich eingeschränkt wird. Gilt nun für ein $o \in D(x)$, dass C_o erfüllbar, aber C_{o-1} inkonsistent ist, ist o der minimale Wert der Zielfunktion. Damit sind Lösungen $\beta\colon X \to A$ von C_o gleichzeitig optimale Lösungen von $(C, x, \min)$.

Ausgehend von dieser Feststellung kann ein COP mit einem Branch-and-Bound-Verfahren durch wiederholtes Lösen von CSPs gelöst werden. Dabei wird ein Intervall $[low, high]$ in jedem Schritt halbiert, indem das CSP $C_{\frac{low+high}{2}}$ betrachtet wird. Wenn es erfüllbar ist, geht es im nächsten Schritt mit dem Intervall $[low, \frac{low+high}{2}[$ weiter, wenn es unerfüllbar ist mit dem Intervall $]\frac{low+high}{2}, high]$. Wenn das Intervall leer wird, ist die zuletzt gefundene Lösung optimal. Einzige Voraussetzung für das Verfahren ist, dass zu Beginn eine untere und eine obere Schranke für das Optimum bekannt sein muss. Dies ist für die meisten Anwendungsfälle aber gegeben.

[19] Für Maximierungsprobleme $(C, x, \max)$ funktioniert das Verfahren analog.

2. Zeitplanung mit Constraints

> Die Zeit ist das, was verhindert,
> dass alles gleichzeitig passiert.
>
> *John Wheeler, Physiker*

Im Bereich der Zeitplanung (englisch *scheduling*) gibt es eine Vielzahl von zu lösenden Problemen. Beispiele sind die folgenden Fragestellungen:

- Wie kann ein Unterrichtsplan (für Universitäten oder Schulen) aussehen, der von niemandem verlangt an zwei Orten gleichzeitig zu sein, in dem keine Räume doppelt belegt sind und die Raum- und Zeitwünsche aller Beteiligten möglichst gut berücksichtigt werden [GG04]?
- Wie kann ein Fahrplan für Straßenbahnen aussehen, der die vorhandenen Gleise möglichst gut nutzt, ohne dass es zu Verspätungen kommt [Zan06]?
- Wie kann die Produktionsplanung einer Fabrik aussehen, in der Werkstücke jeweils auf unterschiedlichen Maschinen bearbeitet werden müssen, sodass alle Werkstücke möglichst früh fertig werden [BPN95]?
- Wie können mit einem Forschungssatelliten möglichst viele Messungen durchgeführt werden, wenn diese sich teilweise gegenseitig ausschließen, weil sie dieselben Instrumente verwenden oder die Kapazität der Energieversorgung überschreiten würden [Sar07]?

Diesen Problemen ist gemeinsam, dass die Qualität der möglichen Lösungen schnell überprüft werden kann, aber keine Algorithmen existieren, die schnell die beste Lösung ermitteln. Es handelt sich um NP-vollständige Probleme[1]. Damit ist es sinnvoll, sie mit Techniken der Constraint-Programmierung in Angriff zu nehmen [BPN01; Bru99; Bar02]. Dies wird auch schon seit geraumer Zeit praktiziert.

[1] Das in der Literatur als *Job-Shop-Scheduling Problem* bekannte dritte Problem wird schon von Garey und Johnson [GJ79, Seite 242] als NP-vollständig erwähnt. Auch die übrigen genannten Probleme sind Verallgemeinerungen des ebenfalls NP-vollständigen [GJ79, Seite 236] Problems der Planung auf exklusiven Ressourcen.

2.1. Aktivitäten

Um Zeitplanungsprobleme mit Mitteln der Constraint-Programmierung lösen zu können, müssen sie zunächst einmal formalisiert werden. Dazu wird von den konkreten Problembeschreibungen zu *Aktivitäten* abstrahiert. Dabei kann es sich gleichermaßen um eine Unterrichtsstunde, die Fahrt einer Straßenbahn über einen Gleisabschnitt oder das Bearbeiten eines Werkstücks auf einer Maschine handeln.

Allen Aktivitäten ist gemeinsam, dass sie eine bestimmte Zeit in Anspruch nehmen und in einem bestimmten Zeitraum erledigt werden müssen. Einmal begonnene Aktivitäten müssen zu Ende geführt werden; Unterbrechungen sind nicht zulässig.

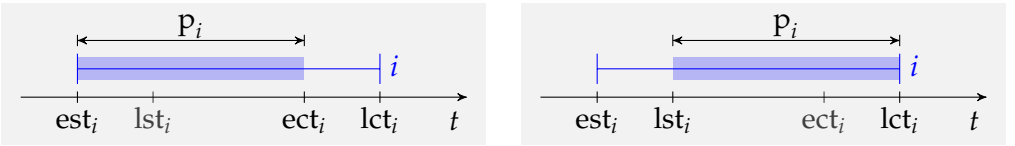
Definition 2.1 (Aktivität)**.** Eine *Aktivität i* wird durch die folgenden Eigenschaften beschrieben:

- die frühestmögliche Startzeit est_i
  (von englisch *earliest starting time*)
- die spätestmögliche Startzeit lst_i
  (von englisch *latest starting time*)
- die minimale Bearbeitungsdauer p_i
  (von englisch *production time*)
- die maximale Bearbeitungsdauer pmax_i
- die frühestmögliche Fertigstellungszeit ect_i
  (von englisch *earliest completion time*)
- die spätestmögliche Fertigstellungszeit lct_i
  (von englisch *latest completion time*)

Aktivitäten werden oft auch als *Aufgaben* (englisch *tasks*) bezeichnet.

In Constraintsystemen werden üblicherweise Startzeit, Dauer und Fertigstellungszeit jeweils als Variable modelliert, deren Domänen die entsprechenden Intervalle umfassen. Es ist im Allgemeinden nicht sinnvoll, aus diesen Intervallen einzelne Werte wegzulassen, da die meisten Algorithmen nur Grenzenkonsistenz herstellen.

In vielen Anwendungen ist die Bearbeitungsdauer konstant. In diesem Fall erübrigt sich die explizite Angabe einer Variable für die Fertigstellungszeit, da deren Intervallgrenzen als $\mathrm{ect}_i = \mathrm{est}_i + \mathrm{p}_i$ und $\mathrm{lct}_i = \mathrm{lst}_i + \mathrm{p}_i$ errechnet werden können. Umgekehrt müssen dann die Änderungen, die durch Filteralgorithmen an ect_i und lct_i vorgenommen werden, direkt an est_i und lst_i weitergereicht werden. Abbildung 2.1 zeigt eine Aktivität mit konstanter Bearbeitungsdauer. Die tatsächliche Zeitspanne kann an einer beliebigen Stelle innerhalb des spezifizierten Intervalls liegen.



(a) Bei frühestmöglicher Planung (b) Bei spätestmöglicher Planung

Abbildung 2.1: Die Eigenschaften einer Aktivität i mit konstanter Bearbeitungsdauer. Das Rechteck symbolisiert die tatsächliche Bearbeitungszeit.

Häufig werden Mengen Θ von Aktivitäten betrachtet, deren Dauer, Start- und Fertigstellungszeiten benötigt werden. Sie lassen sich wie folgt berechnen [Vil04, Seite 336]:

$$p_\Theta = \sum_{i \in \Theta} p_i \tag{2.1}$$

$$est_\Theta = \min_{i \in \Theta} \{est_i\} \tag{2.2}$$

$$lct_\Theta = \max_{i \in \Theta} \{lct_i\} \tag{2.3}$$

Die späteste Start- und früheste Fertigstellungszeit von Θ ließe sich nur mit hohem Aufwand berechnen,[2] weshalb die folgenden Abschätzungen verwendet werden:

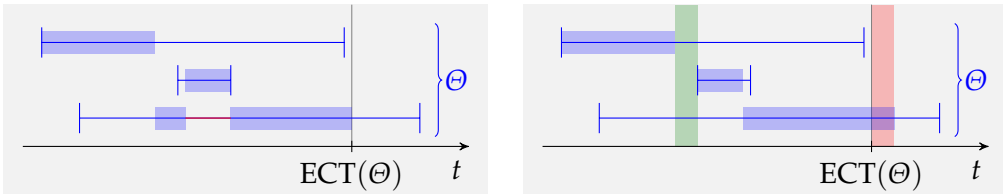
$$LST(\Theta) := \min_{\Theta' \subseteq \Theta} \{lct_{\Theta'} - p_{\Theta'}\} \tag{2.4}$$

$$ECT(\Theta) := \max_{\Theta' \subseteq \Theta} \{est_{\Theta'} + p_{\Theta'}\} \tag{2.5}$$

Diese Definitionen können für den Fall $\Theta = \emptyset$ erweitert werden, indem $p_\emptyset := 0$, $est_\emptyset := ECT(\emptyset) := -\infty$ sowie $lct_\emptyset := LST(\emptyset) := \infty$ gesetzt wird.

Die Abschätzung bei LST und ECT besteht darin, dass Unterbrechungen zugelassen werden. An dem Beispiel in Abbildung 2.2 auf der nächsten Seite ist zu erkennen, dass hierdurch $ECT(\Theta)$ vor der frühestmöglichen Fertigstellungszeit liegen kann, es sich also um eine optimistische Schätzung handelt. (Umgekehrt kann $LST(\Theta)$ nach der spätestmöglichen Startzeit liegen.) Für viele Zwecke ist diese Abschätzung aber ausreichend. In Abschnitt 3.1 wird beschrieben, wie $ECT(\Theta)$ effizient berechnet werden kann und außerdem motiviert, warum diese Abschätzung für die Verwendung in Filteralgorithmen geeignet ist.

[2] Siehe dazu auch Proposition 3.1 auf Seite 42.



(a) Bei der Berechnung von ECT(Θ) werden Unterbrechungen zugelassen.

(b) Korrekterweise würde es zu Pausen und späterer Fertigstellung kommen.

Abbildung 2.2: Abweichung zwischen ECT(Θ) und der frühesten Fertigstellungszeit von Θ.

Um die Abhängigkeiten zwischen den drei Variablen einer Aktivität in einem Constraintsystem zu modellieren, kann ein *Task*-Constraint verwendet werden, das $start_i + duration_i = completion_i$ fordert. Dies ist nur notwendig, wenn tatsächlich drei Variable verwendet werden und das Ende nicht wie oben skizziert aus Start und Dauer errechnet wird.

Das *Task*-Constraint kann durch folgenden Filteralgorithmus realisiert werden, wobei die Domänengrenzen mit den Namen aus Definition 2.1 auf Seite 20 bezeichnet werden:

Listing 2.1: Filteralgorithmus für $Task(start_i, duration_i, completion_i)$

```
1  // Es muss start_i + duration_i = completion_i sichergestellt werden:
2  if est_i < ect_i − pmax_i then
3      est_i := ect_i − pmax_i
4  if lst_i > lct_i − p_i then
5      lst_i := lct_i − p_i
6  if p_i < ect_i − lst_i then
7      p_i := ect_i − lst_i
8  if pmax_i > lct_i − est_i then
9      pmax_i := lct_i − est_i
10 if ect_i < est_i + p_i then
11     ect_i := est_i + p_i
12 if lct_i > lst_i + pmax_i then
13     lct_i := lst_i + pmax_i
```

Im Folgenden wird das Tripel $start_i, duration_i, completion_i$ häufig als i abgekürzt. Aus $Task(start_i, duration_i, completion_i)$ wird dadurch das deutlich kürzere $Task(i)$. Besonders bei Constraints über mehrere Aktivitäten verbessert sich dadurch die Lesbarkeit.

Diese notationelle Vereinbarung lässt sich auch formalisieren, indem eine neue Sorte *task* in die Signatur aufgenommen wird, die in der Struktur als Tripel interpretiert wird:

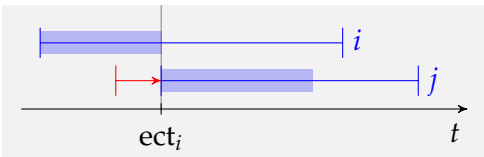
	Σ:	A:
sorts :	*val*	$A_{val} = \mathbb{Z}$
	task	$A_{task} = \mathbb{Z}^3$
opns :	$maketask : val\ val\ val \to task$	$maketask_A(s,d,c) = (s,d,c)$
	$start : task \to val$	$start_A((s,d,c)) = s$
	$duration : task \to val$	$duration_A((s,d,c)) = d$
	$completion : task \to val$	$completion_A((s,d,c)) = c$
rels :	$Task : \langle val\ val\ val \rangle$	$Task_A = \{(s,d,c) \in A_{val}^3 \mid s + d = c\}$
	$Task' : \langle task \rangle$	$Task'_A = \{(s,d,c) \in A_{task} \mid s + d = c\}$
	...	

Zwischen Aktivitäten können nun Constraints formuliert werden:

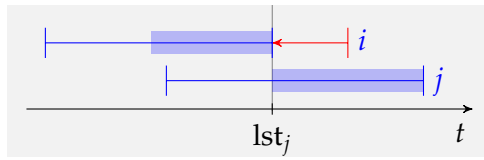
Beispiel (Das *Before*-Constraint)**.** Das Constraint $Before(i,j)$ fordert, dass die Aktivität i abgeschlossen wird, bevor mit der Aktivität j begonnen wird. Es muss also $completion_i \leq start_j$ gelten.

Abbildung 2.3 zeigt, welche Einschränkungen sich daraus ergeben: Zum einen kann est_j auf ect_i angehoben werden, zum anderen kann lct_i auf lst_j abgesenkt werden. Sind die Werte schon vorher größer beziehungsweise kleiner, ergibt sich keine weitere Einschränkung.

Auf diese Weise wird sichergestellt, dass für jede von den Domänen erlaubte Positionierung von i auch eine zulässige Position von j gefunden werden kann und umgekehrt.



(a) Die Aktivität j darf nicht begonnen werden, bevor i fertig sein kann.



(b) Die Aktivität i muss fertig sein, bevor j spätestens begonnen werden muss.

Abbildung 2.3: Mögliche Einschränkungen am Constraint $Before(i,j)$

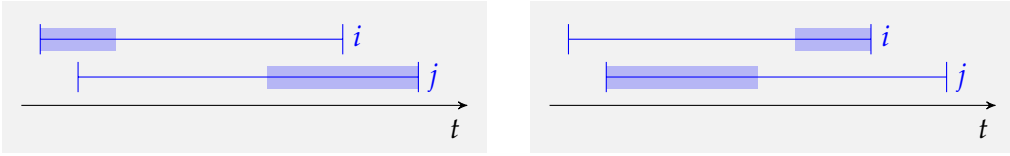
Daraus ergibt sich der folgende Filteralgorithmus:

Listing 2.2: Filteralgorithmus für $\mathit{Before}(i, j)$

```
1 // Die folgenden Anweisungen stellen completion_i ≤ start_j sicher:
2 if lct_i > lst_j then
3     lct_i := lst_j
4 if est_j < ect_i then
5     est_j = ect_i
```

Beispiel (Das *Disjunctive*-Constraint)**.** Das Constraint $\mathit{Disjunctive}(i, j)$ fordert, dass sich die Aktivitäten i und j nicht überlappen. Es muss also $\mathit{Before}(i, j)$ oder $\mathit{Before}(j, i)$ gelten.

Solange unklar ist, welche der beiden Aktivitäten zuerst eingeplant wird (vgl. dazu Abbildung 2.4), kann das *Disjunctive*-Constraint keine Einschränkungen vornehmen.



(a) Die Aktivität i kann sowohl vor ...

(b) ... als auch nach j eingeplant werden

Abbildung 2.4: Das Constraint $\mathit{Disjunctive}(i, j)$ kann keine Einschränkungen vornehmen, wenn die Reihenfolge von i und j nicht feststeht.

Erst wenn die Reihenfolge der beiden Aktivitäten klar ist, kann analog zum *Before*-Constraint gefiltert werden. Um die Reihenfolge erkennen zu können, muss eine der folgenden Bedingungen erfüllt sein:

- Falls $\mathrm{lst}_i < \mathrm{ect}_j$, muss i vor j geplant werden.
- Falls $\mathrm{lst}_j < \mathrm{ect}_i$, muss j vor i geplant werden.

Daraus ergibt sich der folgende Algorithmus:

Listing 2.3: Filteralgorithmus für $\mathit{Disjunctive}(i, j)$

```
1 // Nur filtern, wenn die Reihenfolge feststeht:
2 if lst_i < ect_j then
3     Filtere wie für Before(i, j)
4 if lst_j < ect_i then
5     Filtere wie für Before(j, i)
```

2.2. Exklusive Ressourcen

Bei den eingangs erwähnten Zeitplanungsproblemen müssen mehrere Aktivitäten auf die vorhandenen Ressourcen verteilt werden: Beim Erstellen von Unterrichtsplänen benötigt jede Veranstaltung einen Raum; werden Fahrpläne erstellt, müssen für jede Straßenbahnfahrt die entsprechenden Gleisabschnitte reserviert werden; ebenso müssen bei der Produktionsplanung jedem Werkstück Zeitabschnitte auf Maschinen zugewiesen werden, in denen es bearbeitet werden soll. Tabelle 2.1 fasst zusammen, was bei den einzelnen Problemen als Aktivität und was als Ressource modelliert werden kann.

Tabelle 2.1: Modellierung verschiedener Zeitplanungsprobleme durch Aktivitäten und Ressourcen

Problem	**Aktivitäten**	**Ressourcen**
Unterrichtsplanung	Veranstaltung/Schulstunde	Räume Lehrende
Fahrplanerstellung	Zugfahrten	Gleisabschnitte
Produktionsplanung	Bearbeitung eines Werkstücks	Maschinen
Satellitenplanung	Experimente	Messinstrumente Energieversorgung

Dabei muss noch zwischen verschiedenen Arten von Ressourcen unterschieden werden:

- Wenn eine Ressource genau einmal vorhanden ist und auf ihr zu jedem Zeitpunkt nur eine Aktivität stattfinden kann, spricht man von einer *exklusiven Ressource*. Der Name geht darauf zurück, dass sich die Aktivitäten gegenseitig ausschließen. Beispiele hierfür sind Gleisabschnitte, über die nur ein Zug gleichzeitig fahren, Messinstrumente, die nur eine Messung gleichzeitig durchführen, und Dozenten, die nur eine Veranstaltung gleichzeitig abhalten können.

  In der Literatur sind für exklusive Ressourcen auch die Bezeichnungen *unary resources* [Vil04] und *single resources* [Wol05] weit verbreitet.

- Wenn die Auswahl zwischen mehreren gleichwertigen Ressourcen besteht, auf denen jeweils nur eine Aktivität gleichzeitig stattfinden kann, spricht man von *alternativen Ressourcen* [WS05]. Zum Beispiel stehen mehrere Räume für einen Unterrichtsplan zur Verfügung, in denen aber jeweils nur eine Veranstaltung stattfinden kann.

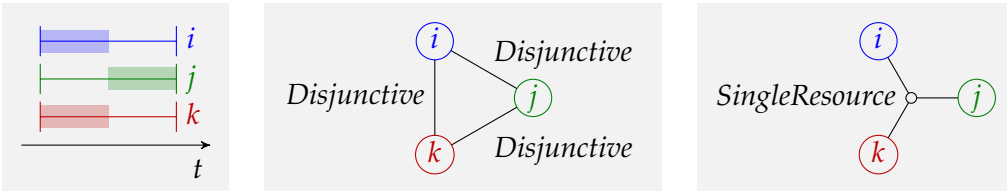
- Wenn eine Ressource mehrere Aktivitäten gleichzeitig versorgen kann, die jeweils unterschiedliche Anteile der Ressource benötigen, spricht man von *kumulativen Ressourcen* [Scho6; SWSo6; WSo6]. Ein Beispiel hierfür ist die Energieversorgung an Bord eines Satelliten, die mehrere, aber nicht beliebig viele Experimente gleichzeitig versorgen kann [Saro7].

Die vorliegende Arbeit beschäftigt sich im Weiteren ausschließlich mit exklusiven Ressourcen. Trotz der einfachen Problemstellung handelt es sich dabei um ein komplexes kombinatorisches Problem, das seit langer Zeit als NP-vollständig bekannt ist [GJ79, Seite 236, Problem SS1]. Es bietet sich daher an, mit den Mitteln der Constraint-Programmierung den Suchraum zu verkleinern, um die Lösungen schneller zu finden.

Eine einfache Modellierung von exklusiven Ressourcen kann durch paarweise *Disjunctive*-Constraints zwischen den beteiligten Aktivitäten erreicht werden. Diese stellen dann sicher, dass zu jedem Zeitpunkt höchstens eine Aktivität auf die Ressource zugreift.

Ähnlich wie beim *NotEqual*-Constraint ergeben sich aber Grenzen dessen, was durch lokale Konsistenz an *Disjunctive*-Constraints erreicht werden kann, wenn, wie in Abbildung 2.5 dargestellt, paarweise Überlappungsfreiheit von mehr als zwei Aktivitäten gefordert wird. Da bei der Constraint-Propagation nur lokale Konsistenz hergestellt wird, werden in der Suche unnötig viele Teile des Suchbaums betrachtet.

Ein weiterer Nachteil der Modellierung mit paarweisen *Disjunctive*-Constraints besteht in der großen Anzahl der benötigten Constraints, was insbesondere bei exklusiven Ressourcen mit vielen Aktivitäten ins Gewicht fällt. Dadurch wird bei der Constraint-Propagation eine große Anzahl von Iterationen benötigt, bis ein Fixpunkt erreicht wird.



(a) Wertebereiche der Aktivitäten

(b) Modellierung mit paarweisen *Disjunctive*-Constraints

(c) Modellierung mit einem *SingleResource*-Constraint

Abbildung 2.5: Die drei Aktivitäten i, j und k sollen paarweise überlappungsfrei sein, obwohl dies nicht möglich ist. Das CSP in (b) ist dennoch lokal konsistent. Das CSP in (c) überwindet dieses Problem.

Diese beiden Probleme sind die gleichen, die auch beim *NotEqual*-Constraint aufgetreten sind und durch das globale *AllDifferent*-Constraint gelöst werden konnten. Der Ansatz, ein neues globales Constraint einzuführen, führt auch im Fall des *Disjunctive*-Constraints zum Erfolg:

Definition 2.2 (*SingleResource*-Constraint)**.** Sei $T = \{i_1, i_2, \ldots, i_n\}$ eine Menge von Aktivitäten.

Das Constraint $SingleResource\colon \langle tasklist \rangle$ kann zur Modellierung von exklusiven Ressourcen verwendet werden. Für $SingleResource(i_1, i_2, \ldots, i_n)$[3] wird gefordert, dass die beteiligten Aktivitäten paarweise überlappungsfrei sind:

$$\forall i \in T : \forall j \in T : completion_i \leq start_j \vee completion_j \leq start_i$$

In Abbildung 2.6 wird eine exklusive Ressource mit fünf Aktivitäten gezeigt, die bereits überlappungsfrei angeordnet sind. Die Verwendung des *SingleResource*-Constraints ist in Abbildung 2.5(c) illustriert.

Wie beim *AllDifferent*-Constraint muss auch für das *SingleResource*-Constraint ein möglichst effizienter Filteralgorithmus entwickelt werden. Diesmal kommt aber eine zusätzliche Schwierigkeit hinzu: Für *AllDifferent* hat Régin [Rég94] einen effizienten Filteralgorithmus entwickelt, der lokale Konsistenz sicherstellen kann. Dies ist für *SingleResource* nicht möglich, da wie bereits erwähnt allein schon die Prüfung der lokalen Konsistenz ein NP-vollständiges Problem ist. Um dennoch effizient filtern zu können, muss auf eine schwächere Form der Konsistenz zurückgegriffen werden. In der Praxis werden eine Reihe von Regeln verwendet, die jeweils Inkonsistenzen erkennen oder Einschränkungen der Wertebereiche ermöglichen. Durch Kombination dieser Regeln kann

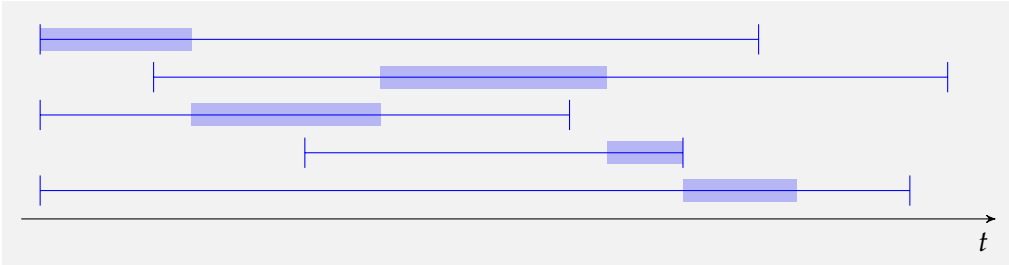


Abbildung 2.6: Exklusive Ressource mit zulässigem Zeitplan

[3] Strenggenommen kann die Aktivitätenliste nicht direkt als Parameteraufzählung angegeben werden, sondern müsste wie *list* in (1.1) auf Seite 8 durch entsprechende Operationssymbole aufgebaut werden. Um die Übersichtlichkeit zu steigern, wird hier aber diese vereinfachte Notation verwendet.

eine abgeschwächte Form der Grenzenkonsistenz hergestellt werden, die für praktische Zwecke ausreichend ist.

Im Folgenden werden die wichtigsten Regeln vorgestellt.

2.2.1. Überlasterkennung

Durch diese Regel ist es nicht möglich, Einschränkungen an den Wertebereichen vorzunehmen. Es können aber viele Fälle erkannt werden, in denen ein *SingleResource*-Constraint inkonsistent ist.

Regel 2.3 (Überlasterkennung[4])**.** Sei T die Menge der an einem *SingleResource*-Constraint C beteiligten Aktivitäten. Für alle $\Theta \subseteq T$ gilt:

Wenn $\text{est}_\Theta + \text{p}_\Theta > \text{lct}_\Theta$ ist, ist die durch C modellierte Ressource überlastet, d. h. C ist inkonsistent.

In Abbildung 2.7 wird eine Anwendung dieser Regel gezeigt.

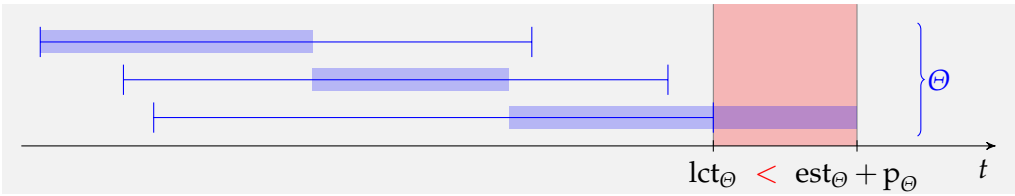


Abbildung 2.7: Eine Anwendung der Regel zur Überlasterkennung. Die Aktivitäten der Menge Θ können nicht rechtzeitig fertiggestellt werden.

Die Überlast-Regel kann mit einem Zeitaufwand von $\mathcal{O}(n \log n)$ implementiert werden, indem folgende äquivalente Regel verwendet wird [Vilo4, Seite 338]:

Regel 2.4 (Variante der Überlasterkennung)**.** Sei T die Menge der an einem *SingleResource*-Constraint C beteiligten Aktivitäten und $j \in T$ eine von ihnen. Für $\Theta := \{i \in T \mid \text{lct}_i \leq \text{lct}_j\}$ gilt:

Wenn $\text{ECT}(\Theta) > \text{lct}_\Theta = \text{lct}_j$, dann ist die durch C modellierte exklusive Ressource überlastet.

[4] Vgl. [Wolo3, Seite 741].

2.2.2. Not-First/Not-Last-Regel

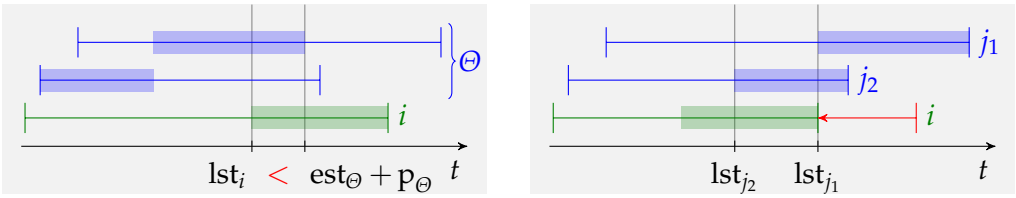
Die Not-First- und die Not-Last-Regel können erkennen, dass eine Aktivität i nicht als erste beziehungsweise letzte Aktivität einer Teilmenge $\Theta \subseteq T$ eingeplant werden kann. Die beiden Regeln sind symmetrisch, sodass hier nur die Not-Last-Regel betrachtet wird.

Regel 2.5 (Not-Last[5])**.** Sei C ein *SingleResource*-Constraint und T die Menge der daran beteiligten Aktivitäten. Für alle $\Theta \subseteq T$ und $i \in T \setminus \Theta$ gilt:

Wenn $\text{est}_\Theta + \text{p}_\Theta > \text{lst}_i$, dann kann i nicht die letzte Aktivität in der Menge $\Theta \cup i$ sein und i muss vor dem spätesten Beginn der letzten Aktivität in Θ beendet sein, sodass folgende Einschränkung möglich ist:

$$\text{lct}_i := \min \left\{ \text{lct}_i, \max_{j \in \Theta} \{\text{lst}_j\} \right\}.$$

In Abbildung 2.8 ist eine Anwendung dieser Regel zu sehen. Sie kann in $\mathcal{O}(n \log n)$ implementiert werden [Vil04, Seite 409].



(a) Die Aktivität i kann nicht als letzte der Menge $\Theta \cup \{i\}$ eingeplant werden.

(b) Durch Anwendung der Regel kann lct_i auf $\max_{j \in \Theta}\{\text{lst}_j\}$ abgesenkt werden.

Abbildung 2.8: Eine Anwendung der Not-Last-Regel

2.2.3. Edgefinding

Die Edgefinding-Regel ist in gewisser Weise das Gegenstück zur Not-First/Not-Last-Regel: Da sie erkennt, wenn eine Aktivität innerhalb einer Menge als letzte (beziehungsweise erste) eingeplant werden muss, könnte man sie auch als Is-First/Is-Last-Regel bezeichnen.[6] Es existieren wieder zwei symmetrische Varianten, von denen hier nur eine vorgestellt wird.

[5] Vgl. [TLoo].

[6] Der Name der Not-First/Not-Last hat sich erst spät herauskristallisiert [TLoo]. Früher wurde diese Regel als eine Variante von Edgefinding betrachtet [BP95], aber selten implementiert. Der konzeptionellen Klarheit wegen bezeichnet Edgefinding in dieser Arbeit ausschließlich die in diesem Abschnitt vorgestellte Regel.

Regel 2.6 (Edgefinding[7])**.** Sei C ein *SingleResource*-Constraint und T die Menge der daran beteiligten Aktivitäten. Für alle $\Theta \subseteq T$ und $i \in T \setminus \Theta$ gilt:

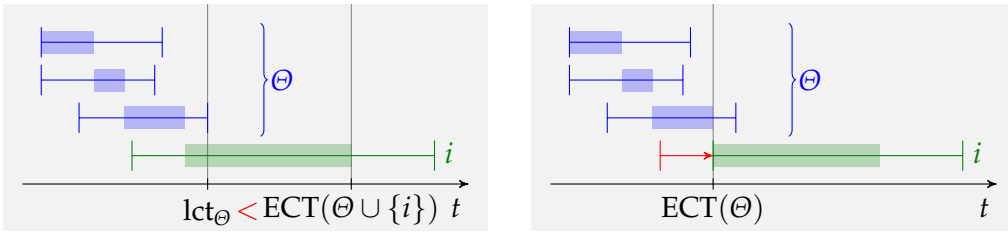
Wenn $\mathrm{lct}_\Theta < \mathrm{est}_{\Theta\cup\{i\}} + \mathrm{p}_{\Theta\cup\{i\}}$, dann muss i die letzte Aktivität in der Menge $\Theta \cup i$ sein, sodass i erst nach dem spätesten Ende der letzten Aktivität in Θ begonnen werden kann und die folgende Einschränkung möglich ist:

$$\mathrm{est}_i := \max\{\mathrm{est}_i, \mathrm{ECT}(\Theta)\}\,.$$

In Abbildung 2.9 ist eine Anwendung dieser Regel illustriert. Sie kann in $\mathcal{O}(n \log n)$ implementiert werden [VBČ04].[8] Dabei werden – ähnlich wie bei der Variante der Überlasterkennung in Regel 2.4 – nur manche $\Theta \subseteq T$ betrachtet, wodurch aber immer noch alle möglichen Einschränkungen gefunden werden:

Proposition 2.7 (Effizientes Edgefinding[9])**.** *Sei C ein SingleResource-Constraint und T die Menge der daran beteiligten Aktivitäten.*

- *Es genügt, die Edgefinding-Regel für alle $i, j \in T$ mit $\mathrm{lct}_i > \mathrm{lct}_j$ auf i und $\Theta := \{j' \in T \mid \mathrm{lct}_{j'} \leq \mathrm{lct}_j\}$ anzuwenden, wenn statt der ursprünglichen Bedingung $\mathrm{lct}_\Theta < \mathrm{est}_{\Theta\cup\{i\}} + \mathrm{p}_{\Theta\cup\{i\}}$ die Bedingung $\mathrm{lct}_\Theta < \mathrm{ECT}(\Theta \cup \{i\})$ verwendet wird.*
- *Ist die Edgefinding-Regel in der obigen Variante sowohl auf i und j_1 als auch auf i und j_2 anwendbar und ist darüberhinaus $\mathrm{lct}_{j_1} \leq \mathrm{lct}_{j_2}$, so genügt es, die Regel für i und j_2 anzuwenden, da die Anwendung auf i und j_1 nur eine schwächere Einschränkung ergeben würde.*



(a) Die Regel ist anwendbar für die Aktivitäten aus Θ und i

(b) Nach der Anwendung der Regel ist est_i mindestens $\mathrm{ECT}(\Theta)$

Abbildung 2.9: Eine Anwendung der Edgefinding-Regel

[7] Vgl. [BP96].

[8] Bereits Carlier und Pinson [CP94] hatten einen Algorithmus mit diesem Aufwand vorgestellt, der jedoch wegen komplexerer Datenstrukturen in der Praxis langsamer arbeitet.

[9] Vgl. [VBČ04, Seite 64], dort findet sich auch ein Beweis.

2.2.4. Erkennbare Präzedenzen

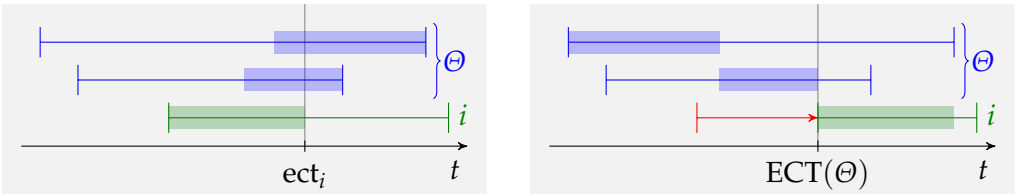
Diese Filterregel ähnelt dem Filteralgorithmus für das *Disjunctive*-Constraint, der in Listing 2.3 auf Seite 24 angegeben ist: Zuerst wird wieder geprüft, ob eine Aussage über die Reihenfolge gemacht werden kann, um in diesem Fall eine geeignete Einschränkung vorzunehmen. Während dafür beim *Disjunctive*-Constraint die Information vorliegt, dass *zwei* Aktivitäten überschneidungsfrei sein müssen, ist dies bei der Verwendung des globalen *SingleResource*-Constraints für eine ganze Menge von Aktivitäten bekannt. Diese Information kann genutzt werden, indem eine Aktivität nicht nur im Vergleich zu einer einzelnen, sondern im Vergleich zu einer Menge von Aktivitäten betrachtet wird:

Regel 2.8 (Erkennbare Präzedenzen[10])**.** Sei C ein *SingleResource*-Constraint und T die Menge der daran beteiligten Aktivitäten. Für alle Aktivitäten $i \in T$ und Mengen $\Theta \subseteq \{j \in T \mid \text{lst}_j < \text{ect}_i, j \neq i\}$ gilt:

Die Aktivität i muss nach allen Aktivitäten in Θ eingeplant werden, was die folgende Einschränkung ermöglicht:

$$\text{est}_i := \max\{\text{est}_i, \text{ECT}(\Theta)\}$$

Diese Regel wird in Abbildung 2.10 veranschaulicht. Wie bei Not-First/Not-Last existiert wieder eine symmetrische Regel. Mit einem Zeitaufwand von $\mathcal{O}(n \log n)$ können alle erkennbaren Präzedenzen gefunden und die entsprechenden Einschränkungen durchgeführt werden [Vilo4, Seite 342].



(a) Keine der Aktivitäten in Θ kann nach i eingeplant werden.

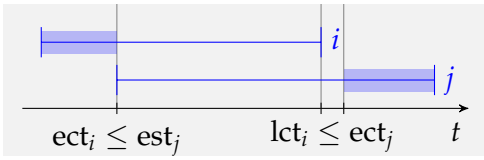
(b) Die Aktivität i kann erst begonnen werden, wenn Θ abgeschlossen ist.

Abbildung 2.10: Eine Anwendung der Regel für erkennbare Präzedenzen

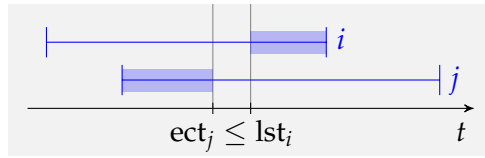
[10] Vgl. [Vilo4, Seite 341].

2.2.5. Präzedenz-Graph

Die zuletzt vorgestellte Regel unterliegt einer nicht unwesentlichen Einschränkung: Sie kann nur dann angewendet werden, wenn die vorliegende Präzedenz zwischen einer Aktivität i und einer Menge Θ *erkennbar* ist, also die Vorbedingung $\Theta \subseteq \{j \in T \mid \mathrm{lst}_j < \mathrm{ect}_i, i \neq j\}$ zutrifft. Daneben gibt es aber auch noch weitere Arten von Präzedenzen: So kann beispielsweise ein *Before*-Constraint abgesetzt werden, ohne dass die dadurch geforderte Präzedenz erkennbar wird. Ein Beispiel hierfür ist in Abbildung 2.11 zu sehen.



(a) Es gilt $\mathit{Before}(i, j)$ und der zugehörige Filteralgorithmus findet keine weitere Einschränkung.



(b) Die Präzedenz »i vor j« kann dennoch nicht aus den Wertebereichen abgeleitet werden, da $\mathrm{lst}_i \not< \mathrm{ect}_j$.

Abbildung 2.11: Die durch $\mathit{Before}(i, j)$ geforderte Präzedenz ist nicht erkennbar.

Solche *Before*-Constraints können sowohl bei der Modellierung eines Problems als auch bei der spezialisierten Suche auftreten, die in Abschnitt 2.3 beschrieben ist. Die Schwierigkeit besteht dabei nicht im Filtern dieser Präzedenzen – dafür werden *Before*-Constraints abgesetzt –, sondern im transitiven Abschluss der Präzedenz-Relation: Wenn »i vor j« und »j vor k« gilt, muss auch »i vor k« gelten. Sobald mindestens eine der ursprünglichen Präzedenzen »i vor j« und »j vor k« erkennbar und die andere propagiert ist (beispielsweise mittels *Before*-Constraint), ist auch die kombinierte Präzedenz »i vor k« erkennbar [Vilo2, Proposition 2].

Wenn zuvor ein Algorithmus für erkennbare Präzedenzen ausgeführt wurde, müssen also nur noch Präzedenzen aus dem transitiven Abschluss der nicht-erkennbaren Präzedenzen betrachtet werden. Dazu müssen beim Filtern für ein bestimmtes *SingleResource*-Constraint die beteiligten Aktivitäten paarweise betrachtet und eine gegebenenfalls vorliegende Präzedenz wie beim *Before*-Constraint (Listing 2.2 auf Seite 24) umgesetzt werden. Das verursacht einen Zeitaufwand von $\mathcal{O}(n^2)$.

Die Berechnung des transitiven Abschlusses der Präzedenz-Relation geschieht am besten global für das gesamte CSP (und nicht lokal für die einzelnen Constraints), weil in einem CSP sehr viele *SingleResource*-Constraints enthalten sein können. Da sehr oft auf den transitiven Abschluss zugegriffen werden muss,

bietet es sich an, diesen in Form einer Adjazenzmatrix im Speicher zu halten. Dadurch kann in $\mathcal{O}(1)$ geprüft werden, ob für ein Paar von zwei Aktivitäten eine Präzedenz vorliegt. Diese Adjazenzmatrix muss jedes Mal aktualisiert werden, wenn eine neue Präzedenz bekannt wird. Der beste hierfür bekannte Algorithmus benötigt $\mathcal{O}(n^2)$ Zeit, wobei n hier die Anzahl der Aktivitäten ist, über die eine (nicht-erkennbare) Präzedenz bekannt ist [Sano4; DIo6].

2.2.6. Kombination der Algorithmen

Diese Regeln werden in Algorithmen umgesetzt, die allerdings bis auf die Überlasterkennung nicht idempotent sind. Da in der Constraint-Propagation die Idempotenz der einzelnen Filteralgorithmen vorausgesetzt wird, ist eine Fixpunkt-Iteration notwendig – die einzelnen Algorithmen werden so oft ausgeführt, bis sie keine weiteren Einschränkungen mehr erkennen. Die Reihenfolge der einzelnen Algorithmen spielt für den errechneten Fixpunkt keine Rolle, wohl aber für die Laufzeit. Der folgende Ansatz ist in der Praxis am schnellsten [Vilo7, Abschnitt 2.3]:

Listing 2.4: Fixpunktiteration im Filteralgorithmus für *SingleResource*

```
repeat
   repeat
      repeat
         repeat
            if Überlasterkennung schlägt an then
               fail
            Filtere durch erkennbare Präzedenzen
         until keine weitere Einschränkungen
         Filtere durch Not-First-Not-Last
      until keine weitere Einschränkungen
      Filtere durch Edgefinding
   until keine weitere Einschränkungen
   Filtere durch Präzedenz-Graph
until keine weitere Einschränkungen
```

2.3. Spezialisierte Suchstrategien

Bei der Zeitplanung mit Mitteln der Constraint-Programmierung wird in der Regel mit einer diskretisierten Zeitachse gearbeitet, um endliche Domänen zu erhalten und damit die Terminierung sicherstellen zu können.[11] Dennoch soll die Zeit möglichst genau nachgebildet werden; Sarkarati [Saro7] verwendet etwa eine Auflösung von fünf Sekunden. Da er hierbei Zeiträume von mehreren Tagen betrachtet, können die Domänen entsprechend viele Werte enthalten. Der auf Seite 12 vorgestellte Algorithmus zur Suche mit Backtracking kommt bei so großen Domänen an seine Grenzen, da der Suchbaum unbeherrschbar groß wird und benachbarte Werte aus Domänen häufig keinen relevanten Unterschied für den Rest der Suche ergeben. Dadurch tritt der Effekt auf, dass benachbarte Teilbäume quasi identisch sind. Besonders wenn erst etliche Ebenen tiefer eine Inkonsistenz festgestellt wird, führt dies zu unnötig häufigem Durchprobieren mit entsprechend langer Laufzeit.

Dieses Problem kann dadurch gelöst werden, dass die einzelnen Suchentscheidungen (Knoten im Suchbaum) nicht einen Wert aus einer Domäne auswählen, sondern ganze Intervalle auf einmal ausschließen. Das kann dadurch erreicht werden, dass für zwei Aktivitäten auf derselben exklusiven Ressource entschieden wird, in welcher Reihenfolge sie ausgeführt werden. Dazu wird vor der Propagation ein entsprechendes *Before*-Constraint abgesetzt, das beim Backtracking wieder entfernt wird. Tabelle 2.2 vergleicht die beiden Suchstrategien miteinander.

Der sehr viel geringeren Anzahl von Möglichkeiten an den einzelnen Knoten steht dabei eine etwas größere Suchbaumtiefe gegenüber. Aufgrund der Transitivität von *Before* ist aber nach den Entscheidungen »i vor j« und »j vor k« die Entscheidung »k vor i« nicht mehr möglich. Dadurch ist an vielen der zusätzlichen Ebenen nur eine Entscheidung möglich. Insgesamt ergibt sich bei CSPs mit exklusiven Ressourcen ein erheblicher Geschwindigkeitsvorteil für die spezialisierte Suche, der das Lösen vieler Probleme überhaupt erst ermöglicht. In der Bibliothek `firstcs` ist diese spezialisierte Suche in der Klasse `ResourceLabel` realisiert.

In der Praxis ergibt sich lediglich eine Schwierigkeit: Da am Ende der spezialisierten Suche nur die Reihenfolge der einzelnen Aktivitäten, aber nicht

[11] Bei unendlichen Domänen kann es vorkommen, dass eine Fixpunktberechnung in eine Endlosschleife läuft, was etwa bei dem CSP $C = \{x \in [0,1], y \in [0,1], x < y, y < x\}$ der Fall ist. Beim Herstellen der lokalen Konsistenz werden aus den Domänen von x und y immer nur einzelne Werte herausgenommen, was wegen der unendlichen Domänen niemals zu einem Fixpunkt führt. Bei endlichen Domänen können nur endlich viele Werte aus den Domänen entfernt werden und die Terminierung ist sichergestellt.

Tabelle 2.2: Vergleich der auf exklusive Ressourcen spezialisierten Suche mit der Standardsuche

	Standardsuche	**Suche für exklusive Ressourcen**
Art der Entscheidungen	Wähle einen Wert $w \in D(x)$ aus einer Domäne	Wähle eine Reihenfolge von zwei Aktivitäten
Kinder pro Knoten	$\lvert D(x)\rvert$	maximal 2
Tiefe des Suchbaums	Anzahl der Variablen	$\frac{1}{2}(n^2 + n)$ bei einer Ressource mit n Aktivitäten
Bei Erfolg	sind alle Domänen einelementig	kann es noch mehrelementige Domänen geben

notwendigerweise ihre genauen Startzeiten und Dauern festgelegt sind, müssen zum Finden einer Lösung noch zusätzliche Schritte unternommen werden. Dies ist in jedem Fall mit normaler Tiefensuche mit Werteauswahl möglich, für spezielle Probleme kann es aber auch effizientere Verfahren geben. So können Lösungen für Job-Shop Probleme ohne weiteres Backtracking ermittelt werden, wenn die Werte in der richtigen Reihenfolge zugewiesen werden. In `firstcs` ist diese Erweiterung in der Klasse `JobShopLabel` realisiert.

2.4. Optionale Aktivitäten

Die bisher betrachteten Aktivitäten waren alle obligatorisch. Häufig ist jedoch nicht von Anfang an bekannt, ob eine Aktivität wirklich Bestandteil einer Lösung werden wird. Ein Beispiel hierfür sind die zu Beginn des Kapitels erwähnten alternativen Ressourcen, deren Aktivitäten auf genau einer der beteiligten exklusiven Ressourcen eingeplant werden müssen. Aus Sicht der exklusiven Ressourcen sind die Aktivitäten optional, da noch nicht bekannt ist, ob sie auf dieser Ressource eingeplant werden.

Ein weiteres Beispiel ist die von Sarkarati [Saro7] beschriebene Planung von Experimenten, die auf einem Forschungssatelliten durchgeführt werden sollen. Hierbei besteht ein Überangebot von möglichen Experimenten, aus dem eine Teilmenge ausgewählt werden muss. Die Aktivitäten, die die Experimente modellieren, sind also optional.

Definition 2.9 (Erweiterte Aktivität)**.**

(a) Eine *erweiterte Aktivität* ist eine Aktivität i (vgl. Definition 2.1 auf Seite 20), deren Status durch eine zusätzliche Eigenschaft beschrieben wird:

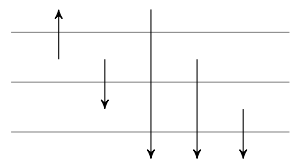
$$\text{status}_i \in \{\text{enabled}, \text{optional}, \text{disabled}\}$$

(b) Bei $\text{status}_i = \text{optional}$ heißt i *optionale Aktivität*, bei $\text{status}_i = \text{enabled}$ heißt i *obligatorische* oder *aktivierte Aktivität* und bei $\text{status}_i = \text{disabled}$ heißt i *deaktivierte Aktivität*.

(c) Eine Menge von erweiterten Aktivitäten T kann in die (disjunkten) Mengen T_{enabled}, T_{optional} und T_{disabled} zerlegt werden, die jeweils alle aktivierten, optionalen und deaktivierten Aktivitäten enthalten.

In Constraintsystemen kann diese Eigenschaft durch eine zusätzliche Variable $status_i$ realisiert werden, deren Domäne die Werte $\{1\}$ für enabled, $\{0\}$ für disabled und $\{0, 1\}$ für optional annimmt. Dadurch wird ausgedrückt, dass eine optionale Aktivität in einer Lösung (die der Statusvariable einen festen Wert zuordnen muss) entweder aktiviert oder deaktiviert sein kann. In Tabelle 2.3 wird gezeigt, wie sich der Status im Verlauf der Suchraumeinschränkung entwickeln kann.

Tabelle 2.3: Mögliche Statusänderungen einer erweiterten Aktivität bei Suchraumeinschränkungen: Es können nur Elemente aus der Domäne von $status_i$ entfernt werden.

status_i	**Domäne**	**mögliche Übergänge**
enabled	$\{1\}$	
optional	$\{0, 1\}$	
disabled	$\{0\}$	
inkonsistent	$\emptyset$	

2.4.1. Suche mit optionalen Aktivitäten

Diese zusätzlichen Variablen müssen auch bei der Suche berücksichtigt werden. Bei der einfachen Tiefensuche mit Backtracking ist dies problemlos möglich, indem sie in die Liste der zu berücksichtigten Variablen aufgenommen werden. Die auf exklusive Ressourcen spezialisierte Suche aus Abschnitt 2.3 muss hingegen für optionale Aktivitäten angepasst werden, da sie an keiner Stelle die Status-Variablen berücksichtigen könnte. Dies kann dadurch geschehen,

dass an den einzelnen Knoten nicht nur die beiden Alternativen »i vor j« und »i nach j«, sondern die folgenden fünf Fälle betrachtet werden:

- i und j aktiviert, i vor j
- i und j aktiviert, i nach j
- i aktiviert, j deaktiviert
- i deaktiviert, j aktiviert
- i und j deaktiviert.[12]

2.4.2. Filtern mit optionalen Aktivitäten

Durch die Erweiterung des Aktivitätsbegriffs ergeben sich auch Änderungen für die Filteralgorithmen für exklusive Ressourcen, da die paarweise Überlappungsfreiheit sich nur auf obligatorische Aktivitäten bezieht. Die bestehenden Algorithmen können nur weiterverwendet werden, wenn statt der Menge T aller am Constraint beteiligten Aktivitäten nur die Menge T_{enabled} der obligatorischen Aktivitäten betrachtet wird. Für die deaktivierten Aktivitäten ist dieses Ignorieren auch die einzig richtige Herangehensweise, da sie aus Sicht der Ressource ja nicht vorhanden sein sollen. Optionale Aktivitäten zu ignorieren bedeutet jedoch, Informationen ungenutzt zu lassen, und damit mögliche Einschränkungen nicht durchführen zu können. Besonders bei CSPs mit vielen optionalen Aktivitäten bedeutete dies einen erheblichen Zeitverlust, da unnötig viele Teile des Suchbaums durchsucht würden.

Die Filteralgorithmen für optionale Aktivitäten müssen der Anforderung genügen, dass wenn eine Aktivität später deaktiviert wird, keine Einschränkung vorgenommen wurde, die nicht auch bei einer deaktivierten Aktivitäten möglich gewesen wäre. Andernfalls wäre der Filteralgorithmus nicht korrekt, da er durch diese ungerechtfertigten Einschränkungen mögliche Lösungen verlieren könnte. Da deaktivierte Aktivitäten keine andere Aktivität beeinflussen dürfen, dürfen auch optionale Aktivitäten nur sich selbst beeinflussen. Dies kann auf zweierlei Weise geschehen:

[12] Diese Suchstrategie hat den Vorteil, dass dort wo die Reihenfolge der Aktivitäten feststeht, nun auch deren Status feststeht, sodass die Auswirkungen auf übrige Ressourcen und Aktivitäten maximiert werden. Ein noch früheres Festlegen des Status wäre nicht sinnvoll, da allein dafür exponentiell viele Möglichkeiten existieren, die ohne feste Reihenfolge kaum Ansätze zum Filtern bieten.

1. Wenn für eine optionale Aktivität $o \in T_{\text{optional}}$ Einschränkungen für die Menge $T_{\text{enabled}} \cup \{o\}$ ableitbar sind, die sich auf Eigenschaftsvariablen ebendieser Aktivität o auswirken, können diese durchgeführt werden. Diese Einschränkungen können sich aus den im vorherigen Abschnitt vorgestellten Regeln ergeben.

2. Wenn eine optionale Aktivität $o \in T_{\text{optional}}$ bei ihrer Aktivierung eine Inkonsistenz verursachen würde, kann sie durch Setzen von $\text{status}_o :=$ disabled deaktiviert werden. Neben der Überlast-Regel können auch die übrigen Regeln Inkonsistenzen erkennen, wenn die von ihnen für die Menge $T_{\text{enabled}} \cup \{o\}$ abgeleiteten Einschränkungen die Domäne einer Eigenschaftsvariable einer beliebigen Aktivität zur leeren Menge verkleinern würde. Dies ist beispielsweise der Fall, wenn die Edgefinding-Regel einen Wert für est_j errechnet, der größer als lst_j ist.

Ein bestehender Filteralgorithmus kann auf triviale Weise für optionale Aktivitäten erweitert werden, indem die optionalen Aktivitäten der Reihe nach durchprobiert werden:

Listing 2.5: Erweiterung zur Berücksichtigung von optionalen Aktivitäten

```
1 Führe den ursprünglichen Filteralgorithmus für T_enabled aus
2 for o ∈ T_optional do
3    Führe den ursprünglichen Filteralgorithmus für T_enabled ∪ {o} aus,
4      übernehme dabei nur die Einschränkungen für o und
5      setze status_o = disabled falls eine Inkonsistenz auftritt
```

Dieses Vorgehen hat aber einen entscheidenden Nachteil: Durch die Schleife verschlechtert sich die asymptotische Laufzeit um Faktor n, da im Extremfall alle Aktivitäten optional sein können. Ein $\mathcal{O}(n^2)$-Algorithmus benötigt in dieser Erweiterung $\mathcal{O}(n^3)$ Zeit, ein $\mathcal{O}(n \log n)$-Algorithmus $\mathcal{O}(n^2 \log n)$. Dabei kommt es insbesondere bei vielen optionalen Aktivitäten zu längeren Laufzeiten, die den Zeitgewinn durch die zusätzlich gefundenen Einschränkungen zunichte machen.

Aus diesem Grund gibt es spezielle Erweiterungen der Filteralgorithmen, die optionale Aktivitäten berücksichtigen und dennoch über eine gute Laufzeitschranke verfügen. Vilím, Barták und Čepek haben Algorithmen zur Überlasterkennung und Not-First/Not-Last-Erkennung [VBČ04] sowie erkennbare Präzedenzen [VBČ05] vorgestellt, die mit $\mathcal{O}(n \log n)$ den gleichen asymptotischen Zeitaufwand haben wie ihre Pendants ohne Behandlung optionaler Aktivitäten.

Das Filtern durch den Präzedenz-Graphen lässt sich nur schwer für optionale Aktivitäten erweitern, da dabei auf eine globale Adjazenzmatrix zugegriffen wird, die dann in einer zusätzlichen Variante pro optionaler Aktivität vorgehalten oder wiederholt neuberechnet werden müsste.

Effizientes Edgefinding für optionale Aktivitäten wird hingegen von Vilím als ein noch offenes Forschungsproblem benannt [Vil07, Seite 51]. Bisher war nur ein $\mathcal{O}(n^3)$-Algorithmus bekannt [BF99], der durch die oben dargestellte Technik (Listing 2.5) und den $\mathcal{O}(n \log n)$-Algorithmus für Edgefinding ohne optionale Aktivitäten auf $\mathcal{O}(n^2 \log n)$ verbessert werden kann. Im folgenden Kapitel wird ein neuer Algorithmus vorgestellt, der Edgefinding für optionale Aktivitäten erstmals in $\mathcal{O}(n^2)$ ermöglicht.

3. Edgefinding mit optionalen Aktivitäten

> Ich bin in meinem Tun frei, und das bedeutet, dass ich immer mehrere Möglichkeiten habe.
>
> *Peter Bieri, Philosoph und Schriftsteller*

In diesem Kapitel wird ein neuer Algorithmus für Edgefinding mit optionalen Aktivitäten vorgestellt, der mit einer asymptotischen Laufzeit von $\mathcal{O}(n^2)$ auskommt.[1] Der beste bisher bekannte Algorithmus benötigt hierfür mindestens $\mathcal{O}(n^2 \log n)$ (vgl. Abschnitt 2.4.2). Die tatsächliche Laufzeit dieser beiden (und zweier weiterer) Algorithmen wird in Kapitel 4 untersucht.

3.1. Berechnung der frühesten Fertigstellungszeit

Im neuen Algorithmus sind einige Ideen aus dem Edgefinding-Algorithmus von Vilím, Barták und Čepek [VBČ04] wiederverwendet und weiterentwickelt worden. Dabei handelt es sich vor allem um die Berechnung der frühesten Fertigstellungszeit. Die bestehenden Ansätze müssen zur Behandlung optionaler Aktivitäten erweitert werden.[2]

Sei also Θ eine Menge von Aktivitäten, deren früheste Fertigstellungszeit gemäß (2.5) wie folgt abgeschätzt werden kann:

$$\mathrm{ECT}(\Theta) \overset{(2.5)}{=} \max_{\Theta' \subseteq \Theta} \left\{ \mathrm{est}_{\Theta'} + \mathrm{p}_{\Theta'} \right\}$$

[1] Dieser Algorithmus wird auch in einem vom Autor dieser Arbeit verfassten Artikel beschrieben [Kuh07], der für die *International Conference on Applications of Declarative Programming and Knowledge Management* (INAP 2007, `http://www1.informatik.uni-wuerzburg.de/databases/INAP/2007/`) angenommen wurde.

[2] Für die symmetrische Variante der Edgefinding-Regel wird auch die späteste Startzeit $\mathrm{LST}(\Theta)$ benötigt. Da deren Berechnung aber ebenfalls symmetrisch zu der der frühesten Fertigstellungszeit ist, wird hier auf eine explizite Darstellung verzichtet.

Die Motivation für die Abschätzung besteht darin, dass kein effizientes Verfahren zur genauen Berechnung der frühesten Fertigstellungszeit bekannt ist. Darüber hinaus gilt:

Proposition 3.1 (Komplexität von ect_Θ)**.** *Die genaue früheste Fertigstellungszeit* ect_Θ *einer Aktivitätenmenge* Θ *kann nicht mit polynomiellem Aufwand berechnet werden, es sei denn* $\mathrm{P} = \mathrm{NP}$.

Beweis. Annahme: ect_Θ kann in polynomieller Zeit berechnet werden.

Wenn alle sogenannten Task-Intervalle $\Theta \subseteq T$ die Ungleichung $\mathrm{ect}_\Theta \leq \mathrm{lct}_\Theta$ erfüllen, können alle am Constraint beteiligten Aktivitäten überlappungsfrei geplant werden [CL94]. Das Task-Intervall $[i_1, i_2]$ zweier Aktivitäten $i_1, i_2 \in T$ umfasst dabei alle Aktivitäten $j \in T$ mit $\mathrm{est}_{i_1} \leq \mathrm{est}_j$ und $\mathrm{lct}_j \leq \mathrm{lct}_{i_2}$. Da es nur $\mathcal{O}(n^2)$ Task-Intervalle gibt und Polynome unter Verkettung abgeschlossen sind, kann das Problem der Planung auf einer exklusiven Ressource durch einen deterministisch polynomiellen Algorithmus entschieden werden. Weil dieses Problem (wie bereits erwähnt) NP-vollständig ist [GJ79, Seite 236], folgt aus der Annahme $\mathrm{P} = \mathrm{NP}$. □

Da durch die Edgefinding-Regel, wie in Abbildung 2.9 auf Seite 30 illustriert, die früheste Startzeit einer Aktivität i auf $\mathrm{ECT}(\Theta)$ angehoben wird, darf $\mathrm{ECT}(\Theta)$ nur nach unten von der tatsächlichen frühesten Fertigstellungszeit abweichen: Auf diese Weise verliert die Einschränkung zwar an Schärfe, bleibt aber auf jeden Fall zulässig.[3]

Proposition 3.2 (Abschätzung durch $\mathrm{ECT}(\Theta)$)**.** *Sei* Θ *eine Menge von Aktivitäten und* ect_Θ *ihre tatsächliche früheste Fertigstellungszeit, so gilt*

$$\mathrm{ECT}(\Theta) \leq \mathrm{ect}_\Theta \ .$$

Beweis. Sei $\Theta' \subseteq \Theta$ beliebig, aber fest.

Es gilt $\mathrm{ect}_{\Theta'} \leq \mathrm{ect}_\Theta$, weil das Hinzunehmen weiterer Aktivitäten die früheste Fertigstellung zwar verzögern, aber nicht beschleunigen kann.

Weiterhin gilt $\mathrm{est}_{\Theta'} + \mathrm{p}_{\Theta'} \leq \mathrm{ect}_{\Theta'}$, weil vom frühestmöglichen Start einer Aktivitätenmenge bis zu deren frühestmöglichen Fertigstellung alle enthaltenen Aktivitäten Zeit finden müssen.

Insgesamt gilt also $\mathrm{est}_{\Theta'} + \mathrm{p}_{\Theta'} \leq \mathrm{ect}_\Theta$ für alle $\Theta' \subseteq \Theta$, und damit auch

$$\mathrm{ECT}(\Theta) \overset{(2.5)}{=} \max_{\Theta' \subseteq \Theta} \left\{ \mathrm{est}_{\Theta'} + \mathrm{p}_{\Theta'} \right\} \leq \mathrm{ect}_\Theta \ . \qquad \square$$

[3] Dies gilt ebenso für die übrigen in Abschnitt 2.2 vorgestellten Regeln.

Es bleibt noch zu zeigen, wie die Abschätzung $\mathrm{ECT}(\Theta)$ effizient berechnet werden kann. Dazu können zwei Verfahren angewandt werden, die beide auf der folgenden Proposition aufbauen:

Proposition 3.3 (Berechnung von $\mathrm{ECT}(\Theta)$[4])**.** *Sei T eine Menge von Aktivitäten und t ein beliebiger Zeitpunkt.*

Für jede Partition $L \cup R = T$, $L \cap R = \emptyset$ von T mit $\forall l \in L : \mathrm{est}_l \leq t$ und $\forall r \in R : \mathrm{est}_r \geq t$ gilt

$$\mathrm{ECT}(T) = \max \left\{ \mathrm{ECT}(L) + \sum_{j \in R} \mathrm{p}_j, \mathrm{ECT}(R) \right\} .$$

Die erste Berechnungsmethode sind die von Vilím [Vil04] eingeführten Θ-Bäume. Ein Θ-Baum ist ein balancierter Binärbaum, dessen Blätter die nach frühester Startzeit sortierten Aktivitäten aus Θ sind.[5] In Abbildung 3.1 ist ein Θ-Baum mit vier Aktivitäten zu sehen.

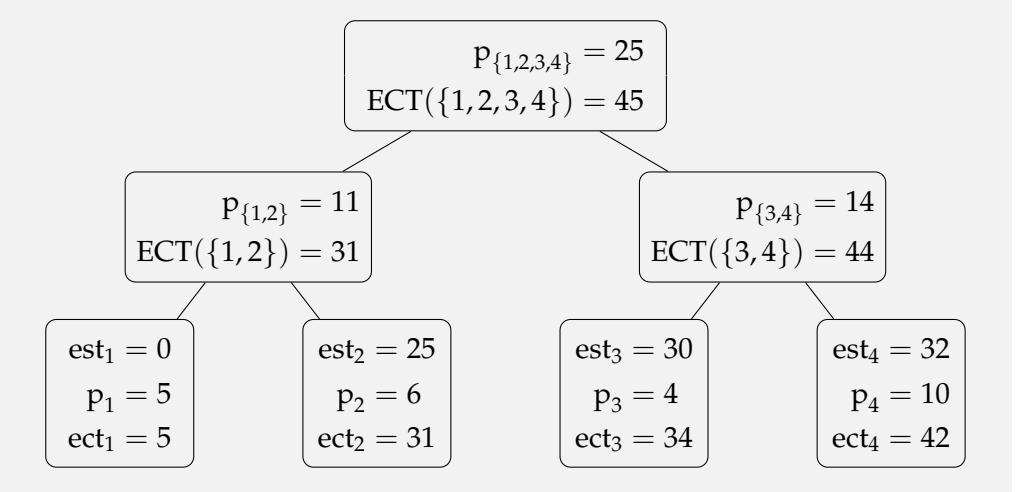


Abbildung 3.1: Ein Θ-Baum mit vier Aktivitäten, vgl. [VBČ05, Seite 406]

[4] Vgl. [VBČ05, Seite 405]. Dort findet sich auch ein Beweis.

[5] Es in der Praxis hat es sich als am effizientesten erwiesen, für Aktivitäten $i \in T \setminus \Theta$, die in der betrachteten exklusiven Ressource, aber nicht in Θ enthalten sind, dennoch Blätter vorzuhalten. Ihnen werden die Werte $\mathrm{p}_i = 0$ und $\mathrm{ect}_i = -\infty$ zugeordnet, sodass sie keinen Einfluss auf die Berechnung haben. Auf diese Weise werden teure Speicherverwaltungsoperationen vermieden, die sonst bei Änderungen an der Baumstruktur notwendig wären.

An den inneren Knoten k des Baums werden für die Aktivitäten in den Blättern des zu k gehörigen Teilbaums jeweils die Dauer p_k und die geschätzte früheste Fertigstellungszeit ECT_k gespeichert, die wie folgt aus den beiden Kindknoten $\mathrm{left}(k)$ und $\mathrm{right}(k)$ berechnet werden können:

$$\mathrm{p}_k = \mathrm{p}_{\mathrm{left}(k)} + \mathrm{p}_{\mathrm{right}(k)} \tag{3.1}$$

$$\mathrm{ECT}_k = \max\left\{\mathrm{ECT}_{\mathrm{left}(k)} + \mathrm{p}_{\mathrm{right}(k)}, \mathrm{ECT}_{\mathrm{right}(k)}\right\} \tag{3.2}$$

Während der erste Zusammenhang offensichtlich ist, ergibt sich der zweite aus Proposition 3.3. Für Blattknoten k_i zu einer Aktivität $i \in \Theta$ gilt $\mathrm{p}_{k_i} = \mathrm{p}_i$ und $\mathrm{ECT}_{k_i} = \mathrm{ect}_i$.

In der Wurzel r dieses Baums kann die geschätzte früheste Fertigstellungszeit von Θ mit konstantem Aufwand ausgelesen werden, weil aufgrund von Proposition 3.3 $\mathrm{ECT}(\Theta) = \mathrm{ECT}_r$ gilt. Der Preis hierfür ist, dass bei jeder Veränderung von Θ der Baum entsprechend aktualisiert werden muss. Aus Tabelle 3.1 ist der damit verbundene Aufwand ersichtlich.

Tabelle 3.1: Zeitaufwand von Operationen in Θ-Bäumen und der einfachen Schleife aus Listing 3.1. Θ_2 bezeichnet eine beliebige, von Θ verschiedene Menge.

Operation	Θ**-Baum** [Vil04, Seite 338]	**Schleife**
Initialisierung	$\mathcal{O}(n \log n)$ einmalig plus $\mathcal{O}(n)$ pro Baum	$\mathcal{O}(n \log n)$ einmalig
$\Theta := \Theta \cup \{i\}$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
$\Theta := \Theta \setminus \{i\}$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
$\mathrm{ECT}(\Theta)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$
$\mathrm{ECT}(\Theta \cup \{j\})$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$
$\mathrm{ECT}(\Theta_2)$	$\mathcal{O}(n)$ *(neuer Baum)*	$\mathcal{O}(n)$

Beim Hinzufügen und Entfernen einzelner Aktivitäten wird jeweils ausgenutzt, dass nur die Knoten auf dem Pfad vom Blattknoten der Aktivität zur Wurzel betrachtet werden müssen, weil sich im Rest des Baumes keine Veränderungen ergeben. Da ein balancierter Binärbaum nur $\mathcal{O}(\log n)$ tief ist, ist dies auch der Aufwand dieser Aktualisierungen. Da Θ in den Filteralgorithmen in der Regel nur einmal initialisiert, aber in einer Schleife n-mal aktualisiert und $\mathrm{ECT}(\Theta)$ ermittelt werden muss, bieten Θ-Bäume hierfür einen Zeitvorteil. Dieser ermöglichte es Vilím [Vil04], den Aufwand zahlreicher Filteralgorithmen für exklusive Ressourcen ohne optionale Aktivitäten von $\mathcal{O}(n^2)$ auf $\mathcal{O}(n \log n)$ zu drücken.

Eine Einschränkung der Θ-Bäume besteht darin, dass nur kleine, iterative Veränderungen an Θ vorgenommen werden können. Treten größere Änderungen auf – etwa wenn eine von Θ unabhängige Aktivitätenmenge Θ_2 betrachtet werden soll –, muss der komplette Baum neu aufgebaut werden. Dies verursacht einen Zeitaufwand von $\mathcal{O}(n)$, wobei sehr große konstante Faktoren auftreten. Wenn für eine Aktivitätenmenge Θ_2 nur einmalig die früheste Fertigstellungszeit berechnet werden soll, verbessert es daher die Laufzeit, wenn statt eines Θ-Baums ein spezialisiertes Verfahren zum Einsatz kommt.

Dazu kann eine Schleife verwendet werden, die über die Aktivitäten $j \in T$ in aufsteigender Reihenfolge ihrer frühesten Startzeit est_j iteriert. Im Rumpf dieser Schleife wird jeweils berechnet, welche Dauer und geschätzte früheste Fertigstellungszeit die bisher betrachteten Aktivitäten zusammen haben, wobei Aktivitäten aus $T \setminus \Theta$ nicht berücksichtigt werden. Bezeichne L die Menge der bisher betrachteten Aktivitäten, $L' := L \cup \{j\}$ die Menge der nach diesem Schleifendurchlauf betrachteten Aktivitäten und ect_L eine Variable, in der die geschätzte früheste Fertigstellungszeit von L gespeichert wird – es gilt also $ect_L = \text{ECT}(\Theta \cap L)$.[6]

Ausgehend von Proposition 3.3 auf Seite 43 kann $R := \{j\}$ und $T := L'$ gewählt werden, sodass sich $ect_{L'}$ wie folgt berechnen lässt:

$$ect_{L'} = \begin{cases} \max\left\{ect_L + \text{p}_j, \text{ect}_j\right\} & \text{falls } j \in \Theta \\ ect_L & \text{sonst} \end{cases} \tag{3.3}$$

Dabei fällt wie bei Θ-Bäumen einmalig ein Aufwand von $\mathcal{O}(n \log n)$ zur Sortierung der Aktivitäten nach est_i an. Das Auslesen von $\text{ECT}(\Theta)$ benötigt $\mathcal{O}(n)$, jedoch entfällt das Aktualisieren einer Datenstruktur, wenn Θ verändert wird.

Insgesamt ergibt sich folgender Algorithmus:

Listing 3.1: Berechnung von $\text{ECT}(\Theta)$ mit einer Schleife

```
1 // Voraussetzung: Die Aktivitäten liegen sortiert nach est_j vor.
2 ect := −∞ // ect initialisieren
3 for j ∈ T in aufsteigender Reihenfolge von est_j do
4     if j ∈ Θ then
5         aktualisiere ect gemäß (3.3)
6 return ect
```

[6] Der Index in ect_L dient nur der konzeptuellen Klarheit; bei der Implementierung genügt eine einfache Variable *ect* anstelle eines Arrays, weil immer nur auf den zuletzt berechneten Wert zugegriffen wird.

3.1.1. Auswahl zusätzlicher Aktivitäten

Bereits beim Edgefinding-Algorithmus von Vilím, Barták und Čepek [VBČ04] ist eine Erweiterung der Θ-Bäume nötig: Es muss (eine untere Schranke für) die früheste Fertigstellungszeit einer Aktivitätenmenge Θ plus eine zusätzliche Aktivität i aus einer zweiten Menge Λ ermittelt werden, wobei letztere danach ausgewählt werden soll, dass sich ein möglichst später Wert ergibt:

$$\begin{aligned} \mathrm{ECT}(\Theta, \Lambda) &:= \max_{i \in \Lambda} \{\mathrm{ECT}(\Theta \cup \{i\})\} \\ &\overset{(2.5)}{=} \max_{i \in \Lambda} \left\{ \max_{\Theta' \subseteq \Theta \cup \{i\}} \{\mathrm{est}_{\Theta'} + \mathrm{p}_{\Theta'}\} \right\} \end{aligned} \tag{3.4}$$

Die von Vilím, Barták und Čepek entwickelten Θ-Λ-Bäume bieten eine Möglichkeit, diesen Wert zu berechnen. Sie sind ähnlich wie Θ-Bäume aufgebaut: In den Blättern sind wieder die nach frühester Startzeit sortierten Aktivitäten zu finden, und in den inneren Knoten k werden Eigenschaften der Aktivitätenmengen der entsprechenden Teilbäume gespeichert. Neben p_k und ECT_k werden zusätzlich die Werte $\mathrm{p}\Lambda_k$ und $\mathrm{ECT}\Lambda_k$ in den Knoten gespeichert, die jeweils bis zu eine Aktivität $i \in \Lambda$ mit einbeziehen, sodass die jeweiligen Werte maximal werden. Dabei muss jeweils entschieden werden, ob eine Λ-Aktivität aus dem linken oder rechten Teilbaum einbezogen wird. Dies führt zu den folgenden Berechnungsvorschriften [VBČ05, Seite 415]:

$$\mathrm{p}\Lambda_k = \max \left\{ \mathrm{p}\Lambda_{\mathrm{left}(k)} + \mathrm{p}_{\mathrm{right}(k)}, \mathrm{p}_{\mathrm{left}(k)} + \mathrm{p}\Lambda_{\mathrm{right}(k)} \right\} \tag{3.5}$$

$$\begin{aligned} \mathrm{ECT}\Lambda_k = \max \Big\{ & \mathrm{ECT}_{\mathrm{left}(k)} + \mathrm{p}\Lambda_{\mathrm{right}(k)}, \\ & \mathrm{ECT}\Lambda_{\mathrm{left}(k)} + \mathrm{p}_{\mathrm{right}(k)}, \mathrm{ECT}\Lambda_{\mathrm{right}(k)} \Big\} \end{aligned} \tag{3.6}$$

Dabei werden die Werte in den Knoten k_i wie folgt initialisiert:

- Für Aktivitäten $i \in \Theta$ werden $\mathrm{p}_{k_i} = \mathrm{p}\Lambda_{k_i} = \mathrm{p}_i$ und $\mathrm{ECT}_{k_i} = \mathrm{ECT}\Lambda_{k_i} = \mathrm{ect}_i$ gesetzt.
- Für Aktivitäten $i \in \Lambda$ werden $\mathrm{p}_{k_i} = 0$, $\mathrm{p}\Lambda_{k_i} = \mathrm{p}_i$, $\mathrm{ECT}_{k_i} = -\infty$ und $\mathrm{ECT}\Lambda_{k_i} = \mathrm{ect}_i$ verwendet.
- Für Aktivitäten $i \in T \setminus (\Theta \cup \Lambda)$ werden wieder $\mathrm{p}_{k_i} = \mathrm{p}\Lambda_{k_i} = 0$ und $\mathrm{ECT}_{k_i} = \mathrm{ECT}\Lambda_{k_i} = -\infty$ gesetzt, sodass sie keine Auswirkungen haben.

Diese Erweiterung der Θ-Bäume lässt die asymptotische Zeitkomplexität aus Tabelle 3.1 auf Seite 44 unverändert, wobei nun auch inkrementelle Änderungen an Λ einen Aufwand von $\mathcal{O}(\log n)$ verursachen.

Der Wert $\mathrm{ECT}(\Theta, \Lambda)$ kann auch wieder mit der Schleifenmethode aus Listing 3.1 berechnet werden. Zusätzlich zu ect_L muss dazu im Schleifenrumpf eine zusätzliche Variable $ect\Lambda_L = \mathrm{ECT}(\Theta \cap L, \Lambda \cap L)$ aktualisiert werden, die zu Beginn wie $v\,\mathrm{ECT}$ mit ∞ initialisiert wird:

$$ect\Lambda_{L'} := \begin{cases} \max\{ect\Lambda_L + \mathrm{p}_j, \mathrm{ect}_j\} & \text{falls } j \in \Theta \\ \max\{ect\Lambda_L, \mathrm{ect}_j, ect_L + \mathrm{p}_j\} & \text{falls } j \in \Lambda \\ ect\Lambda_L & \text{sonst} \end{cases} \tag{3.7}$$

Durch diese Berechnungsvorschrift ist sichergestellt, dass niemals mehr als eine Aktivität aus Λ für $ect\Lambda_{L'}$ einbezogen wird: Im 2. Fall ($j \in \Lambda$) wird die für $ect\Lambda_L$ berücksichtigte Aktivität verworfen, wenn nun stattdessen j einbezogen wird. Dies geschieht durch die Verwendung von $ect_L + \mathrm{p}_j$ anstelle von $ect\Lambda_L + \mathrm{p}_j$.

Auch bei der Schleifenberechnung bleibt die asymptotische Komplexität aus Tabelle 3.1 unverändert.

Die für $\mathrm{ECT}(\Theta, \Lambda)$ einbezogene Aktivität $i \in \Lambda$ wird in dem im weiteren Verlauf dieses Kapitels vorgestellten Edgefinding-Algorithmus weiterverwendet. Sie kann direkt mitermittelt werden, indem die Baumknoten zusätzlich die für $\mathrm{ECT}\Lambda$ und $\mathrm{p}\Lambda$ verwendeten Λ-Aktivitäten speichern. Je nachdem, welcher der Terme sich bei der Maximumsbildung durchsetzt, werden diese Aktivitäten von den Kindknoten übernommen:

Listing 3.2: Speichern der verwendeten Λ-Aktivität bei der Maximumsbildung

```
1  // es soll folgender Wert berechnet werden:
2  // ECTΛ_k = max{ECT_left(k) + pΛ_right(k), ECTΛ_left(k) + p_right(k), ECTΛ_right(k)}
3  r = rechtes Kind von k
4  l = linkes Kind von k
5
6  ectΛ_k = ectΛ_r                  // 3. Alternative
7  ectΛakt_k = ectΛakt_r             // verwendete Aktivität übernehmen
8
9  if ectΛ_k < ect_l + pΛ_r then     // 1. Alternative
10     ectΛ_k = ect_l + pΛ_r
11     ectΛakt_k = pΛakt_r           // verwendete Aktivität übernehmen
12
13 if ectΛ_k < ectΛ_l + p_r then     // 2. Alternative
14     ectΛ_k = ectΛ_l + p_r
15     ectΛakt_k = ectΛakt_l         // verwendete Aktivität übernehmen
```

Die zur Speicherung der einbezogenen Aktivitäten verwendeten Variablen müssen dann auch in den Blattknoten gesetzt werden: Im zu einer Aktivität $o \in T_{\text{optional}}$ gehörigen Blatt k_o wird $ect\text{⟂}akt_{k_o} := p\text{⟂}akt_{k_o} := o$ gesetzt, während im zu einer Aktivität $i \in T_{\text{enabled}}$ gehörigen Knoten $ect\text{⟂}akt_{k_i} := p\text{⟂}akt_{k_i} := \bot$ gesetzt wird.

Bei der Berechnung durch die Schleife ist dieses Verfahren analog anwendbar. In diesem Fall wird die für $ect\text{⟂}_L$ einbezogene Aktivität in einer zusätzlichen Variable gespeichert.

3.1.2. Früheste Fertigstellungszeit bei optionalen Aktivitäten

Um den Edgefinding-Algorithmus für optionale Aktivitäten formulieren zu können, wird eine weitere Variante der Berechnung der frühesten Fertigstellungszeit benötigt, die eine zusätzliche Aktivität aus einer dritten Menge Ω auswählt. Im Gegensatz zur Auswahl aus Λ, bei der es zulässig ist keine Aktivität auszuwählen, soll aus Ω selbst dann eine Aktivität berücksichtigt werden, wenn dadurch ein früherer Wert für ECT zustande kommt.

$\mathrm{ECT}(\Theta, \Omega, \Lambda)$ bezeichnet deshalb eine untere Schranke der frühesten Fertigstellungszeit

- aller Aktivitäten aus Θ,
- bis zu einer Aktivität $j \in \Lambda$ und
- genau einer Aktivität $o \in \Omega$ (sofern $\Omega \neq \emptyset$),

wobei j und o so gewählt werden, dass der Wert maximal wird:

$$\mathrm{ECT}(\Theta, \Omega, \Lambda) := \max_{o \in \Omega} \left\{ \max_{j \in \Lambda} \left\{ \max_{\Theta' \subseteq \Theta \cup \{j\}} \left\{ \mathrm{est}_{\Theta' \cup \{o\}} + \mathrm{p}_{\Theta' \cup \{o\}} \right\} \right\} \right\} \tag{3.8a}$$

Diese Definition muss noch für $\Omega = \emptyset$ erweitert werden:

$$\mathrm{ECT}(\Theta, \emptyset, \Lambda) := \mathrm{ECT}(\Theta, \Lambda) \tag{3.8b}$$

Auch dieser Wert kann durch eine Baumstruktur berechnet werden. Da in dem in Abschnitt 3.2 vorgestellten Algorithmus Λ dynamisch berechnet wird, ist die Verwendung der Schleifenmethode effizienter. Bei letzterer werden wieder alle am *SingleResource*-Constraint beteiligten Aktivitäten $j \in T$ sortiert nach ihrer frühesten Startzeit est_j betrachtet, wobei L die Menge der bisher betrachteten Aktivitäten bezeichnet.

Es müssen dazu neben $ect_L = \text{ECT}(\Theta \cap L)$ (vgl. (3.3) auf Seite 45) und $ect\Lambda_L = \text{ECT}(\Theta \cap L, \Lambda \cap L)$ (vgl. (3.7) auf Seite 47) auch die folgenden Werte berechnet werden:

- Die geschätzte früheste Fertigstellungszeit der Aktivitäten aus Θ plus genau einer Aktivität aus Ω (sofern es eine gibt). Dieser Wert wird in der Variable $ect\Omega_L = \text{ECT}(\Theta \cap L, \Omega \cap L, \emptyset)$ gespeichert.
- Die geschätzte früheste Fertigstellungszeit von Θ, bis zu einer Aktivität aus Λ und genau einer aus Ω (sofern letztere existiert). Dieser Wert wird in der Variable $ect\Omega\Lambda_L = \text{ECT}(\Theta \cap L, \Omega \cap L, \Lambda \cap L)$ gespeichert.[7]

Die beiden Variablen $ect\Omega_L$ und $ect\Omega\Lambda_L$ können wie folgt aktualisiert werden:

$$ect\Omega_{L'} := \begin{cases} \max\{ect\Omega_L + \mathrm{p}_j, \mathrm{ect}_j\} & \text{falls } j \in \Theta \wedge \Omega \cap L = \emptyset \\ ect\Omega_L + \mathrm{p}_j & \text{falls } j \in \Theta \wedge \Omega \cap L \neq \emptyset \\ \max\{\mathrm{ect}_j, ect_L + \mathrm{p}_j\} & \text{falls } j \in \Omega \wedge \Omega \cap L = \emptyset \\ \max\{\mathrm{ect}_j, ect_L + \mathrm{p}_j, ect\Omega_L\} & \text{falls } j \in \Omega \wedge \Omega \cap L \neq \emptyset \\ ect\Omega_L & \text{sonst} \end{cases} \tag{3.9}$$

$$ect\Omega\Lambda_{L'} := \begin{cases} \max\{ect\Omega\Lambda_L + \mathrm{p}_j, \mathrm{ect}_j\} & \text{falls } j \in \Theta \wedge \Omega \cap L = \emptyset \\ ect\Omega\Lambda_L + \mathrm{p}_j & \text{falls } j \in \Theta \wedge \Omega \cap L \neq \emptyset \\ \max\{\mathrm{ect}_j, ect_L + \mathrm{p}_j, ect\Lambda_L + \mathrm{p}_j\} & \text{falls } j \in \Omega \wedge \Omega \cap L = \emptyset \\ \max\{\mathrm{ect}_j, ect_L + \mathrm{p}_j, ect\Lambda_L + \mathrm{p}_j, ect\Omega\Lambda_L\} & \text{falls } j \in \Omega \wedge \Omega \cap L \neq \emptyset \\ \max\{ect\Omega\Lambda_L, ect\Omega_L + \mathrm{p}_j, \mathrm{ect}_j\} & \text{falls } j \in \Lambda \wedge \Omega \cap L = \emptyset \\ \max\{ect\Omega\Lambda_L, ect\Omega_L + \mathrm{p}_j\} & \text{falls } j \in \Lambda \wedge \Omega \cap L \neq \emptyset \\ ect\Omega\Lambda_L & \text{sonst} \end{cases} \tag{3.10}$$

Dabei wurde in den Fällen mit $\Omega \cap L = \emptyset$ jeweils noch keine Aktivität aus Ω berücksichtigt. Bei $ect\Omega_L$ wirkt sich das dahingehend aus, dass im 1. Fall die bisher ausgewählte Aktivität zugunsten von j aufgegeben werden kann. Dies wird im 2. Fall nicht zugelassen, weil sonst eine Aktivität aus Ω unberücksichtigt bleiben könnte. Analog erzwingt der 3. Fall (abweichend vom 4.) die Einbeziehung der ersten optionalen Aktivität. Im Übrigen entsprechen der 2., 4. und 5. Fall den drei Fällen bei der Berechnung von $ect\Lambda_L$ in (3.7) auf Seite 47.

Etwas komplizierter verhält es sich bei der Berechnung von $ect\Omega\Lambda_L$: Auch hier darf eine Aktivität aus Ω nur zugunsten einer anderen Ω-Aktivität aufgegeben

[7] Dabei dient der Index wie schon zuvor ausschließlich der konzeptuellen Klarheit. Wenn im Schleifenrumpf zuerst $ect\Omega\Lambda$ und zuletzt ect aktualisiert werden, reichen einfache Variablen zum Speichern der Werte aus. Die genannte Reihenfolge ist notwendig, weil bei der Aktualisierung teilweise auf die vorigen Werte der übrigen Variablen zugegriffen wird.

werden, zusätzlich können aber auch Aktivitäten aus Λ einbezogen werden, für die diese Beschränkung nicht gilt. Ersteres hat zur Folge, dass ect_j für $j \in \Theta$ und $j \in \Lambda$ nur dann in die Maximumsbildung einbezogen werden, wenn $\Omega \cap L \neq \emptyset$. Umgekehrt wird in diesem Fall bei $j \in \Omega$ die Einbeziehung von j erzwungen.

Wie schon bei $\text{ECT}(\Theta, \Lambda)$ kann auch bei der Berechnung von $\text{ECT}(\Theta, \Omega, \Lambda)$ gleich mitermittelt werden, welche Aktivitäten aus Ω und Λ mit einbezogen wurden.

3.2. Ein neuer Edgefinding-Algorithmus

Damit sind Voraussetzungen für die Formulierung des Edgefinding-Algorithmus mit optionalen Aktivitäten vorhanden. Dieser für diese Arbeit zentrale Algorithmus ist in Listing 3.3 dargestellt. Zu einem *SingleResource*-Constraint mit Aktivitätenmenge T findet er alle Anwendungen der Edgefinding-Regel 2.6 auf Seite 30 und der Überlast-Regel 2.4 auf Seite 28 auf Aktivitäten in T_{enabled}. Darüberhinaus deaktiviert der Algorithmus Aktivitäten aus T_{optional}, wenn ihre Aktivierung zu einer durch die Edgefinding-Regel erkennbaren Überlast führten. Dies ist der Fall, wenn die Edgefinding-Regel eine Einschränkung erlaubt, durch welche die Domäne einer Variable leer würde.

Dabei bezeichnet $\Lambda'(o)$ in Zeile 27 die folgende Menge:

$$\begin{aligned}\Lambda'(o) := \Big\{ i \in T_{\text{enabled}} \setminus \Theta \;\Big|\; & \text{lst}_i < \text{ECT}(\Theta \cup \{o\}) \\ & \wedge\ \nexists\, i_2 \in T_{\text{enabled}} \setminus \Theta : \Big(\text{lst}_{i_2} \geq \text{ECT}(\Theta \cup \{o\}) \\ & \qquad \wedge \text{ECT}(\Theta \cup \{i_2\}) > \text{ECT}(\Theta \cup \{i\}) \Big) \Big\} \end{aligned} \tag{3.11}$$

Bei der Ausführung des Algorithmus ist zweierlei möglich:

1. Es können die Wertebereiche derjenigen Variablen eingeschränkt werden, die die Eigenschaften der beteiligten Aktivitäten $i \in T$ beschreiben. Dies betrifft die Eigenschaften est_i und status_i. (In der symmetrischen Fassung des Algorithmus sind es lct_i und status_i.)

2. Es kann eine Inkonsistenz erkannt werden. Dies ist der Fall, wenn die Überlast-Regel anwendbar ist und `fail` in Zeile 13 erreicht wird, oder wenn der Wertebereich einer Variable so weit eingeschränkt wird, dass er leer wird. Letzteres ist beim Deaktivieren von i in Zeile 18 möglich.

Listing 3.3: Edgefinding mit optionalen Aktivitäten

```
1  for i ∈ T do // Änderungen an est_i zwischenspeichern:
2     est'_i := est_i // sonst würden die Θ-Bäume durcheinander kommen
3
4  Q := Warteschlange aller j ∈ T \ T_disabled, absteigend sortiert nach lct_j
5  j := Q.first // gehe zur ersten Aktivität in Q
6  (Θ, Ω, Λ) = (T_enabled, T_optional, ∅) // Bäume initialisieren
7  repeat
8     if status_j ≠ disabled then // verschiebe j nach Λ
9        (Θ, Ω, Λ) := (Θ \ {j}, Ω \ {j}, Λ ∪ {j})
10    Q.dequeue; j := Q.first // gehe zur nächsten Aktivität in Q
11    if status_j = enabled then
12       if ECT(Θ) > lct_j then
13          fail // Die Überlast-Regel ist anwendbar
14       while ECT(Θ, Λ) > lct_j do
15          i := die für ECT(Θ, Λ) verantwortliche Λ-Aktivität
16          if ECT(Θ) > est_i then // Edgefinding-Regel ist anwendbar
17             if ECT(Θ) > lst_i then // Inkonsistenz erkannt
18                status_i := disabled // schlägt fehl für i ∈ T_enabled
19             else
20                est'_i := ECT(Θ) // Edgefinding-Regel anwenden
21          Λ := Λ \ {i} // keine bessere Einschränkung für i möglich
22       while ECT(Θ, Ω) > lct_j do // Überlast-Regel ist anwendbar
23          o := die für ECT(Θ, Ω) verantwortliche Ω-Aktivität
24          status_o := disabled
25          Ω := Ω \ {o}
26       while Ω ≠ ∅
27       and ECT(Θ, Ω, Λ'(o)) > lct_j do // Λ'(o) wird in (3.11) definiert
28          // Edgefinding-Regel erkennt Überlast
29          o := die für ECT(Θ, Ω, Λ'(o)) verantwortliche Ω-Aktivität
30                          // schon in Zeile 27 mit dieser Bedeutung verwendet
31          status_o := disabled
32          Ω := Ω \ {o}
33    else if status_j = optional then
34       if ECT(Θ ∪ {j}) > lct_j // Überlast-Regel anwendbar ...
35       or ECT(Θ ∪ {j}, {i ∈ T_enabled \ Θ | lst_i < ECT(Θ ∪ {j})}) > lct_j
36       then // ... oder die Edgefinding-Regel erkennt Überlast
37          status_j := disabled
38          Ω := Ω \ {j} // keine weiteren Einschränkungen für j möglich
39 until j = Q.last
40
41 for i ∈ T do
42    est_i := est'_i // zwischengespeicherte Änderungen anwenden
```

Zum leichteren Verstehen des Algorithmus wird zunächst der Inhalt der Mengen Θ, Λ und Ω betrachtet:

Lemma 3.4. *In jedem Durchlauf der* `repeat`*-Schleife gilt:*

- *Θ enthält alle Aktivitäten aus T_{enabled}, die nicht vor j in der Warteschlange Q einsortiert wurden. Dadurch gilt für alle $i \in \Theta$:* $\text{lct}_i \leq \text{lct}_j$.
- *Ω enthält alle Aktivitäten aus T_{optional}, die nicht vor j in der Warteschlange Q einsortiert wurden. Wieder gilt für alle $o \in \Omega$:* $\text{lct}_o \leq \text{lct}_j$.
- *Λ enthält (manche, aber nicht alle) Aktivitäten aus $T_{\text{enabled}} \cup T_{\text{optional}}$, die vor j in der Warteschlange einsortiert wurden. Es gilt für alle $i \in \Lambda$:* $\text{lct}_i \geq \text{lct}_j$.

Dabei ist zu beachten, dass in manchen Fällen Aktivitäten von T_{enabled} oder T_{optional} nach T_{disabled} verschoben werden, die dann auch in keiner der drei Mengen Θ, Ω und Λ mehr vorkommen.

Die Aktivität j ist entweder in Θ oder in Ω enthalten, je nachdem ob j in T_{enabled} oder T_{optional} enthalten ist. Im übrigen folgt aus dem Lemma, dass die drei Mengen Θ, Ω und Λ disjunkt sind.

Beweis durch Induktion. Induktionsanfang: Zu Beginn werden in Zeile 6 die Mengen als $\Theta = T_{\text{enabled}}$, $\Omega = T_{\text{optional}}$ und $\Lambda = \emptyset$ initialisiert. Dies ist korrekt, weil j zu diesem Zeitpunkt die erste Aktivität in Q ist.

Für den Induktionsschluss muss gezeigt werden, dass die Aussage des Lemmas, wenn sie zu Beginn der `repeat`-Schleife zutrifft, dies auch an deren Ende gilt: Wenn j in Zeile 10 zur nächsten Aktivität in Q weiterwandert, wird direkt zuvor die bisherige Aktivität j nach Λ verschoben – es sei denn, sie ist bereits deaktiviert worden. Umgekehrt wird jedes Mal, wenn der Status einer Aktivität auf disabled geändert wird (andere Änderungen kommen nicht vor), diese Aktivität aus der Menge entfernt, in der sie enthalten war: Bei den Statusänderungen in den Zeilen 24, 31 und 37 geschieht dies jeweils in der darauffolgenden Zeile, bei der Statusänderung in Zeile 18 geschieht es in Zeile 21 nach der Fallunterscheidung. Dies ist auch die einzige Stelle, an der eine Aktivität aus den Mengen entfernt werden kann, ohne dass diese zuvor zwingend deaktiviert wird. Da es sich um Λ handelt und Λ nicht *alle* in Frage kommenden Aktivitäten umfassen muss, ist dies jedoch zulässig. □

Während j über die Warteschlange Q iteriert, nimmt lct_j immer weiter ab, da Q absteigend nach spätester Fertigstellungszeit sortiert wurde. Dementsprechend wandern immer mehr Aktivitäten von Θ und Ω nach Λ.

Wie zahlreiche andere Filteralgorithmen ist auch der Algorithmus aus Listing 3.3 auf der vorherigen Seite nicht von sich aus idempotent. Es muss daher ein Fixpunkt gebildet werden, um den Algorithmus für die Constraint-Propagation nutzbar zu machen. Dabei kann der Edgefinding-Algorithmus

auch mit anderen Filteralgorithmen kombiniert werden, was auf die in Abschnitt 2.2.6 beschriebene Weise bewerkstelligt werden kann.

3.2.1. Wohldefiniertheit

An dieser Stelle muss noch gezeigt werden, dass der Algorithmus wohldefiniert ist, also alle Anweisungen ausführbar sind. Bei den meisten ist dies unkritisch, einige verlangen aber nach mehr Aufmerksamkeit. Zum einen handelt es sich dabei um die Anweisungen, die auf jene Aktivitäten in Λ beziehungsweise Ω zugreifen, die für $\mathrm{ECT}(\Theta, \dots)$ ausgewählt wurden. Für diese Werte müssen aber nicht zwingend entsprechende Aktivitäten mit einbezogen worden sein – die in Listing 3.2 auf Seite 47 angegebene Berechnungsmethode liefert in diesem Fall statt einer Aktivität $\perp$ zurück. An allen drei Stellen, an denen solche Anweisungen im Algorithmus auftreten, ist dies jedoch ausgeschlossen:

- In Zeile 15 geht dem Zugriff auf die an $\mathrm{ECT}(\Theta, \Lambda)$ beteiligte Λ-Aktivität die Bedingung $\mathrm{ECT}(\Theta, \Lambda) > \mathrm{lct}_j$ voraus. Zuvor wurde aber schon für den Fall $\mathrm{ECT}(\Theta) > \mathrm{lct}_j$ ein `fail` abgesetzt, sodass an der betrachteten Stelle $\mathrm{ECT}(\Theta) \leq \mathrm{lct}_j$ gelten muss. Daraus folgt, dass wegen
$$\mathrm{ECT}(\Theta, \Lambda) \overset{(3.4)}{=} \max_{i \in \Lambda} \left\{ \max_{\Theta' \subseteq \Theta \cup \{i\}} \left\{ \mathrm{est}_{\Theta'} + \mathrm{p}_{\Theta'} \right\} \right\} > \mathrm{lct}_j$$
in der gewählten Menge Θ' eine Aktivität $i \in \Lambda$ enthalten sein muss.
- In Zeile 23 gilt die gleiche Argumentation, nur wird hier Ω statt Λ verwendet.
- In Zeile 29 ist durch das in (3.8) auf Seite 48 definierte $\mathrm{ECT}(\Theta, \Omega, \Lambda)$ sichergestellt, dass für $\Omega \neq \emptyset$ immer ein $o \in \Omega$ einbezogen wird. Der Fall $\Omega = \emptyset$ wird in der ersten Bedingung der `while`-Schleife ausgeschlossen (Zeile 26).

Ebenfalls problematisch ist die Berechnung von $\mathrm{ECT}(\Theta, \Omega, \Lambda'(o))$ in Zeile 27, weil hier mit o bereits die aus Ω ausgewählte Aktivität referenziert wird und damit die Auswahl eines optimalen i und eines optimalen o nicht unabhängig voneinander stattfinden kann. Die Definition von $\Lambda'(o)$ in (3.11) auf Seite 50 ermöglicht es jedoch, die Auswahl von $o \in \Omega$ und $i \in \Lambda$ während des einen Schleifendurchlaufs zur Berechnung der frühesten Fertigstellungszeit vorzunehmen. Dazu muss von den Definitionen von $ect\Omega_{\Lambda_{L'}}$ in (3.10) auf Seite 49 und $ect\Lambda_{L'}$ in (3.7) auf Seite 47 in der folgenden Weise abgewichen werden:

- Bei der Berechnung von $ect\Lambda_{L'}$ werden alle $i \in T_{\text{enabled}} \setminus \Theta$ berücksichtigt (nicht nur die aus $\Lambda'(o)$), da noch unklar ist, für welches $o \in \Omega$ die Bedingung $\mathrm{lst}_i < \mathrm{ECT}(\Theta \cup \{o\})$ erfüllt sein muss.

- Bei der Berechnung von $ect\Omega\Lambda_{L'}$ wird in den Fällen mit $j \in \Omega$ der Wert $ect\Lambda_L + \mathrm{p}_j$ im Gegenzug nur dann in die Maximumsbildung einbezogen, wenn $\mathrm{lst}_i < \mathrm{ECT}(\Theta \cup \{j\})$, wobei i die für $ect\Lambda_L$ aus $T_{\text{enabled}} \setminus \Theta$ ausgewählte Aktivität bezeichnet.
- Außerdem wird bei der Berechnung von $ect\Omega\Lambda_{L'}$ in den Fällen mit $j \in \Lambda$ der Wert $ect\Omega_L + \mathrm{p}_j$ nur dann in die Maximumsbildung einbezogen, wenn $\mathrm{lst}_j < \mathrm{ECT}(\Theta \cup \{o\})$, wobei o die für $ect\Omega_L$ aus Ω ausgewählte Aktivität bezeichnet.

Auf diese Weise wird sichergestellt, dass für $ect\Omega\Lambda$ niemals zwei Werte $o \in \Omega$ und $i \in T_{\text{enabled}} \setminus \Theta$ ausgewählt werden, welche die Bedingung $\mathrm{lst}_i < \mathrm{ECT}(\Theta \cup \{o\})$ verletzten.

Die in (3.11) auf Seite 50 für $i \in \Lambda'(o)$ zusätzlich geforderte Eigenschaft

$$\nexists i_2 \in T_{\text{enabled}} \setminus \Theta : \Big(\mathrm{lst}_{i_2} \geq \mathrm{ECT}(\Theta \cup \{o\}) \wedge \mathrm{ECT}(\Theta \cup \{i_2\}) > \mathrm{ECT}(\Theta \cup \{i\}) \Big)$$

beschreibt die Tatsache, dass bei dem gerade geschilderten Vorgehen Kombinationen von o und i übersehen werden können, wenn ein solches i_2 existiert: In diesem Fall würde i_2 anstelle von i für $ect\Lambda_L$ berücksichtigt werden (weil $\mathrm{ECT}(\Theta \cup \{i_2\}) > \mathrm{ECT}(\Theta \cup \{i\})$ gilt), aber wegen $\mathrm{lst}_{i_2} \geq \mathrm{ECT}(\Theta \cup \{o\})$ kann $ect\Lambda_L$ nicht bei der Berechnung von $ect\Omega\Lambda_{L'}$ herangezogen werden, obwohl mit i eine geeignete Aktivität zur Verfügung stünde.

Aus diesem Grund berechnet die angegebene Methode gerade den Wert $\mathrm{ECT}(\Theta, \Omega, \Lambda'(o))$.

3.2.2. Korrektheit

Korrektheit ist eine notwendige Bedingung zum Einsatz eines Filteralgorithmus. Korrektheit bedeutet in diesem Zusammenhang, dass keine unzulässigen Einschränkungen vorgenommen werden. Bei der Suche würden sonst Teile des Suchbaums übersprungen, obwohl sie noch zu Lösungen führen könnten.

Satz 3.5 (Korrektheit)**.** *Der in Listing 3.3 auf Seite 51 vorgestellte Edgefinding-Algorithmus für optionale Aktivitäten trifft nur zulässige Einschränkungen.*

Beweis. Es müssen alle Einschränkungen betrachtet und auf Zulässigkeit untersucht werden. Im vorliegenden Fall ergeben sich alle Einschränkungen aus der Edgefinding-Regel 2.6 und der Überlast-Regel 2.4:

- Das `fail` in Zeile 13 wird nur erreicht, wenn die Bedingung $\mathrm{ECT}(\Theta) > \mathrm{lct}_j$ erfüllt ist. Nach Lemma 3.4 gilt $\Theta \subseteq T_{\text{enabled}}$ und $\mathrm{lct}_j \geq \mathrm{lct}_\Theta$. Damit folgt $\mathrm{ECT}(\Theta) > \mathrm{lct}_\Theta$ und die Bedingung der Überlast-Regel ist erfüllt. Da nur obligatorische Aktivitäten beteiligt sind, ist eine Überlast aufgetreten.

- Das $\text{est}'_i := \text{ECT}(\Theta)$ in Zeile 20 wird nur dann erreicht, wenn die Bedingungen $\text{ECT}(\Theta, \Lambda) > \text{lct}_j$ und $j \in T_{\text{enabled}}$ erfüllt sind. Weil i die für $\text{ECT}(\Theta, \Lambda)$ aus Λ ausgewählte Aktivität bezeichnet, gilt $\text{ECT}(\Theta \cup \{i\}) > \text{lct}_j$. Mit $\text{lct}_j \geq \text{lct}_\Theta$ aus Lemma 3.4 folgt $\text{ECT}(\Theta \cup \{i\}) > \text{lct}_\Theta$. Damit ist die in Proposition 2.7 genannte Variante der Vorbedingung zur Edgefinding-Regel erfüllt. Da zusätzlich in Zeile 16 $\text{ECT}(\Theta) > \text{est}_i$ gefordert wurde, kann die Einschränkung $\text{est}_i := \max\{\text{est}_i, \text{ECT}(\Theta)\}$ der Edgefinding-Regel zu $\text{est}_i := \text{ECT}(\Theta)$ vereinfacht werden. Da unter den beteiligten Aktivitäten höchstens i selbst optional sein kann und bei optionalem i keine Inkonsistenz verursacht wird (es gilt $\text{est}'_i = \text{ECT}(\Theta) \leq \text{lst}_i$, vgl. Zeile 17), ist diese Einschränkung zulässig.

- Mit $\text{status}_i :=$ disabled in Zeile 18 wird der von der vorigen Einschränkung abweichende Fall erfasst, dass $\text{ECT}(\Theta) > \text{lst}_i$ wird. Dadurch würden in der Domäne von $start_i$ keine Werte mehr übrigbleiben. Wenn i optional ist, muss es also deaktiviert werden – genau das geschieht durch die betrachtete Einschränkung. Ist hingegen i obligatorisch, verursacht die abgesetzte Einschränkung genauso eine Inkonsistenz, wie das auf die Edgefinding-Regel gegründete $\text{est}_i := \text{ECT}(\Theta)$.

- Die nächste Einschränkung ist $\text{status}_o :=$ disabled in Zeile 24. Die Anweisung kann nur erreicht werden, wenn $j \in T_{\text{enabled}}$ und $\text{ECT}(\Theta, \Omega) > \text{lct}_j$, wobei o die dafür aus Ω ausgewählte Aktivität bezeichnet. Es gilt also $\text{ECT}(\Theta \cup \{o\}) > \text{lct}_j$. Außerdem gilt nach Lemma 3.4 $\text{lct}_j \geq \text{lct}_{\Theta \cup \Omega}$ und damit $\text{lct}_j \geq \text{lct}_{\Theta \cup \{o\}}$, was die Vorbedingung der Überlast-Regel darstellt. Da mit o genau eine optionale Aktivität beteiligt ist, kann o deaktiviert werden.

- Eine weitere Einschränkung ist mit $\text{status}_o :=$ disabled auf Zeile 31 gegeben. Sie kann nur erreicht werden, wenn $\text{status}_j =$ enabled, $\Omega \neq \emptyset$ und $\text{ECT}(\Theta, \Omega, \Lambda'(o)) > \text{lct}_j$ gilt, wobei o die aus Ω ausgewählte Aktivität bezeichnet. Es gilt also für ein $i \in \Lambda'(o)$ die Ungleichung $\text{ECT}(\Theta \cup \{o, i\}) < \text{lct}_j$. Darüberhinaus muss wegen der Definition von $\Lambda'(o)$ in (3.11) auf Seite 50 $\text{lst}_i < \text{ECT}(\Theta \cup \{o\})$ gelten.

  Mit Lemma 3.4 folgt wieder $\text{lct}_j \geq \text{lct}_{\Theta \cup \{o\}}$, sodass durch die Edgefinding-Regel die Einschränkung $\text{est}_i := \max\{\text{est}_i, \text{ECT}(\Theta \cup \{o\})\}$ ermöglicht wird. Mit $\text{lst}_i < \text{ECT}(\Theta \cup \{o\})$ folgt, dass die Domäne von $start_i$ leer würde und o deaktiviert werden kann, weil o die einzige optionale der beteiligten Aktivitäten ist.

- Die letzte Einschränkung im Algorithmus, $\text{status}_j :=$ disabled, findet sich in Zeile 37. Beim Erreichen dieser Stelle muss $\text{ECT}(\Theta \cup \{j\}) > \text{lct}_j$ oder

$\mathrm{ECT}(\Theta \cup \{j\}, \{i \in T_{\text{enabled}} \setminus \Theta \mid \mathrm{lst}_i < \mathrm{ECT}(\Theta \cup \{j\})\}) > \mathrm{lct}_j$ gelten. Im ersten Fall ist wegen $\mathrm{lct}_j \geq \mathrm{lct}_\Theta$ die Überlast-Regel anwendbar; im zweiten Fall gilt $\mathrm{ECT}(\Theta \cup \{j, i\}) > \mathrm{lct}_j = \mathrm{lct}_{\Theta \cup \{j\}}$, sodass die Edgefinding-Regel die Einschränkung $\mathrm{est}_i := \max\{\mathrm{est}_i, \mathrm{ECT}(\Theta \cup \{j\})\}$ zulassen würde. Da aber $\mathrm{lst}_i < \mathrm{ECT}(\Theta \cup \{j\})$ gilt, würde die Domäne von $start_i$ leerlaufen und dadurch wie im ersten Fall eine Inkonsistenz auftreten. Da j in beiden Fällen die einzige optionale Aktivität ist, kann j deaktiviert werden. □

3.2.3. Komplexität

In diesem Abschnitt wird gezeigt, dass der in Listing 3.3 auf Seite 51 vorgestellte Edgefinding-Algorithmus für optionale Aktivitäten mit einem Zeitaufwand von $\mathcal{O}(n^2)$ auskommt, wobei n die Anzahl der beteiligten Aktivitäten bezeichnet.

Das Kopieren von est nach est′ und zurück (Zeilen 1 bis 2 und 41 bis 42) benötigt $\mathcal{O}(n)$. Das Sortieren der Aktivitäten und Initialisieren der Θ-Bäume (Zeilen 4 bis 6) kann jeweils in $\mathcal{O}(n \log n)$ erfolgen. Es muss also nur noch die große `repeat`-Schleife betrachtet werden.

Lemma 3.6. *Jede Einzelanweisung in der großen `repeat`-Schleife (Zeilen 7 bis 39) benötigt höchstens $\mathcal{O}(n)$ Zeit.*

Beweis. Die meisten Anweisungen sind unkritisch und benötigen offensichtlich nur konstante Zeit. Die Übrigen werden im Folgenden betrachtet:

- Die Änderungen an Θ, Ω und Λ müssen an die entsprechenden Baumstrukturen weitergegeben werden. Da es sich in allen Fällen um inkrementelle Änderungen handelt, die nur eine einzelne Aktivität aus einer der Mengen entfernen und in einigen Fällen zu einer anderen hinzufügen, ist dies in $\mathcal{O}(\log n)$ möglich (vgl. Tabelle 3.1 auf Seite 44).

- Die Zugriffe auf die frühesten Fertigstellungszeiten $\mathrm{ECT}(\Theta)$ (Zeilen 12, 16, 17 und 20), $\mathrm{ECT}(\Theta, \Omega)$ (Zeile 22) sowie $\mathrm{ECT}(\Theta, \Lambda)$ (Zeile 14) können durch die Verwendung von Θ-Λ-Bäumen in konstanter Zeit erfolgen. Dabei bietet es sich an, anstelle zweier Θ-Λ-Bäume einen kombinierten Baum zu verwenden, der alle drei Werte berechnet. Dieser speichert dann in jedem Knoten k neben p_k, ECT_k, $\mathrm{p}\Lambda_k$ und $\mathrm{ECT}\Lambda_k$ auch die Werte $\mathrm{p}\Omega_k$ und $\mathrm{ECT}\Omega_k$, die bis zu eine Aktivität aus Ω miteinbeziehen und analog zu $\mathrm{ECT}\Lambda_k$ und $\mathrm{p}\Lambda_k$ initialisiert und aktualisiert werden.

- Der Zugriff auf $\mathrm{ECT}(\Theta \cup \{j\})$ (Zeile 34) kann, wie bereits in Tabelle 3.1 angegeben, in $\mathcal{O}(\log n)$ erfolgen.

- Der Wert $\mathrm{ECT}(\Theta \cup \{j\}, \{i \in T_{\text{enabled}} \setminus \Theta \mid \mathrm{lst}_i < \mathrm{ECT}(\Theta \cup \{j\})\})$ (Zeile 35) kann nicht mit einem Θ-Λ-Baum berechnet werden, weil der zweite Parameter dynamisch ist: Der Wert von $\mathrm{ECT}(\Theta \cup \{j\})$ kann zwischen den Durchläufen der `repeat`-Schleife beliebig schwanken. Deshalb muss die Berechnung über die Schleifenmethode erfolgen, die in (3.7) auf Seite 47 beschrieben ist. Hierbei können für jedes i die Bedingungen $i \in T_{\text{enabled}}$, $i \notin \Theta$ und $\mathrm{lst}_i < \mathrm{ECT}(\Theta \cup \{j\})$ geprüft werden, wenn die Schleife auf i stößt. Der Wert $\mathrm{ECT}(\Theta \cup \{j\})$ kann dabei einmalig zu Beginn berechnet und dann zwischengespeichert werden, sodass das Prüfen der Bedingungen in konstanter Zeit möglich ist. Insgesamt ergibt sich ein Zeitaufwand von $\mathcal{O}(n)$.

- Die Berechnung von $\mathrm{ECT}(\Theta, \Omega, \Lambda'(o))$ (Zeile 27) kann ebenfalls mit der Schleifenmethode erfolgen, wie in Abschnitt 3.2.1 gezeigt wurde. Damit dies in $\mathcal{O}(n)$ möglich ist, darf aber der Zugriff auf $\mathrm{ECT}(\Theta \cup \{o\})$ für die verschiedenen betrachteten o nur konstante Zeit benötigen, weil dieser Wert bei der angegebenen Methode für mehrere (potentiell $\mathcal{O}(n)$ viele) Aktivitäten benötigt wird. Dies kann erreicht werden, wenn zuvor ein Array berechnet wird, das diese Werte für alle Aktivitäten speichert. Listing 3.4 auf der nächsten Seite zeigt, wie dies mit linearem Aufwand möglich ist. Dabei wird wieder auf Proposition 3.3 auf Seite 43 zurückgegriffen, um die früheste Fertigstellungszeit einer Aktivität in Kombination mit einer Menge von Aktivitäten zu berechnen.

  Durch diese Vorausberechnung der Werte von $\mathrm{ECT}(\Theta \cup \{o\})$ kann im Anschluss $\mathrm{ECT}(\Theta, \Omega, \Lambda'(o))$ mit ebenfalls linearem Aufwand berechnet werden. □

Damit sind die Grundlagen gelegt, um die Gesamtlaufzeit zu betrachten:

Satz 3.7 (Laufzeit)**.** *Der in Listing 3.3 auf Seite 51 eingeführte Edgefinding-Algorithmus für optionale Aktivitäten hat einen Zeitaufwand von* $\mathcal{O}(n^2)$.

Beweis. Ausgehend von den vorigen Überlegungen bleibt noch zu zeigen, dass jede Zeile in der `repeat`-Schleife höchstens n-mal ausgeführt wird. Für die direkt enthaltenen Anweisungen ist dies unmittelbar klar, weil die `repeat`-Schleife genau über die n Aktivitäten iteriert. Es müssen also nur noch die inneren Schleifen betrachtet werden:

- Die Schleife in den Zeilen 14 bis 21 entfernt bei jedem Durchlauf eine Aktivität i aus der Menge Λ. Da jede Aktivität höchstens einmal zu Λ hinzugefügt werden kann (und zwar in Zeile 9 beim Entfernen aus Θ beziehungsweise Ω), kann der Rumpf der Schleife höchstens n-mal ausgeführt werden.

Listing 3.4: Berechnung von $\mathrm{ECT}(\Theta \cup \{o\})$ für alle $o \in T$

```
1  // Voraussetzung: L liegt bereits sortiert vor
2  // (schon wegen des Θ-Baums notwendig)
3  L := Liste aller Aktivitäten mit aufsteigender est_i
4  first := erste Aktivität in L
5  last := letzte Aktivität in L
6
7  ectBefore_first := −∞ // ectBefore_o = ECT({i ∈ Θ | i vor o in L})
8  ectAfter_last := −∞ // ectAfter_o = ECT({i ∈ Θ | i nach o in L})
9  pAfter_last := 0 // pAfter_o = p_{i∈Θ | i nach o in L}
10
11 // In den Schleifen wird Proposition 3.3 auf Seite 43 genutzt,
12 // um die Arrays ectAfter, ectBefore und ectWith zu füllen
13 for i ∈ L von hinten nach vorne (first auslassen) do
14   n := Aktivität vor i in L
15   if i ∈ Θ then
16     pAfter_n := pAfter_i + p_i
17     ectAfter_n := max{ectAfter_i, ect_i + pAfter_i}
18   else
19     pAfter_n := pAfter_i
20     ectAfter_n := ectAfter_i
21
22 for i ∈ L von vorne nach hinten (last auslassen) do
23   n := Aktivität nach i in L
24   if i ∈ Θ then
25     ectBefore_n := max{ect_i, ectBefore_i + p_i}
26   else
27     ectBefore_n := ectBefore_i
28   ectWith_n := max{ectAfter_n, ect_n + pAfter_n, ectBefore_n + p_n + pAfter_n}
29
30 ectWith_first := max{ect_first + pAfter_first, ectAfter_first}
31
32 // Nun gilt für alle o ∈ T: ectWith_o = ECT(Θ ∪ {o})
```

- Die Schleifen in den Zeilen 22 bis 25 und 26 bis 32 entfernen bei jedem Durchlauf eine Aktivität aus Ω. Nach der Initialisierung in Zeile 6 werden zu Ω keine Aktivitäten mehr hinzugefügt, weshalb auch diese beiden Schleifen höchstens n-mal ausgeführt werden können. □

3.2.4. Optimalität

Für eine effiziente Einschränkung der Domänen ist es wünschenswert, dass ein Filteralgorithmus so viele Regelanwendungen wie möglich findet, weil dadurch der durchsuchte Teil des Suchbaums kleiner wird und Lösungen schneller gefunden werden. Gleichzeitig ist es aber wichtig, dass ein Filteralgorithmus schnell arbeitet, da er durch Fixpunktbildung und Propagation nach jedem Suchschritt wiederholt aufgerufen werden kann.

Tabelle 3.2 zeigt zwei Extremformen von Filteralgorithmen: Der eine verfügt über eine möglichst geringe Laufzeit, der andere über möglichst große Einschränkungen. Beide Ansätze sind für die Praxis ungeeignet: Beim ersten muss der gesamte Suchbaum durchlaufen werden, was zu exponentieller Laufzeit führt und gegenüber der einfachen Tiefensuche ohne Constraint-Propagation keinen Vorteil bietet. Beim zweiten Ansatz steht die optimale Filterleistung der exponentiellen Laufzeit nach jedem einzelnen Suchschritt gegenüber, die ebenfalls inakzeptabel ist. Es muss also ein Kompromiss zwischen kurzer Lauf-

Tabelle 3.2: Unterschiedliche Ansätze für Filteralgorithmen am *SingleResource*-Constraint mit ihren Vor- und Nachteilen

	Schneller Ansatz	**Umfangreicher Ansatz**
Vorgehensweise	Wenn alle Domänen einelementig sind: Überlappungsfreiheit prüfen; sonst werden keine Einschränkungen vorgenommen	Alle Möglichkeiten durchprobieren und alle Werte aus den Domänen entfernen, die in keiner Lösung vorkommen
Einschränkungen	Verkleinert den Suchraum nicht, entscheidet nur, ob eine Lösung vorliegt	Findet alle Einschränkungen, für die eine Begründung existiert
Laufzeit	$\mathcal{O}(n \log n)$ (Sortieren nach est_i und Prüfen der Überlappungsfreiheit)	Vermutlich exponentiell (das Problem der exklusiven Ressourcen ist NP-vollständig)

zeit der einzelnen Aufrufe und optimaler Filterleistung gefunden werden. Die Beschränkung auf die in Abschnitt 2.2 vorgestellten Regeln geht bereits in diese Richtung.

Der in diesem Kapitel vorgestellte Algorithmus geht hier noch weiter, indem er in seltenen Fällen einige Anwendungen der Edgefinding-Regel übersieht:

Satz 3.8 (Optimalität)**.** *Der in Listing 3.3 auf Seite 51 eingeführte Edgefinding-Algorithmus für optionale Aktivitäten findet*

(a) alle Anwendungen der Überlast-Regel auf Aktivitäten aus T_{enabled}*,*
(b) alle Anwendungen der Überlast-Regel auf optionale Aktivitäten,
(c) alle Anwendungen der Edgefinding-Regel auf Aktivitäten aus T_{enabled}*,*
(d) viele Anwendungen der Edgefinding-Regel auf optionale Aktivitäten.

Beweis. Die Teilaussagen werden getrennt betrachtet:

(a) Wenn die Überlast-Regel 2.4 auf Seite 28 für Aktivitäten aus T_{enabled} anwendbar ist, muss es ein $j' \in T_{\text{enabled}}$ geben, sodass für die Menge $\Theta' := \left\{ i \in T_{\text{enabled}} \mid \text{lct}_i \leq \text{lct}_{j'} \right\}$ eine Überlast auftritt. Dieser Fall tritt ein, wenn $\text{ECT}(\Theta') > \text{lct}_{\Theta'}$ ist.
Zu dem Zeitpunkt, an dem in der `repeat`-Schleife j die erste in Θ' enthaltene Aktivität erreicht (dies kann j' oder ein j'' mit $\text{lct}_{j''} = \text{lct}_{j'}$ sein), gilt nach Lemma 3.4 auf Seite 52 $\Theta = \Theta'$. Damit wird die Bedingung $\text{ECT}(\Theta) > \text{lct}_j$ in Zeile 12 erfüllt und in der folgenden Zeile durch `fail` abgebrochen.

(b) Wenn für ein $o \in T_{\text{optional}}$ die Überlast-Regel für $T' := T_{\text{enabled}} \cup \{o\}$ anwendbar ist, muss eine Aktivität $j' \in T'$ existieren, sodass für die Menge $\Theta' := \left\{ i \in T' \mid \text{lct}_i \leq \text{lct}_{j'} \right\}$ eine Überlast auftritt und $\text{ECT}(\Theta') > \text{lct}_{\Theta'}$ ist. Sei j'' die Aktivität aus Θ', die als erste in Q einsortiert wurde. An dieser Stelle müssen zwei Fälle unterschieden werden:

Fall 1: $j'' = o$

In diesem Fall gilt zu dem Zeitpunkt, an dem j in der `repeat`-Schleife j'' erreicht, dass $\Theta = \Theta' \cap T_{\text{enabled}}$, woraus wegen $j = o$ folgt, dass $\Theta' = \Theta \cup \{j\}$. Mit $\text{lct}_{\Theta'} = \text{lct}_j$ folgt, dass die Bedingung $\text{ECT}(\Theta \cup \{j\}) > \text{lct}_j$ in Zeile 34 erfüllt ist und in den folgenden Zeilen mit $j = o$ die einzige beteiligte optionale Aktivität deaktiviert wird, was genau der von der Überlast-Regel ermöglichten Einschränkung entspricht.

Fall 2: $j'' \neq o$

In diesem Fall muss o nach j'' in Q einsortiert worden sein und ist nach Lemma 3.4 zu dem Zeitpunkt, zu dem in der `repeat`-Schleife $j = j''$ wird, noch in Ω enthalten. Damit wird die Bedingung $\text{ECT}(\Theta, \Omega) > \text{lct}_j$ in Zeile 22 erfüllt. Nachdem gegebenenfalls weitere $o' \in \Omega$, die $\text{ECT}(\Theta \cup \{o'\}) > \text{lct}_j$ erfüllen, ausgewählt und im Rumpf der `while`-Schleife deaktiviert und aus Ω entfernt wurden,

kommt auch o an die Reihe und wird in Zeile 24 deaktiviert. Damit wird auch dieser Fall vom Algorithmus erfasst.

(c) Wenn die Edgefinding-Regel 2.6 auf Seite 30 auf T_{enabled} anwendbar ist, müssen aufgrund von Proposition 2.7 auf Seite 30 zwei Aktivitäten $i', j' \in T_{\text{enabled}}$ existieren, für die $\text{lct}_{i'} > \text{lct}_{j'}$ und $\text{ECT}(\Theta' \cup \{i'\}) > \text{lct}_{\Theta'}$ gilt, wobei $\Theta' := \{i \in T_{\text{enabled}} \mid \text{lct}_i \leq \text{lct}_{j'}\}$ bezeichnet. Ebenfalls aufgrund von Proposition 2.7 kann ohne Beschränkung der Allgemeinheit angenommen werden, dass die Edgefinding-Regel für kein j_2' mit $\text{lct}_{j_2'} > \text{lct}_{j'}$ für j_2' und i anwendbar ist, weil sich daraus eine mindestens genausogroße Einschränkung ergeben würde – in diesem Fall könnte die folgende Argumentation auf j_2' angewendet werden.
Sei j'' wieder die Aktivität aus Θ', die als erste in Q einsortiert wurde. Zu dem Zeitpunkt, an dem in der `repeat`-Schleife j die Aktivität j'' erreicht, gilt damit nach Lemma 3.4 $\Theta = \Theta'$. Da darüberhinaus keine Aktivität j_2' mit $\text{lct}_{j_2'} > \text{lct}_{j'}$ eine Anwendung der Edgefinding-Regel ermöglicht, kann bei keinem der vorigen Durchläufe der `repeat`-Schleife in Zeile 14 i' aus Λ ausgewählt worden sein. Damit muss zu dem aktuell betrachteten Zeitpunkt $i' \in \Lambda$ gelten. Deshalb wird in Zeile 14 (gegebenenfalls nach mehreren Durchläufen der dort beginnenden `while`-Schleife) $i = i'$ aus Λ ausgewählt; in den folgenden Zeilen wird entweder die durch die Edgefinding-Regel ermöglichte Einschränkung $\text{est}_i := \max\{est_i, \text{ECT}(\Theta)\}$ umgesetzt oder mit $\text{status}_i :=$ disabled eine Inkonsistenz ausgelöst, falls die Domäne der Variable $start_i$ durch diese Einschränkung leer würde (was gleichfalls eine Inkonsistenz verursachen würde).
So werden alle Anwendungen der Edgefinding-Regel auf Aktivitäten aus T_{enabled} gefunden.

(d) Wenn für ein $o' \in T_{\text{optional}}$ die Edgefinding-Regel auf $T' := T_{\text{enabled}} \cup \{o'\}$ anwendbar ist, muss es $i', j' \in T'$ geben, für die $\text{lct}_{j'} < \text{lct}_{i'}$ und $\text{ECT}(\Theta' \cup \{i'\}) < \text{lct}_{\Theta'}$ gilt, wobei $\Theta' := \{i \in T' \mid \text{lct}_i \leq \text{lct}_{j'}\}$ ist.
Sei j'' wieder die erste Aktivität aus Θ' in Q. Nun können die folgenden vier Fälle auftreten:

Fall 1: $o' \notin \Theta' \cup \{i'\}$

In diesem Fall ist die Edgefinding-Regel bei gleicher Wahl von i' und j' auch auf T_{enabled} anwendbar. Damit gilt die in (c) bewiesene Teilaussage.

Fall 2: $o' = i'$

Dieser Fall wird zusammen mit dem nicht-optionalen Fall in den Zeilen 14 bis 21 behandelt, weil in Λ auch optionale Aktivitäten enthalten sind. Es wird $\text{est}_{o'}$ angehoben oder o' deaktiviert, falls in der Domäne von $start_{o'}$ keine Werte übrigblieben. Dies stellt jeweils die optimale Einschränkung für diesen Fall dar.

Fall 3: $o' = j''$

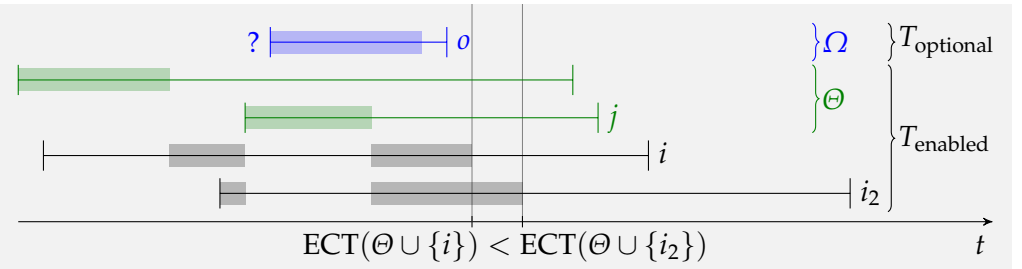
In diesem Fall darf $\text{est}_{i'}$ nicht angepasst werden, weil mit o' eine optionale Aktivität beteiligt ist und nicht-optionale Aktivitäten nicht von optionalen beeinflusst werden dürfen (vgl. Abschnitt 2.4.2). Falls die resultierende Einschränkung zu einer Inkonsistenz führte, ist es aber möglich, o' zu deaktivieren. Eine solche Inkonsistenz tritt genau dann auf, wenn $est_{i'}$ auf einen Wert größer als $\text{lst}_{i'}$ angehoben werden könnte. Hierzu muss $\text{ECT}(\Theta') > \text{lst}_{i'}$ gelten.
Es gilt $\Theta' = \Theta \cup \{j''\}$, da j'' die einzige optionale Aktivität in Θ' ist. Die Aktivität i' ist wegen $\text{lct}_{i'} > \text{lct}_{j'}$ nicht in Θ, wohl aber in T_{enabled} enthalten, weil j'' die einzige optionale unter den beteiligten Aktivitäten ist. Damit kommt bei der Berechnung von $\text{ECT}(\Theta \cup \{j\}, \{i \in T_{\text{enabled}} \setminus \Theta \mid \text{lst}_i < \text{ECT}(\Theta \cup \{j\})\})$ in Zeile 35 mindestens i' für i in Betracht. Folglich muss dieser Wert mindestens $\text{ECT}(\Theta' \cup \{i'\})$ betragen, was wegen der Anwendbarkeit der Edgefinding-Regel größer als $\text{lct}_{\Theta'} = \text{lct}_j$ sein muss. Somit wird die Bedingung des `if`-Konstrukts in Zeile 35 erfüllt und $j = o'$ wird wie erforderlich deaktiviert.

Fall 4: $o' \in \Theta' \setminus \{j''\}$

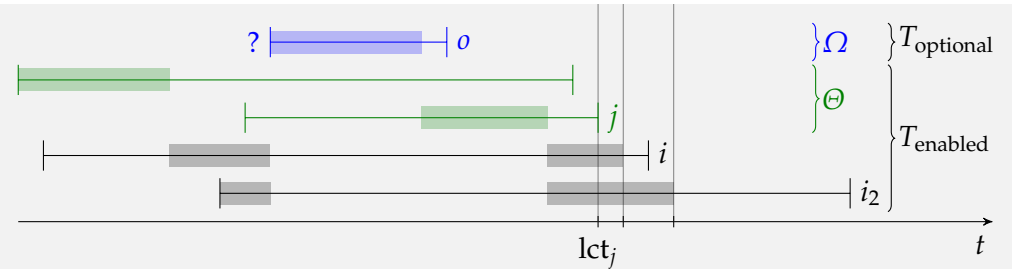
Wie im vorigen Fall ist die Deaktivierung von o', falls $\text{ECT}(\Theta') > \text{lst}_{i'}$ ist, die einzig mögliche Einschränkung.
Desweiteren muss im betrachteten Durchlauf der `repeat`-Schleife $o' \in \Omega$, $j \in T_{\text{enabled}}$ und $\Theta' = \Theta \cup \{o\}$ gelten. Damit wird die `while`-Schleife in den Zeilen 26 bis 32 erreicht, die $\Omega \neq \emptyset$ und $\text{ECT}(\Theta, \Omega, \Lambda'(o)) > \text{lct}_j$ als Bedingung hat. Ersteres ist im betrachteten Fall gegeben, weil $o' \in \Omega$. Letzteres gilt aber nach der Definition von $\Lambda'(o)$ in (3.11) auf Seite 50 nur, falls zusätzlich folgende Bedingung erfüllt ist:

$$\nexists i_2 \in T_{\text{enabled}} \setminus \Theta : \Big(\text{lst}_{i_2} \geq \text{ECT}(\Theta \cup \{o\}) \\ \wedge \text{ECT}(\Theta \cup \{i_2\}) > \text{ECT}(\Theta \cup \{i\}) \Big) \quad (3.12)$$

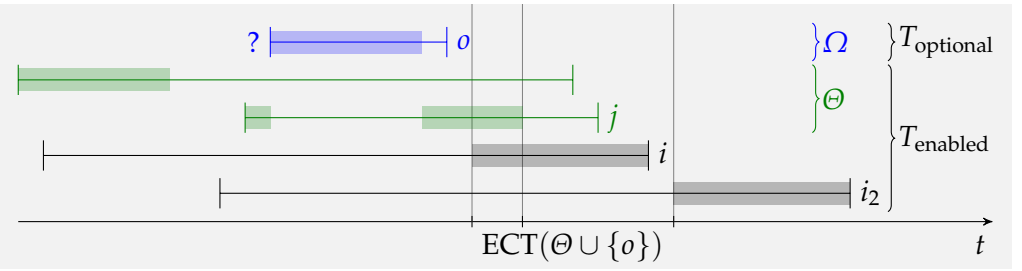
Wenn dies gilt, wird o' (gegebenenfalls nach einigen Iterationen) als o ausgewählt und im Schleifenrumpf deaktiviert.
Diese Bedingung ist häufig, aber nicht immer erfüllt; in Abbildung 3.2 ist sie beispielsweise verletzt. Umgekehrt wird sie von allen Standard-Instanzen der Probleme erfüllt, die im folgenden Kapitel betrachtet werden. Für alle diese Probleme wird damit immer die optimale Einschränkung gefunden. □



(a) Bei der Berechnung von $ect\Lambda$ wird i_2 gegenüber i bevorzugt, weil sich so der größere Wert ergibt.



(b) $\mathrm{ECT}(\Theta \cup \{o, i\})$ und $\mathrm{ECT}(\Theta \cup \{o, i_2\})$ sind beide größer als lct_j und ermöglichen damit die Anwendung der Edgefinding-Regel.



(c) Da lst_i im Gegensatz zu lst_{i_2} kleiner als $\mathrm{ECT}(\Theta \cup \{o\})$ ist, ermöglicht nur i das Deaktivieren von o.

Abbildung 3.2: Bei diesem Beispiel wird eine Einschränkung übersehen, weil i_2 anstelle von i verwendet wird, obwohl nur i eine Einschränkung ermöglicht.

Um auch im letzten Fall alle Anwendungen der Edgefinding-Regel zu finden, müsste $\Lambda'(o)$ derart umdefiniert werden, dass die zusätzliche Bedingung aus (3.12) wegfällt, was abweichend von (3.11) auf Seite 50 folgende Gleichung ergibt:

$$\widehat{\Lambda'}(o) := \{i \in T_{\text{enabled}} \setminus \Theta \mid \text{lst}_i < \text{ECT}(\Theta \cup \{o\})\}$$

Damit fällt aber die Möglichkeit weg, $\text{ECT}(\Theta, \Omega, \widehat{\Lambda'}(o))$ mit der Schleifentechnik aus Abschnitt 3.1.2 zu berechnen, weil abhängig von dem gerade betrachteten $o \in \Omega$ der Wert ect_Λ für unterschiedliche i bekannt sein müsste.

Die Versuche des Autors, $\text{ECT}(\Theta, \Omega, \widehat{\Lambda'}(o))$ mit linearem oder amortisiert linearem Aufwand zu berechnen, haben trotz intensiver Bemühungen nicht zum Erfolg geführt. Da o und i jeweils aus $\mathcal{O}(n)$ vielen Aktivitäten ausgewählt werden müssen und sich deren Wahl gegenseitig beeinflusst, wäre es notwendig, das Verhältnis der beiden Aktivitäten geschickt auszunutzen, um nicht alle Kombinationen durchprobieren zu müssen. Die am Ende von Abschnitt 3.2.1 vorgestellte Variante der Schleifenmethode ist das beste Vorgehen, das auf dieser Grundlage erreicht werden konnte; es findet aber nicht immer die beste Kombination von o und i.

Eine optimale Implementierung würde an dieser Stelle also eine höhere Zeitkomplexität mit sich bringen und damit näher an das zweite Extrem aus Tabelle 3.2 auf Seite 59 heranrücken. Ob die gewählte Herangehensweise eine sinnvolle Abwägung zwischen Stärke der Einschränkung und kurzer Laufzeit darstellt, wird im folgenden Kapitel untersucht.

4. Anwendungen und Laufzeitmessungen

Zeit ist Leben.

Michael Ende, »Momo«

Der in Listing 3.3 auf Seite 51 eingeführte Edgefinding-Algorithmus wurde zusammen mit Algorithmen für die übrigen in Abschnitt 2.2 vorgestellten Filterregeln im Rahmen dieser Diplomarbeit in der Java-Bibliothek `firstcs` [HMS+03] implementiert. Da diese Bibliothek die Auswahl zwischen unterschiedlichen Edgefinding-Algorithmen erlaubt, können diese miteinander verglichen werden. In diesem Kapitel werden die verschiedenen Verfahren auf Laufzeit und Stärke der Suchraumeinschränkung hin untersucht.

Dabei werden die folgenden Edgefinding-Algorithmen betrachtet:

Neuer Algorithmus: Der im vorigen Kapitel eingeführte Algorithmus (Listing 3.3). Er findet fast alle Einschränkungen und benötigt dafür $\mathcal{O}(n^2)$ Zeit.

Vereinfachter Algorithmus: Dieser Algorithmus arbeitet ähnlich wie der vorherige, lässt aber die Schleife in den Zeilen 26 bis 32 aus. Damit findet er zwar unter Umständen weniger Einschränkungen, spart aber mit der Berechnung von $\mathrm{ECT}(\Theta, \Omega, \Lambda'(o))$ die aufwendigste Anweisung des Algorithmus ein. Die Komplexität ist weiterhin $\mathcal{O}(n^2)$, weil in Zeile 35 eine weitere $\mathcal{O}(n)$-Anweisung in der `repeat`-Schleife enthalten ist.

Nur-aktivierte-Algorithmus: Der Algorithmus von Vilím, Barták und Čepek [VBČ05]. Er lässt Aktivitäten aus T_{optional} unberücksichtigt, bietet aber mit $\mathcal{O}(n \log n)$ eine bessere Zeitkomplexität.

Iterativer Algorithmus: Die Erweiterung des vorigen Algorithmus mit der Technik aus Listing 2.5 auf Seite 38. Er findet als einziger der getesteten Algorithmen alle Anwendungen der Edgefinding-Regel, hat aber mit $\mathcal{O}(n^2 \log n)$ die größte asymptotische Zeitkomplexität.

Die Auswahl der Algorithmen ermöglicht die Untersuchung der folgenden Fragestellungen:

- Ermöglicht das Einbeziehen optionaler Aktivitäten in das Edgefinding überhaupt einen Zeitgewinn?

  Dieser Frage kann durch den Vergleich mit dem Algorithmus nachgegangen werden, der nur aktivierte Aktivitäten berücksichtigt.

- Wie viel Aufwand lohnt sich bei der Suche nach Einschränkungen für optionale Aktivitäten?

  Die übrigen drei Algorithmen verfolgen hier unterschiedliche Ansätze, die jeweils andere Kompromisse zwischen Anzahl der gefundenen Einschränkungen und dem dafür benötigten Aufwand bedeuten. In dieser Hinsicht ist auch interessant, ob sich der neue und der vereinfachte Algorithmus trotz ihrer komplexeren Datenstrukturen, die mit zusätzlichem Verwaltungsaufwand und damit höheren konstanten Faktoren einhergehen, in praktischen Anwendungen vom iterativen Algorithmus mit seinem höheren asymptotischen Aufwand absetzen können.

Für die Suche wurde das in Abschnitt 2.4.1 beschriebene Verfahren eingesetzt. Zur Fixpunktbildung wurde eine Variante des Verfahrens aus Listing 2.4 auf Seite 33 verwendet, bei der auf die Überlasterkennung verzichtet wird. Da alle betrachteten Edgefinding-Algorithmen auch die Überlast-Regel verwenden, werden so aussagekräftigere Messergebnisse erzielt.

Der prinzipiellen Ungenauigkeit bei der Zeitmessung mit Java-Programmen wurde dadurch begegnet, dass jede Messung fünfmal in Folge durchgeführt und anschließend daraus der Mittelwert gebildet wurde. Dabei wurde der schlechteste Wert nicht miteinbezogen, um den Einfluss der Garbage-Collection auf das Messergebnis zu reduzieren. Trotz dieser Herangehensweise sind die Laufzeiten nicht exakt reproduzierbar und schwanken zwischen mehreren Messungen, sodass trotz der angegebenen Genauigkeit von 1ms erst ab etwa 100ms Differenz von einer signifikanten Abweichung ausgegangen werden sollte. Hinzu kommt, dass die Genauigkeit der verwendeten Bibliotheksfunktion `System.currentTimeMillis()` begrenzt ist: Obwohl Millisekunden zurückgegeben werden, können wiederholte Messungen desselben Programmabschnitts entweder 15, 16 oder 31ms ergeben, ohne dass jemals Werte dazwischen erreicht werden. Dies könnte auf Prozess- und Thread-Wechsel zurückzuführen sein.

Im Folgenden werden unterschiedliche Probleme vorgestellt, zu deren Lösung exklusive Ressourcen verwendet werden, sodass dabei die unterschiedlichen Edgefinding-Algorithmen miteinander verglichen werden können.

4.1. Modifizierte Instanzen des Job-Shop-Problems

Das Job-Shop-Problem kommt aus der industriellen Produktionsplanung. Verschiedene Werkstücke (englisch *jobs*) müssen in einer bestimmten Reihenfolge mehrere Maschinen durchlaufen, an denen sie bearbeitet werden. In der mathematischen Modellierung wird eine Menge von n Werkstücken angegeben, die jeweils m Arbeitsschritte (englisch *tasks*) mit individueller Dauer durchlaufen. Daneben gibt es ebenfalls m Maschinen. Jeder Arbeitsschritt benötigt eine dieser Maschinen, und die m Arbeitsschritte jedes Werkstücks verteilen sich jeweils auf alle m Maschinen. Zu einer Instanz des Job-Shop-Problems wird dann die frühestmögliche Zeit gesucht, zu der alle Werkstücke fertig sein können.

Das Job-Shop-Problem ist NP-vollständig [GJ79, Problem SS18 auf Seite 242], weshalb die Constraint-Programmierung sich als ein Lösungsansatz anbietet, der wegen seiner Schnelligkeit seit einiger Zeit viel Beachtung findet [BPN95; Col96].

Da die Maschinen immer nur ein Werkstück gleichzeitig bearbeiten können und dabei nicht unterbrochen werden dürfen, können sie mit einem *SingleResource*-Constraint pro Maschine modelliert werden. Daneben werden bei jedem Werkstück *Before*-Constraints verwendet, um die richtige Reihenfolge der einzelnen Aktivitäten sicherzustellen.

Wird das Maximum der Fertigstellungszeiten der Aktivitäten mit zusätzlichen Constraints an eine Variable x gebunden, muss deren Wert minimiert werden, um eine möglichst gute Lösung zu erhalten. Das Tripel $(C, x, \min)$ ist damit ein Constraint-Optimierungs-Problem (vgl. Definition 1.13 auf Seite 17), wobei C die Menge der verwendeten Constraints bezeichnet.

Das Job-Shop-Problem bietet sich für Laufzeitmessungen an, weil es eine große Anzahl von veröffentlichten Benchmark-Instanzen gibt [FT63; Law84; ABZ88; SWV92; YN92].

Beispiel. In Tabelle 4.1 auf der nächsten Seite wird die Instanz fto6 von Fisher und Thompson [FT63] gezeigt, die jeweils 6 Werkstücke und Maschinen umfasst. Um diese Instanz mit Constraints zu modellieren, muss für alle Werkstücke $j \in \{1, \ldots, 6\}$ und Arbeitsschritte $t_j \in \{1, \ldots, 6\}$ jeweils eine Aktivität i_{j,t_j} angelegt werden. Diesen Aktivitäten wird jeweils die in der Tabelle angegebene Dauer zugewiesen. Da die Zeitpunkte der Bearbeitung noch unbekannt sind, wird den Start- und Fertigstellungsvariablen der Aktivitäten das Intervall $[0, end]$ als Domäne zugewiesen, wobei *end* eine geeignete obere Schranke ist. Hierzu kann beispielsweise die Summe aller Dauern der einzelnen Aktivitäten verwendet werden.

Tabelle 4.1: Die Instanz fto6 [FT63] des Job-Shop-Problems mit 6 Werkstücken und 6 Maschinen. Für jedes Werkstück j_i ist angegeben, in welcher Reihenfolge wieviel Zeit auf den einzelnen Maschinen benötigt wird.

Werkstück	**Aufg. 1**		**Aufg. 2**		**Aufg. 3**		**Aufg. 4**		**Aufg. 5**		**Aufg. 6**	
j_1	M_2:	1	M_0:	3	M_1:	6	M_3:	7	M_5:	3	M_4:	6
j_2	M_1:	8	M_2:	5	M_4:	10	M_5:	10	M_0:	10	M_3:	4
j_3	M_2:	5	M_3:	4	M_5:	8	M_0:	9	M_1:	1	M_4:	7
j_4	M_1:	5	M_0:	5	M_2:	5	M_3:	3	M_4:	8	M_5:	9
j_5	M_2:	9	M_1:	3	M_4:	5	M_5:	4	M_0:	3	M_3:	1
j_6	M_1:	3	M_3:	3	M_5:	9	M_0:	10	M_4:	4	M_2:	1

Über diesen Aktivitäten müssen dann *Before*-Constraints abgesetzt werden, die die richtige Reihenfolge der Arbeitsschritte an einem Werkstück sicherstellen:

$$\mathit{Before}(i_{j,t}, i_{j,t+1}) \qquad \forall j \in \{1,\ldots,6\} \forall t \in \{1,\ldots,5\}$$

Außerdem muss mit *SingleResource*-Constraints sichergestellt werden, dass die unterschiedlichen Arbeitsschritte an einer Maschine überlappungsfrei sind. Für das vorliegende Beispiel leisten das die folgenden Constraints:

$$\begin{aligned}
\text{Für } M_0: &\quad \mathit{SingleResource}(i_{1,2}, i_{2,5}, i_{3,4}, i_{4,2}, i_{5,5}, i_{6,4}) \\
\text{Für } M_1: &\quad \mathit{SingleResource}(i_{1,3}, i_{2,1}, i_{3,5}, i_{4,1}, i_{5,2}, i_{6,1}) \\
\text{Für } M_2: &\quad \mathit{SingleResource}(i_{1,1}, i_{2,2}, i_{3,1}, i_{4,3}, i_{5,1}, i_{6,6}) \\
\text{Für } M_3: &\quad \mathit{SingleResource}(i_{1,4}, i_{2,6}, i_{3,2}, i_{4,4}, i_{5,6}, i_{6,2}) \\
\text{Für } M_4: &\quad \mathit{SingleResource}(i_{1,6}, i_{2,3}, i_{3,6}, i_{4,5}, i_{5,3}, i_{6,5}) \\
\text{Für } M_5: &\quad \mathit{SingleResource}(i_{1,5}, i_{2,4}, i_{3,3}, i_{4,6}, i_{5,4}, i_{6,3})
\end{aligned}$$

Abbildung 4.1 zeigt eine optimale Lösung dieser Instanz.

Für die Zeitmessungen wurden die von Vilím, Barták und Čepek [VBČ05, Seite 423] eingeführten *-alt* Varianten der publizierten Job-Shop-Instanzen verwendet. In dieser Variante kann bei jedem Werkstück entweder der 5. oder der 6. Arbeitsschritt weggelassen werden. In der Modellierung wirkt sich das dahingehend aus, dass für diese Arbeitsschritte optionale Aktivitäten verwendet werden, deren *status*-Variablen mit einem *NotEqual*-Constraint verknüpft werden. So wird sichergestellt, dass genau eine der beiden optionalen Aktivitäten eines Werkstücks in einer Lösung enthalten ist. Auf diese Weise sollen Alternativen im Fertigungsprozess modelliert werden.

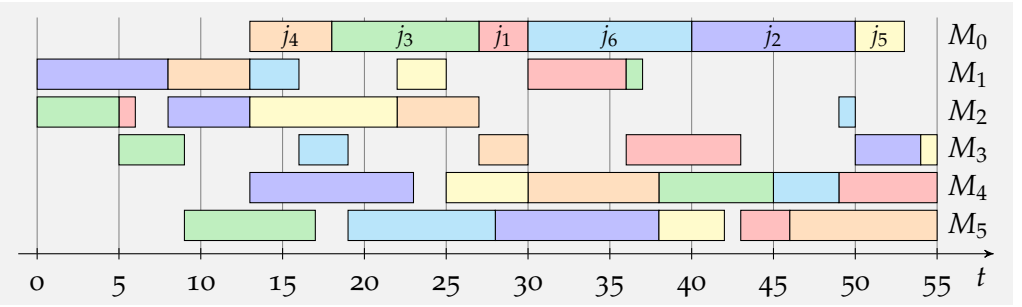


Abbildung 4.1: Eine optimale Lösung für die Job-Shop-Instanz fto6. Jede Zeile entspricht einer Maschine, jedes Rechteck einem Arbeitsschritt und jede Farbe einem Werkstück.

In Tabelle 4.2 auf der nächsten Seite sind Laufzeit und Anzahl der Backtracking-Schritte der zu Beginn des Kapitels vorgestellten Algorithmen für einige Job-Shop-Instanzen angegeben. Die Auswahl der Instanzen erfolgte danach, dass Lösungen in weniger als zehn Minuten gefunden werden können. Gemessen wurde jeweils die Zeit, um bei bekannter optimaler Dauer eine Lösung zu finden und anschließend deren Optimalität dadurch zu beweisen, dass die Instanz bei um 1 verringerter Gesamtdauer unlösbar wird.

Zunächst soll die Anzahl der Backtracking-Schritte betrachtet werden. Dabei ist auffällig, dass zwar wie zu erwarten der nur-aktivierte-Algorithmus am schlechtesten abschneidet, die drei anderen Algorithmen jedoch bei allen Instanzen den gleichen Wert aufweisen. Dies ist insofern überraschend, als die Anzahl der Backtracking-Schritte ein Maß für die Anzahl der gefundenen Einschränkungen ist und nur der iterative Algorithmus alle Einschränkungen finden kann. Bei den betrachteten Instanzen werden aber weder vom neuen noch vom vereinfachten Algorithmus Einschränkungen übersehen.[1]

Bei der Betrachtung der Laufzeiten ist zu erkennen, dass unter den drei Algorithmen, die optionale Aktivitäten einbeziehen, der vereinfachte meistens am besten abschneidet und der neue in vielen, aber nicht allen Fällen schneller als der iterative ist. Wie zu erwarten ist damit der vereinfachte Algorithmus in allen Fällen schneller als der neue – er benötigt schließlich weniger Anweisungen und findet in allen Fällen genausoviele Einschränkungen.

Der Vergleich mit dem iterativen Algorithmus zeigt, dass der vereinfach-

[1] Hierfür werden vom neuen Algorithmus bis zu 0,006% und vom vereinfachten bis zu 0,01% mehr Fixpunktiterationen benötigt. Diese Abweichungen sind aber so gering, dass sie bei der Gesamtlaufzeit kaum ins Gewicht fallen und im weiteren vernachlässigt werden. Die genauen Werte finden sich in Tabelle A.1 auf Seite 79.

Tabelle 4.2: Laufzeit und Anzahl der Backtracking-schritte bei verschiedenen Job-Shop-Problemen. Am Ende der Tabelle sind zum Vergleich drei unmodifizierte Instanzen ohne optionale Aktivitäten angegeben. Alle Instanzen haben 10 Werkstücke und 10 Maschinen, was bei den *-alt* Instanzen zu 20 optionalen Aktivitäten führt.

	neu		**vereinfacht**		**nur-aktivierte**		**iterativ**	
Instanz	**Zeit/ms**	**#bt**	**Zeit/ms**	**#bt**	**Zeit/ms**	**#bt**	**Zeit/ms**	**#bt**
abz5-alt	4843	5859	4484	5859	4515	6441	4964	5859
orb01-alt	55747	56344	53662	56344	49361	56964	57013	56344
orb02-alt	7800	7265	7394	7265	9590	10610	7609	7265
orb07-alt	81856	99471	79063	99471	78309	104201	79786	99471
orb10-alt	136	85	125	85	121	85	132	85
la16-alt	7269	8294	6886	8294	6593	8841	7241	8294
la17-alt	46	9	31	9	35	11	31	9
la18-alt	26780	26846	25147	26846	24877	29039	25897	26846
la19-alt	2566	2022	2406	2022	4609	4632	2574	2022
la20-alt	62	53	63	53	55	53	62	53
abz5	5863	5107	5863	5107	5587	5107	5570	5107
orb01	29612	21569	29706	21569	28198	21569	28336	21569
orb02	10144	7937	10187	7937	9644	7937	9687	7937

te schon bei der kleinen Instanzgröße von nur 10 Aktivitäten pro exklusiver Ressource von seiner geringeren Komplexität profitiert, während dies für den neuen nur in einigen Fällen zutrifft. Dabei ist auch zu berücksichtigen, dass der iterative Algorithmus nur bei vielen optionalen Aktivitäten wirklich langsamer wird, weil ein n in seinem asymptotischen $\mathcal{O}(n^2 \log n)$-Aufwand sich nur auf optionale Aktivitäten bezieht.[2] Die einzigen Fälle, in denen der vereinfachte signifikant schlechter als der iterative ist, sind dementsprechend die drei unmodifizierten Instanzen am Fuß der Tabelle, die überhaupt keine optionalen Aktivitäten aufweisen.

Interessant ist auch der Vergleich zwischen dem vereinfachten Algorithmus und dem, der nur aktivierte Aktivitäten einbezieht: Letzterer ist erwartungsgemäß bei den drei Instanzen ohne optionale Aktivitäten schneller, weil hier der zusätzliche Aufwand der übrigen Algorithmen keine zusätzlichen Einschränkungen finden kann. Bei den übrigen getesteten Instanzen ist das Bild

[2] Dieses n ist eigentlich ein $n + 1$, sodass auch bei 0 optionalen Aktivitäten noch ein Aufwand von $\mathcal{O}(n \log n)$ anfällt.

uneinheitlich: Bei sechs von zehn Instanzen ist der nur-aktivierte-Algorithmus schneller, die stärksten Abweichungen gehen aber zugunsten des vereinfachten Algorithmus: Bei orb02-alt benötigt der nur-aktivierte knapp 30%, bei la19-alt sogar über 90% mehr Zeit. Diese großen Abweichungen kommen in den Fällen zustande, in denen auch die Anzahl der Backtracking-schritte weit auseinanderliegt.

Da alle der hier getesteten Job-Shop-Instanzen nur zehn Aktivitäten pro Ressource aufweisen, von denen nur 20% optional sind, liegt die Frage nach dem Verhalten der Algorithmen bei Problemen mit einem höheren Anteil von optionalen Aktivitäten nahe. Ein solches wird im folgenden Abschnitt untersucht und verwendet zusätzlich auch mehr optionale Aktivitäten pro Ressource.

4.2. Random-Placement-Problem

Beim Random-Placement-Problem geht es darum, ob und wie eine Menge von Rechtecken überlappungsfrei auf einer rechteckigen Grundfläche platziert werden kann [BMR04, Seite 244]. Dabei können noch zusätzliche Randbedingungen für die Platzierung der einzelnen Rechtecke in Form von Intervallen zulässiger x- und y-Koordinaten angegeben werden.[3] Der Name des Problems rührt daher, dass die Autoren eine größere Anzahl von Testfällen für eines ihrer Programme benötigten und deshalb großen Wert auf die automatische Erzeugbarkeit zufälliger Instanzen gelegt haben. Im Internet ist eine Sammlung vorgenerierter Instanzen verfügbar [RV]. In Abbildung 4.2 auf der nächsten Seite ist eine aus dieser Sammlung entnommene Beispielinstanz dargestellt.

Das Problem ist eng mit der Zeitplanung verwandt: Wenn man die x-Achse als Zeitachse und unterschiedliche y-Werte als verschiedene Räume sowie die zu platzierenden Rechtecke als Veranstaltungen interpretiert, entspricht eine Lösung gerade einer zulässigen Verteilung von Veranstaltungen auf die Räume und Zeitfenster.

Da die Beispielinstanzen alle eine feste Höhe von 1 aufweisen, entspricht dieses Problem gerade einer alternativen Ressource, wie sie in der Einführung zu Abschnitt 2.2 erwähnt wurde: Jeder y-Wert entspricht einer Einzelressource, und jedes Rechteck kann zwischen den Ressourcen beliebig verschoben werden. Wenn die y-Position eines Rechtecks eingeschränkt ist, kommen nur die entsprechenden Ressourcen in Betracht.

[3] In der Regel wird neben den Grenzen der Grundfläche nur eine untere Schranke für die y-Koordinate angegeben.

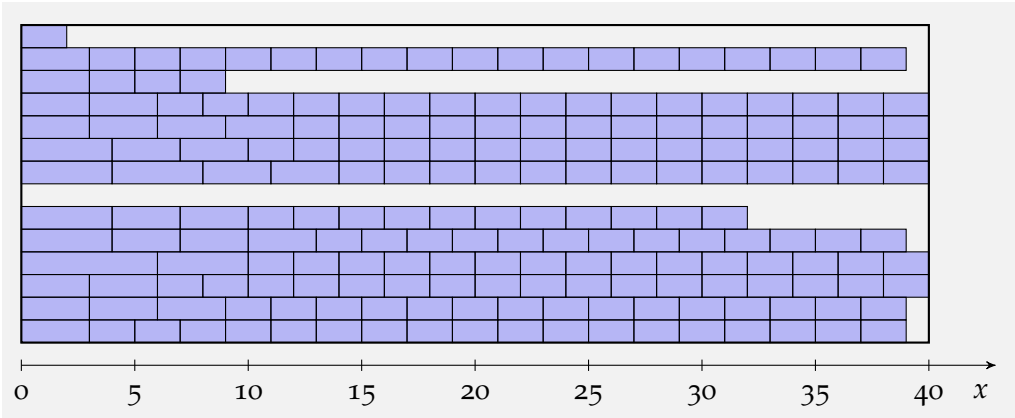


Abbildung 4.2: Die Instanz 80gen01 (die erste Beispielinstanz mit 80% Auslastung) des Random-Placement-Problems mit einer Lösung. Die Lücken kommen durch Einschränkungen der y-Koordinaten zustande.

Da das alternative-Ressourcen-Constraint mithilfe mehrerer *SingleResource*-Constraints implementiert werden kann, deren Aktivitäten zunächst alle optional sind [WS05], eignet sich das vorgestellte Problem hervorragend für Zeitmessungen bezüglich der unterschiedlichen Edgefinding-Algorithmen. In Abbildung 4.3 sind die Ergebnisse dieser Messungen dargestellt. Über die x-Achse sind dabei die verschiedenen Beispielinstanzen verteilt, die nach Auslastung gruppiert sind. Eine Auslastung von 80% bedeutet in diesem Zusammenhang, dass die zu platzierenden Rechtecke 80% der Gesamtfläche ausfüllen. Von den jeweils 50 unter [RV] verfügbaren Beispielinstanzen mit 80%, 85%, 90%, 95% und 100% Auslastung wurden wieder diejenigen ausgewählt, die in weniger als zehn Minuten lösbar sind.[4]

Beim Betrachten der Ergebnisse fällt auf, dass der neue Algorithmus über alle Instanzen hinweg die mit Abstand kürzeste Laufzeit aufweist. Danach folgen der Algorithmus, der nur aktivierte Aktivitäten berücksichtigt sowie der vereinfachte und zuletzt der iterative Algorithmus. Dabei werden der neue und der iterative mit steigender Auslastung besser, während die anderen beiden Algorithmen bei diesen Instanzen mehr Zeit benötigen. Das ist vermutlich darauf zurückzuführen, dass das Problem mit höherer Auslastung schwieriger wird (es gibt weniger mögliche Lösungen), die beiden erstgenannten Algorithmen dies jedoch durch bessere Einschränkungen ausgleichen können.

[4] Dies umfasst alle Instanzen mit 80% und 85% Auslastung, 44 der 50 Instanzen mit 90% Auslastung, 46 der 50 Instanzen mit 95% Auslastung und 29 der 50 Instanzen mit 100% Auslastung.

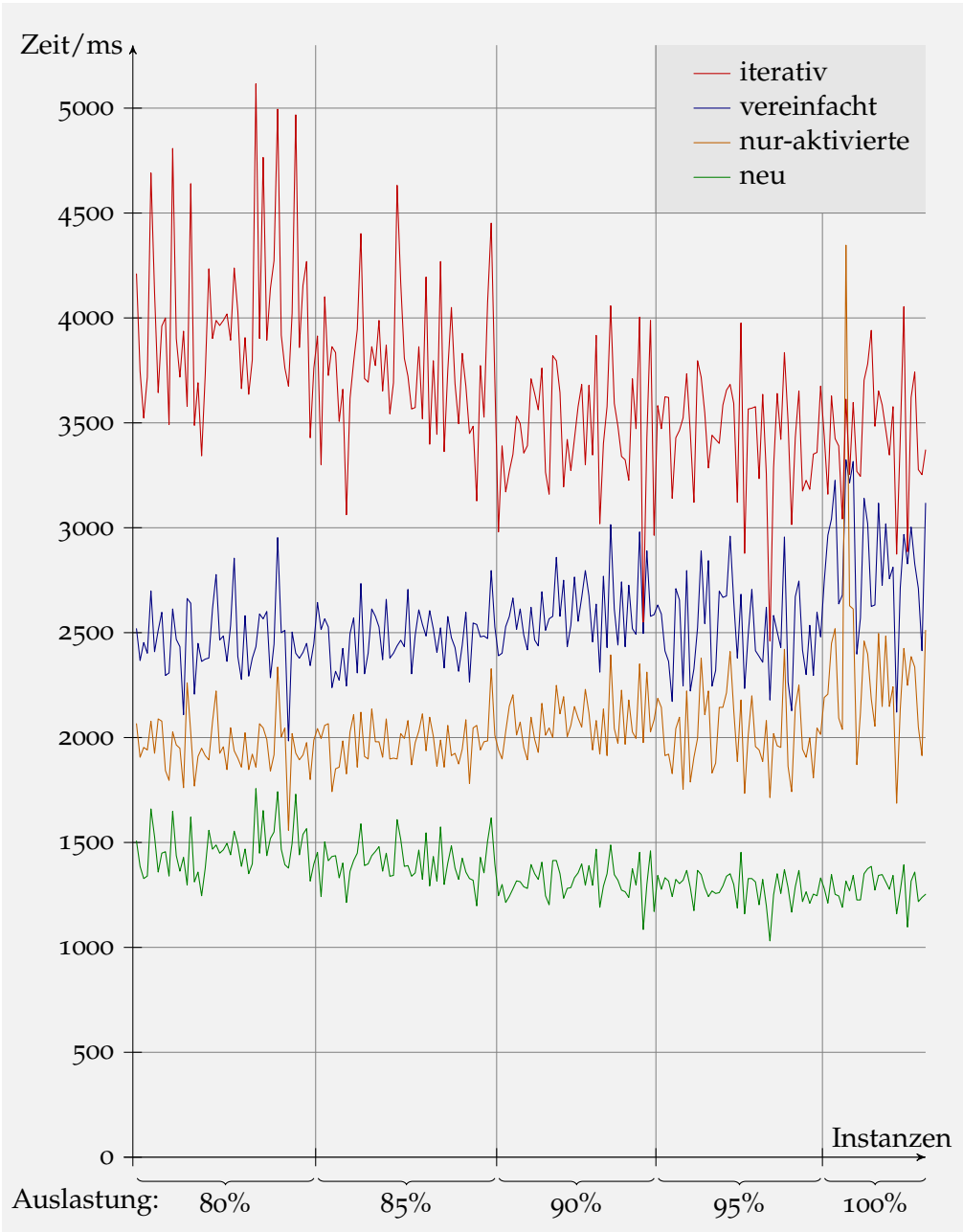


Abbildung 4.3: Laufzeit der Lösungssuche für Standardinstanzen des Random-Placement-Problems mit den verschiedenen Edgefinding-Algorithmen

An dieser Stelle ist auch bemerkenswert, dass in fast allen Fällen kein Backtracking zum Finden einer Lösung notwendig ist – die einzige Ausnahme bildet der nur-aktivierte-Algorithmus bei einer der Instanzen mit 100% Auslastung. Dies führt in Abbildung 4.3 dazu, dass der zugehörige Messwert sogar noch über dem des iterativen liegt. Trotz der gleichen Anzahl von Backtracking-Schritten hat die Stärke der Einschränkung einen Einfluss auf die Gesamtlaufzeit, weil beim Traversieren des Suchbaums nur dann propagiert werden muss, wenn zuvor mehrere Möglichkeiten bestanden. Wenn also bei einem Suchschritt alle bis auf eine der Möglichkeiten durch vorheriges Filtern nicht mehr in Frage kommen, kann die anschließende Propagation übersprungen werden, weil sich gegenüber dem letzten Propagieren nichts geändert hat und der dabei gefundene Fixpunkt nach wie vor Bestand hat.

In diesem Zusammenhang ist überraschend, dass der vereinfachte Algorithmus bei diesem Problem schlechter als der nur-aktivierte abschneidet, obwohl er von der Zahl der gefundenen Einschränkungen wie auch der praktischen Laufzeit her zwischen dem neuen und dem nur-aktivierten liegt. Ein Blick auf Abbildung 4.4 liefert die Erklärung: Dort wird die Anzahl der Suchschritte gezeigt, bei denen tatsächlich eine Auswahl stattfindet. Je niedriger der Wert, desto mehr Einschränkungen wurden von dem entsprechenden Algorithmus gefunden. Der iterative und der neue Algorithmus liegen hier gleichauf – wieder verwirklicht der neue Algorithmus bei allen Instanzen alle zulässigen Suchraumeinschränkung.[5] Die beiden anderen Algorithmen folgen mit einigem Abstand, liegen aber nahe beieinander. Dieser Befund legt nahe, dass der vereinfachte Algorithmus bei diesem Problem die entscheidenden Einschränkungen nicht findet, die dem neuen Algorithmus so gute Ergebnisse ermöglichen. Diese Einschränkungen müssen folglich durch den Programmabschnitt zustandekommen, der im vereinfachten ausgelassen wurde.

Eine weitere Beobachtung ist, dass der iterative Algorithmus zwar genausoviele Einschränkungen wie der neue findet, bei der Laufzeit aber das Schlusslicht bildet. Offenbar reichen schon die bei den getesteten Random-Placement-Instanzen auftretenden 50 bis 100 Aktivitäten pro Ressource dazu aus, dass die schlechtere asymptotische Laufzeit den Vorteil der einfacheren Datenstrukturen überwiegt.

Insgesamt ist festzustellen, dass das Einbeziehen optionaler Aktivitäten beim Edgefinding zu einer kürzeren Gesamtlaufzeit führen kann. Der neue Algorithmus liegt besonders bei vielen optionalen Aktivitäten eindeutig vor den bisher bekannten Algorithmen. Sind wie bei den betrachteten Job-Shop-Instanzen

[5] Bei der Anzahl der Fixpunktiterationen gibt es in Einzelfällen geringfügige Abweichungen, die aber wieder unter der Promillegrenze liegen. Siehe dazu auch Abbildung A.1 auf Seite 80.

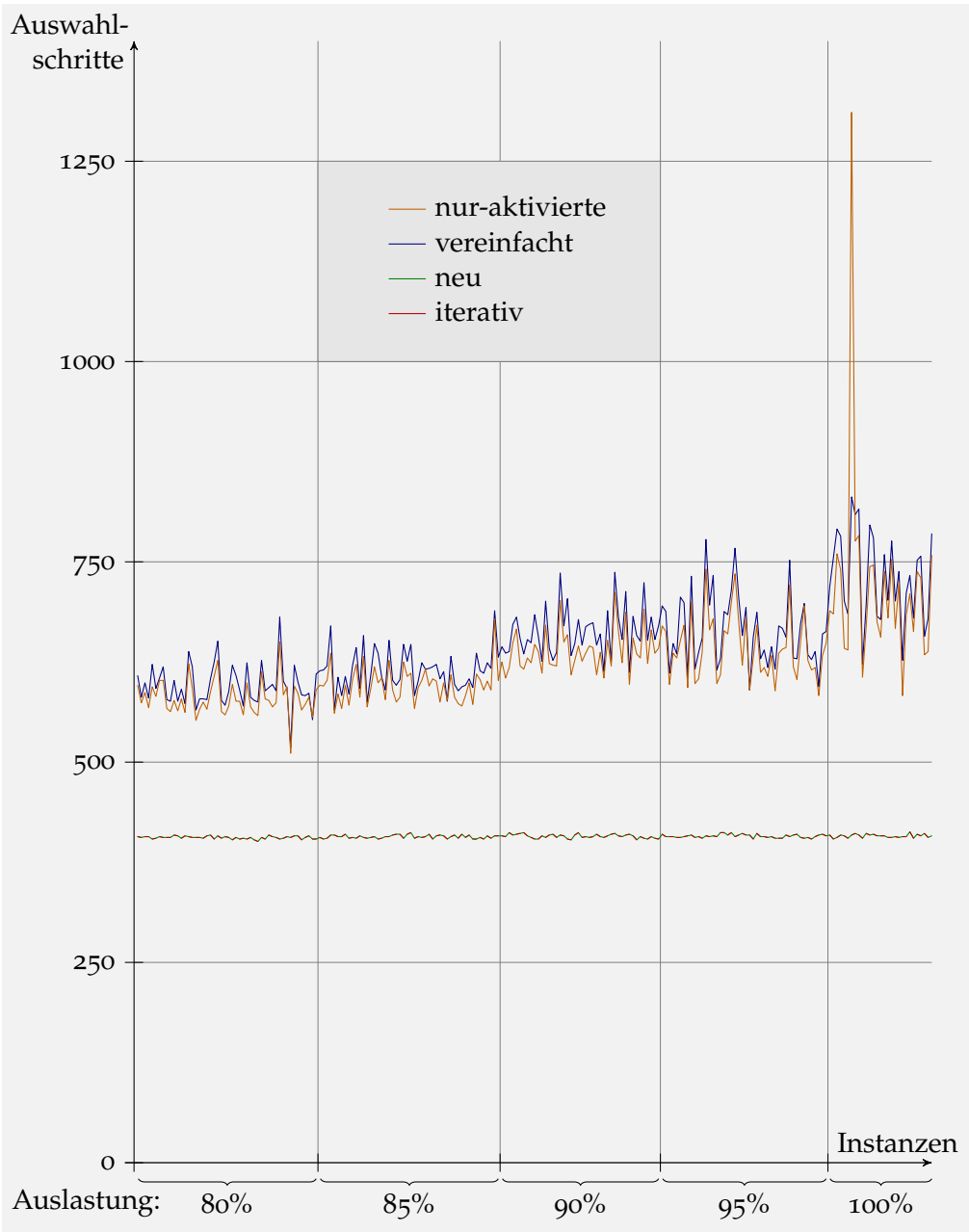


Abbildung 4.4: Anzahl der Auswahlschritte bei den verschiedenen Edge-finding-Algorithmen für Standardinstanzen des Random-Placement-Problems

weniger Aktivitäten vorhanden und nur wenige davon optional, ist mal der vereinfachte und mal der nur-aktivierte-Algorithmus am schnellsten. Nur wenn überhaupt keine optionalen Aktivitäten beteiligt sind, weist der nur-aktivierte-Algorithmus die kürzeste Laufzeit auf.

Durch die Messungen am Random-Placement-Problem wurde deutlich, dass der neu entwickelte Algorithmus besonders bei vielen und optionalen Aktivitäten Vorteile gegenüber den übrigen Algorithmen bietet. Bei dem von Sarkarati betrachteten Satellitenplanungsproblem [Saro7] dürfte sich dieser Effekt noch verstärken, weil dort nicht nur (wie bei den Random-Placement-Instanzen) 50 bis 100, sondern mehrere hundert oder sogar mehrere tausend Aktivitäten pro Ressource vorliegen, die alle optional sind.

5. Zusammenfassung und Ausblick

> Alles nimmt ein gutes Ende
> für den, der warten kann.
>
> *Leo Tolstoi, Schriftsteller*

In dieser Diplomarbeit wurde ausgehend von den Grundlagen der Constraint-Programmierung untersucht, wie Zeitplanungsprobleme modelliert und gelöst werden können. Der Schwerpunkt lag dabei auf exklusiven Ressourcen, an denen optionale Aktivitäten beteiligt sind. Diese stellen eine besondere Herausforderung dar, weil sie bei der Modellierung zahlreicher praxisrelevanter Probleme eine zentrale Rolle spielen, herkömmliche Filteralgorithmen aus ihnen aber keine Einschränkungen ableiten können, welche die Domänen – und damit den Suchraum – einschränken. Zur Beschleunigung der Suche sind solche Algorithmen jedoch äußerst wünschenswert.

In dieser Arbeit wurde insbesondere die Edgefinding-Regel betrachtet, die ein zentrales Werkzeug zum Filtern an exklusiven Ressourcen ist. Es wurde ein neuer Algorithmus entwickelt, der optionale Aktivitäten in das Filtern miteinbezieht. Bei der mathematischen Untersuchung dieses Algorithmus konnte gezeigt werden, dass er

- korrekt ist und damit keine unzulässigen Einschränkungen vornimmt,
- mit einer asymptotischen Komplexität von $\mathcal{O}(n^2)$ implementiert werden kann und
- fast alle Anwendungen der Edgefinding-Regel findet.

Bei den Experimenten mit praktischen Problemen zeigte sich, dass kaum Einschränkungen verloren gehen: Gegenüber einem optimalen Algorithmus, der alle Einschränkungen findet, wurden nur bis zu einem Promille zusätzliche Fixpunktiterationen benötigt, nach deren Abschluss die gleiche Anzahl von Einschränkungen gefunden wurde.

Die abschließenden Zeitmessungen zum Vergleich verschiedener Edgefinding-Algorithmen führten zu dem Ergebnis, dass der neue Algorithmus insbesondere bei Problemen mit vielen optionalen Aktivitäten deutlich weniger Zeit

benötigt als die bisher bekannten Algorithmen: Bei den Messungen am Random-Placement-Problem war er im Durchschnitt über 30% besser als der beste seiner Konkurrenten.

Sind wie bei den untersuchten Instanzen des Job-Shop-Problems nicht so viele optionale Aktivitäten beteiligt, ist die Lage weniger eindeutig: Der neue Algorithmus findet zwar auch hier mehr Einschränkungen, die dadurch gewonnene Zeit steht aber der längeren Laufzeit pro Aufruf gegenüber, sodass nur in wenigen Fällen ein Zeitgewinn möglich ist. Eine vereinfachte Variante des neuen Algorithmus ermöglicht aber auch hier gute Resultate.

Eine interessante Frage für zukünftige Untersuchungen ist daher, wie ausgehend von der Anzahl der beteiligten Aktivitäten und dem Anteil der optionalen Aktivitäten eine Heuristik entwickelt werden kann, die möglichst zuverlässig den schnellsten Algorithmus auswählt. Die in dieser Arbeit präsentierten Messergebnisse liefern dafür zwar erste Anhaltspunkte, für zuverlässige Aussagen wäre aber die Betrachtung weiterer Problemklassen notwendig.

Auch die Untersuchung verschiedener Suchstrategien für optionale Aktivitäten bietet Gelegenheit für künftige Forschung: Die in Abschnitt 2.4.1 beschriebene Methode wurde im Rahmen dieser Diplomarbeit aufgrund einiger theoretischer Überlegungen entwickelt, aber nicht mit anderen Ansätzen verglichen.

A. Weitere Messergebnisse

Tabelle A.1: Anzahl der Fixpunktiterationen bei verschiedenen Job-Shop-Problemen. Ein direkter Vergleich ist nur zwischen dem neuen, vereinfachten und iterativen Algorithmus sinnvoll, da der nur-aktivierte-Algorithmus eine andere Anzahl von Backtracking-Schritten aufweist. Bemerkenswert sind vor allem die sehr geringen, aber vorhandenen Abweichungen zwischen diesen drei Algorithmen. Siehe auch Fußnote 1 auf Seite 69.

	neu	**vereinfacht**	**nur aktivierte**	**iterativ**
abz5-alt	115995	115995	128837	115996
orb01-alt	1445423	1445423	1466121	1445340
orb02-alt	202854	202854	282923	202907
orb07-alt	2215950	2215949	2364201	2215907
orb10-alt	3572	3572	3600	3572
la16-alt	177864	177864	187223	177864
la17-alt	831	831	888	831
la18-alt	665371	665418	710744	665330
la19-alt	65991	65991	136212	65988
la20-alt	1475	1475	1502	1475
abz5	169502	169502	169502	169502
orb01	885970	885970	885970	885970
orb02	299851	299851	299851	299851

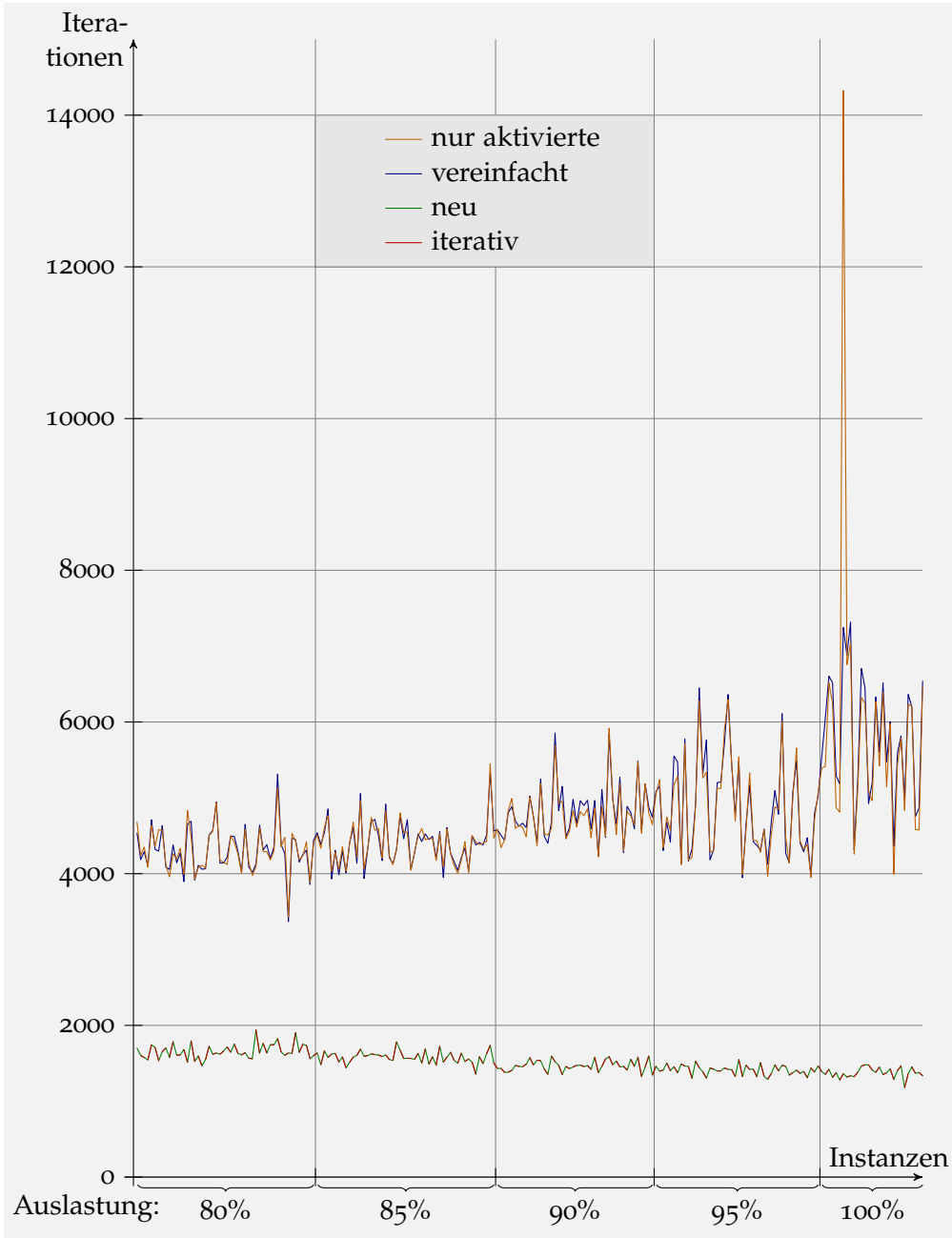


Abbildung A.1: Fixpunktiterationen bei der Lösungssuche für Standardinstanzen des Random-Placement-Problems mit den einzelnen Edgefinding-Algorithmen. Siehe auch Fußnote 5 auf Seite 74.

Literaturverzeichnis

[ABZ88] Adams, Joseph, Egon Balas und Daniel Zawack. »The Shifting Bottleneck Procedure for Job Shop Scheduling«. In: *Management Science* 34.3 (März 1988). 391–401. Focussed Issue on Heuristics. ISSN 0025-1909.

[Bar02] Barták, Roman. *Constraint-Based Scheduling. An Introduction For Newcomers*. Technical Report TR 2002/2. Charles University in Prague, Institute for Theoretical Computer Science. 2002. URL: http://www.cs.sfu.ca/CC/827/havens/papers/IMS2003-1.pdf (besucht am 07. 09. 2007).

[BF99] Beck, J. Christopher, und Mark S. Fox. »Scheduling Alternative Activities«. In: *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI-99)*. AAAI Press, 1999. ISBN 0-262-51106-1. 680–687. URL: http://4c.ucc.ie/web/upload/publications/inProc/AltActs.aaai99.pdf (besucht am 07. 09. 2007).

[BMR04] Barták, Roman, Tomáš Müller und Hana Rudová. »A New Approach to Modeling and Solving Minimal Perturbation Problems«. In: *Recent Advances in Constraints*. Joint ERCIM/CoLogNET International Workshop on Constraint Solving and Constraint Logic Programming, CSCLP 2003. Lecture Notes on Computer Science 3010. Berlin Heidelberg: Springer, 2004. ISBN 978-3-540-21834-0. DOI: 10.1007/b96986. 233–249.

[BP95] Baptiste, Philippe, und Claude le Pape. »A Theoretical and Experimental Comparison of Constraint Propagation Techniques for Disjunctive Scheduling«. In: *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*. Morgan Kaufmann, 1995. 600–606. URL: http://www.ilog.cn/products/optimization/tech/research/ijcai95.pdf (besucht am 07. 09. 2007).

[BP96] Baptiste, Philippe, und 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*. Liverpool, United Kingdom 1996.

[BPN01] Baptiste, Philippe, Claude le Pape und Wim Nuijten. *Constraint-Based Scheduling. Applying Constraint Programming to Scheduling Problems*. Boston: Kluwer Academic Publishers, 2001. ISBN 0792374088.

[BPN95] Baptiste, Philippe, Claude le Pape und Wim Nuijten. »Constraint-Based Optimization and Approximation for Job-Shop Scheduling«. In: *Proceedings of the AAAI-SIGMAN Workshop on Intelligent Manufacturing Systems*. 1995. 5–16.

[Bru99] Brucker, Peter. »Complex Scheduling Problems«. In: *Osnabrücker Schriften zur Mathematik*. Reihe P. 214 (1999). URL: `ftp://ftp.mathematik.uni-osnabrueck.de/pub/osm/preprints/complex.ps.gz` (besucht am 07. 09. 2007).

[CL94] Caseau, Yves, und François Laburthe. »Improved CLP Scheduling with Task Intervals«. In: *Proceedings of the 11th International Conference on Logic Programming*. ICLP'94. Cambridge, USA: The MIT Press, 1994. URL: `http://www.cs.sfu.ca/research/groups/ISL/library/Resource.scheduling/caseau94improved.pdf` (besucht am 14. 09. 2007).

[Col96] Colombani, Yves. »Constraint programming: an efficient and practical approach to solving the job-shop problem«. In: *Principles and Practice of Constraint Programming*. CP96. Lecture Notes in Computer Science 1118. Berlin Heidelberg: Springer, 1996. ISBN 3-540-61551-2. DOI: `10.1007/3-540-61551-2_72`. 149–163.

[CP94] Carlier, Jaques, und Eric Pinson. »Adjustments of head and tails for the job-shop problem«. In: *European Journal of Operational Research* 78 (2 1994). 146–161. ISSN 0377-2217. DOI: `10.1016/0377-2217(94)90379-4`.

[Dec92] Dechter, Rina. »From local to global consistency«. In: *Artificial Intelligence* 55.1 (1992). 87–107. ISSN 0004-3702. DOI: `10.1016/0004-3702(92)90043-W`.

[DI06] Demetrescu, Camil, und Giuseppe F. Italiano. »Dynamic shortest paths and transitive closure: Algorithmic techniques and data structures«. In: *Journal of Discrete Algorithms* 4.3 (2006). 353–383. URL: `http://www.dis.uniroma1.it/~demetres/docs/dynamic-techniques.pdf` (besucht am 07. 09. 2007).

[EMG+01] Ehrig, Hartmut, Bernd Mahr, Martin Große-Rhode, Felix Cornelius und Philip Zeitz. *Mathematisch-strukturelle Grundlagen der Informatik*. 2. Aufl. Berlin Heidelberg: Springer, 2001. ISBN 3-540-41923-3.

[FA97] Frühwirth, Thom, und Slim Abdennadher. *Constraint-Programmierung. Grundlagen und Anwendungen*. Berlin Heidelberg: Springer, 1997. ISBN 3-540-60670-X. URL: http://www.informatik.uni-ulm.de/pm/fileadmin/pm/home/fruehwirth/buch.html (besucht am 07. 09. 2007).

[FHK+92] Frühwirth, Thom, Alexander Herold, Volker Küchenhoff u. a. »Constraint Logic Programming. An Informal Introduction«. In: *Logic Programming in Action*. Hg. von Gérard Comyn, Norbert E. Fuchs und Michael J. Ratcliffe. Lecture Notes in Computer Science 636. Berlin Heidelberg: Springer, 1992. ISBN 3-540-55930-2. DOI: 10.1007/3-540-55930-2_2. 3–35.

[FT63] Fisher, Henry, und Gerald L. Thompson. »Probabilistic learning combinations of local job-shop scheduling rules«. In: *Industrial scheduling*. Hg. von John F. Muth und Gerald Luther Thompson. Englewood Cliffs, New Jersey: Prentice-Hall, 1963. 225–251.

[GG04] Geske, Ulrich, und Hans-Joachim Goltz. »Automatische und interaktive Stundenplanung«. In: *Informatik – Forschung und Entwicklung* 19.2 (Nov. 2004). 65–73. ISSN 0178-3564. DOI: 10.1007/s00450-004-0160-x.

[GJ79] Garey, Michael R., und David S. Johnson. *Computers and Intractability. A Guide to the Theory of NP-Completeness*. New York: W. H. Freeman, 1979. ISBN 0-7167-1045-5.

[HMS+03] Hoche, Matthias, Henry Müller, Hans Schlenker und Armin Wolf. »firstcs—A Pure Java Constraint Programming Engine«. In: *2nd International Workshop on Multiparadigm Constraint Programming Languages*. MultiCPL'03. 2003. URL: http://uebb.cs.tu-berlin.de/MultiCPL03/Proceedings.MultiCPL03.RCoRP03.pdf (besucht am 07. 09. 2007).

[Hoe01] Hoeve, Willem-Jan van. »The Alldifferent Constraint. A Survey«. In: *Sixth Annual Workshop of the ERCIM Working Group on Constraints*. Prague 2001. URL: http://www.andrew.cmu.edu/user/vanhoeve/#ercim2001 (besucht am 07. 09. 2007).

[HW07] Hofstedt, Petra, und Armin Wolf. *Einführung in die Constraint-Programmierung*. eXamen.press. Berlin Heidelberg: Springer, 2007. ISBN 978-3-540-23184-4.

[Kuh07] Kuhnert, Sebastian. »Efficient Edge-Finding on Unary Resources with Optional Activities«. In: *International Conference on Applications of Declarative Programming and Knowledge Management (INAP 2007)*. Würzburg 2007. URL: http://www1.informatik.uni-wuerzburg.de/databases/INAP/2007/ (besucht am 07. 09. 2007).

[Law84] Lawrence, Stephen R. Resource constrained project scheduling: an experimental investigation of heuristic scheduling techniques (Supplement). Technical Report. Graduate School of Industrial Administration, Carnegie-Mellon University, Pittsburgh, Pennsylvania, 1984.

[Rég94] Régin, Jean-Charles. »A Filtering Algorithm for Constraints of Difference in CSPs«. In: *Proceedings of the 12th National Conference on Artificial Intelligence*. AAAI 1994. AAAI Press, 1994. ISBN 0-262-61102-3. 362–367. URL: http://www.constraint-programming.com/people/regin/papers/alldiff.pdf (besucht am 07. 09. 2007).

[RV] Rudová, Hana, und Kamil Veřmiřovský. *Random Placement Problem (RPP)*. URL: http://www.fi.muni.cz/~hanka/rpp/ (besucht am 07. 09. 2007).

[San04] Sankowski, Piotr. »Dynamic transitive closure via dynamic matrix inverse. Extended abstract«. In: *Proceedings of the 45th Annual IEEE Symposium on Foundations of Computer Science*. Los Alamitos, CA, USA: IEEE Computer Society, 2004. ISBN 0-7695-2228-9. DOI: 10.1109/FOCS.2004.25. 509–517.

[Sar07] Sarkarati, Mehran. »Scheduling Optional Tasks in an Over-Constrained System of Dependent Unary Resources, Using a COP Approach«. Master's Thesis. Technische Universität Berlin, 2007.

[Sch06] Schutt, Andreas. »Entwicklung suchraumeinschränkender Verfahren zur Constraint-basierten Lösung kumulativer Ressourcenplanungsprobleme«. Diplomarbeit. Humbold-Universität zu Berlin, 2006.

[SWS06] Schutt, Andreas, Armin Wolf und Gunnar Schrader. »Not-First and Not-Last Detection for Cumulative Scheduling in $\mathcal{O}(n^3 \log n)$«. In: *Declarative Programming for Knowledge Management*. 16th International Conference on Applications of Declarative Programming and Knowledge Management, INAP 2005. Lecture Notes in Computer Science 4369. Berlin Heidelberg: Springer, 2006. ISBN 978-3-540-69233-1. DOI: 10.1007/11963578_6. 66–80.

[SWV92] Storer, Robert H., S. David Wu und Renzo Vaccari. »New search spaces for sequencing instances with application to job shop scheduling«. In: *Management Science* 38.10 (1992). 1495–1509. ISSN 0025-1909.

[TL00] Torres, Philippe, und Pierre Lopez. »On Not-First/Not-Last Conditions in Disjunctive Scheduling«. In: *European Journal of Operational Research* 127 (2000). 332–343. ISSN 0377-2217. URL: http://hal.archives-ouvertes.fr/hal-00022745/en/ (besucht am 07. 09. 2007).

[VBČ04] Vilím, Petr, Roman Barták und Ondřej Čepek. »Unary Resource Constraint with Optional Activities«. In: *Principles and Practice of Constraint Programming*. 10th International Conference, CP 2004. Lecture Notes in Computer Science 3258. Berlin Heidelberg: Springer, 2004. ISBN 978-3-540-23241-4. DOI: 10.1007/b100482. 62–76.

[VBČ05] Vilím, Petr, Roman Barták und Ondřej Čepek. »Extension of $\mathcal{O}(n \log n)$ Filtering Algorithms for the Unary Resource Constraint to Optional Activities«. In: *Constraints* 10.4 (2005). 403–425. DOI: 10.1007/s10601-005-2814-0.

[Vil02] Vilím, Petr. »Batch Processing With Sequence Dependent Setup Times: New Results«. In: *Proceedings of the 4th Workshop of Constraint Programming for Decision and Control*. CPDC'02. 2002. URL: http://ktilinux.ms.mff.cuni.cz/~vilim/cpdc2002.pdf (besucht am 07. 09. 2007).

[Vil04] Vilím, Petr. »$\mathcal{O}(n \log n)$ Filtering Algorithms for Unary Resource Constraint«. In: *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*. First International Conference, CPAIOR 2004. Lecture Notes in Computer Science 3011. Berlin Heidelberg: Springer, 2004. ISBN 978-3-540-21836-4. DOI: 10.1007/b96957. 335–347.

[Vil07] Vilím, Petr. »Global Constraints in Scheduling«. Diss. Charles University in Prague, 2007.

[Wol03] Wolf, Armin. »Pruning while Sweeping over Task Intervals«. In: *Principles and Practice of Constraint Programming*. 9th International Conference, CP 2003. Lecture Notes in Computer Science 2833. Berlin Heidelberg: Springer, 2003. ISBN 978-3-540-20202-8. DOI: 10.1007/b13743. 739–753.

[Wol05] Wolf, Armin. »Better Propagation for Non-preemptive Single-Resource Constraint Problems«. In: *Recent Advances in Constraints*. Joint ERCIM/CoLogNet International Workshop on Constraint Solving and Constraint Logic Programming, CSCLP2004. Lecture Notes in Computer Science 3419. Berlin Heidelberg: Springer, 2005. ISBN 3-540-25176-6. DOI: 10.1007/11402763_15.

[WS05] Wolf, Armin, und Hans Schlenker. »Realising the Alternative Resources Constraint«. In: *Applications of Declarative Programming and Knowledge Management*. 15th International Conference on Applications of Declarative Programming and Knowledge Management, INAP 2004. Lecture Notes in Computer Science 3392. Berlin Heidelberg: Springer, 2005. ISBN 978-3-540-25560-4. DOI: 10.1007/11415763_12. 185–199.

[WS06] Wolf, Armin, und Gunnar Schrader. »$\mathcal{O}(n \log n)$ Overload Checking for the Cumulative Constraint and Its Application«. In: *Declarative Programming for Knowledge Management*. 16th International Conference on Applications of Declarative Programming and Knowledge Management, INAP 2005. Lecture Notes in Computer Science 4369. Berlin Heidelberg: Springer, 2006. ISBN 978-3-540-69233-1. DOI: 10.1007/11963578_8. 88–101.

[YN92] Yamada, Takeshi, und Ryohei Nakano. »A Genetic Algorithm Applicable to Large-Scale Job-Shop Problems«. In: *Parallel instance solving from nature* 2 (1992). 281–290.

[Zan06] Zander, Raphael. »Constraint-basierte/objekt-orientierte Fahrplan-Erzeugung für ein Straßenbahnnetz mit Behandlung von Zugangs- und Übergangszeiten«. Diplomarbeit. Universität Potsdam, 2006.

Index

Seiten, auf denen ein Begriff definiert wird, sind fett angegeben.

Erklärung

Die selbständige und eigenhändige Anfertigung versichere ich an Eides Statt.

Berlin, den 18. September 2007 _____