Algorithmen zur Modellierung und effizienten Verarbeitung disjunktiver Constraints über endlichen Mengen

Alexander Nareyek
alex@ai-center.com

Diplomarbeit an der Technischen Universität Berlin, eingereicht am 20. Dezember 1996, publiziert als:

Alexander Nareyek. 1998. Algorithmen zur Modellierung und effizienten Verarbeitung disjunktiver Constraints über endlichen Mengen. GMD Research Series, no. 1. GMD - Forschungszentrum Informationstechnik GmbH, Sankt Augustin, Germany.

Danksagung

Es ist mir eine Freude, an dieser Stelle den Mitarbeitern der GMD-FIRST-Projekte WISPRO und VERMEIL für die Unterstützung und das angenehme Arbeitsklima sowie meiner Mutter für das Korrekturlesen zu danken. Vor allem möchte ich mich bei meinem Betreuer Herrn Prof. Dr. Geske bedanken, der mir viele wertvolle Hinweise auf Verbesserungsmöglichkeiten gab.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einführung

Der folgende Abschnitt liefert zunächst einen allgemeinen Überblick zur constraintlogischen Programmierung. In Abschnitt 1.2 wird anschließend näher auf den behandelten Themenbereich eingegangen und ein Überblick zum Aufbau der weiteren Arbeit gegeben.

1.1 Constraintlogische Programmierung

> „Constraint programming is the closest computer science has come to the Holy Grail of programming: the user states the problem, the computer solves it." [Freuder96]

> „Programmieren in Prolog bedeutet in hohem Mass, Spezifikationen zu schreiben. Es wird vornehmlich spezifiziert, *was* zu erreichen ist und kaum *wie* die Abarbeitungsschritte beschaffen sein müssen." [Geske93]

Logische Programmierung und Constraint-Programmierung haben eine stark deklarative Orientierung. Die Constraint-Programmierung zielt jedoch sehr speziell auf den Bereich der kombinatorischen Probleme, während der Schwerpunkt der Logischen Programmierung bei der Inferenzmethodik der Regelverarbeitung liegt. Durch die Vereinigung der beiden Ansätze wird eine Kombination der Ausdrucksstärke, Flexibilität und Effizienz erreicht. Die Vereinigung der Unifikation der Logischen Programmierung mit dem allgemeineren Mechanismus des Lösens von Constraints führt zu einer sehr engen Kopplung der Verfahren[1].

Das constraintlogische Programmierparadigma findet sich beispielsweise in Sprachen wie Clp(R) ([Heintze92]), ECLiPSe ([ECRC93]), Chip ([Simonis95]) oder Protos-L ([Beierle93]). Innerhalb dieser Arbeit wurde die Programmiersprache Chip verwendet.

Da im weiteren hauptsächlich der Constraint-Anteil des constraintlogischen Programmierparadigmas verwendet wird, geben die folgenden Abschnitte eine kurze Einführung zur Constraint-Programmierung.

1.1.1 Constraint-Netzwerke

Ein Constraint-Netzwerk (CN) besteht aus einer Menge von Variablen $X = \{X_1, ..., X_n\}$, von denen jede mit einem Wertebereich $D_1, ..., D_n$ verbunden ist, und einer Menge von Constraints $C = \{C_1, ..., C_m\}$ über X.

[1] Eine Einführung in die constraintlogische Programmierung geben z.B. [Hentenryck89] und [Jaffar94].

Die Wertebereiche D_i können im allgemeinen diskret wie kontinuierlich sein. In der vorliegenden Arbeit werden nur diskrete Wertebereiche betrachtet.

Ein Constraint-Netzwerk läßt sich durch einen Graphen beschreiben. Die folgende Abbildung zeigt einen derartigen Graphen:

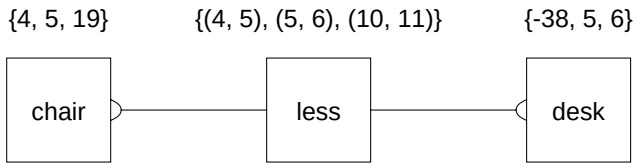


Abbildung 1.1: Beispiel eines Constraint-Netzwerkes (nach [Guesgen92])

Die Variable *chair* hat dabei einen Wertebereich von $\{4, 5, 19\}$, die Variable *desk* einen von $\{-38, 5, 6\}$. Diese beiden Variablen sind durch das Constraint $less$ verbunden, das die Tupel der zulässigen Kombinationen explizit durch die Menge $\{(4, 5), (5, 6), (10, 11)\}$ festlegt. Die gängigen Programmiersprachen lassen allerdings nur implizite Beziehungen zwischen den Variablen zu ($chair < desk$), was deren Ausdruckskraft erheblich mindert.

1.1.2 Suche

Zum Finden einer bzw. aller Lösungen eines Constraint-Systemes (Constraint Satisfaction Problem) können verschiedenste Suchstrategien angewendet werden. Darunter fallen Backtracking, Lookahead, Backjumping etc. ([Dechter90]), aber auch iterative Methoden wie Simulated Annealing, Genetische Algorithmen, Tabu Search usw. ([Pesant96]) lassen sich anwenden.

In der constraintlogischen Programmierung wird diese Suche durch in der Logischen Programmierung erstellte Inferenzverfahren durchgeführt.

Beim Beispiel der Abbildung 1.1 scheint die Anwendung komplexer Suchstrategien nicht angebracht, da sich die möglichen Lösungen leicht als $\{chair = 4, desk = 5\}$ und $\{chair = 5, desk = 6\}$ identifizieren lassen. Im allgemeinen fallen die Probleme jedoch in die Klasse der NP-vollständigen Probleme.

Ein weiteres Problem ist es, die nach bestimmten Kriterien **beste** Lösung zu finden (Constraint Optimization Problem). Hierfür wird meist ein Branch-and-Bound-Ansatz verwendet.

1.1.3 Propagation

Um den Suchraum zu verringern, können Verfahren der Propagation verwendet werden. Diese prüfen vor Beginn bzw. während der Suche, ob sich die Wertebereiche von Variablen einschränken lassen. In Weiterführung des Beispiels von Abbildung 1.1 könnten z.B. von vorneherein bereits einige Werte entfallen:

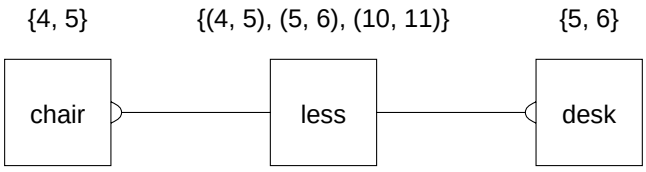


Abbildung 1.2: Constraint-Netzwerk nach der Propagation

Dies macht die Suche nach einer Lösung allerdings nicht überflüssig, da zwar einzelne Werte aus den Wertebereichen entfernt werden können ($chair \neq 19, desk \neq -38$), ungültige Kombinationen aber weiterhin möglich sind ($\{chair = 4, desk = 6\}$ und $\{chair = 5, desk = 5\}$).

Die Propagation kann global und lokal erfolgen. Im Falle der globalen Propagation werden die Auswirkungen auf sämtliche Variablen des Constraint-Systems untersucht, was in der Regel einen nicht beherrschbaren Aufwand nach sich zieht. Bei der lokalen Propagation wird deshalb nur ein beschränktes Umfeld einbezogen. Zum Erreichen der lokalen Propagation werden Konsistenz-Mechanismen, wie z.B. *Arc consistency* ([Bessière95]) verwendet.

Da die Untersuchung sämtlicher Werte der Wertebereiche der Variablen äußerst aufwendig ist, geschieht meist eine intervall-basierte Auswertung, d.h. es werden nur die Maxima und Minima der Domänen betrachtet.

1.2 Disjunktive Verknüpfungen

Zunächst wird in Abschnitt 1.2.1 die behandelte Problemstellung erläutert. Die Darstellung bisheriger Lösungsansätze erfolgt in Abschnitt 1.2.2. Anschließend liefert Abschnitt 1.2.3 einen Überblick zur vorliegenden Arbeit.

1.2.1 Problemstellung

In der Praxis enthalten Problembeschreibungen häufig statt mathematischer Zusammenhänge, die als implizite Beschreibung von Alternativen angesehen werden können, **explizit** aufgeführte Alternativen. So stehen bei der Wahl des Produktionsverfahrens, der Entscheidung über eine Auftragsvergabe oder der Durchführung einer Diagnose meist konkrete Möglichkeiten zur Auswahl.

Eine Alternative läßt sich durch eine spezielle Wertausprägung bestimmter Variablen charakterisieren. Die folgende Tabelle verdeutlicht diese Struktur:

Alternative	Ausgaben	Einnahmen			
		Jahr 1	Jahr 2	Jahr 3	Jahr 4
1	102200	0	12200	32500	85000
2	79400	52700	52700	19500	1200
3	64700	24900	24900	23800	0

Tabelle 1.1: Investitionsalternativen

Die Formulierung expliziter Alternativen ist aber nicht nur über Variablen möglich, sondern auch über Constraints. Die so modellierten Relationen stellen dann sogenannte *Meta-Constraints* dar (z.B. $(A > B) \rightarrow (A \neq C)$). Diese werden für die Darstellung komplexer Zusammenhänge benötigt, wie z.B. die Modellierung technischer Schaltkreise.

Es müssen geeignete Repräsentationen für die expliziten Relationen (in Form von Alternativen-Tabellen) gefunden werden, die das Aufsetzen einer weiteren Verarbeitung ermöglicht. Das zentrale Problem stellt dabei die effiziente Verarbeitung der Alternativen dar.

1.2.2 Bisherige Lösungsansätze

Es existieren bereits vielfältige Ansatzpunkte zum Umgang mit Disjunktionen (bzw. Alternativen). Im einfachsten Fall wird die Disjunktion nicht als geschlos-

sener Ausdruck dem Constraint-Solver übergeben, sondern anderweitig verarbeitet. Logische Programmiersprachen stellen hier z.B. den Mechanismus des Backtrackings zur Verfügung. Durch diesen lassen sich die einzelnen Alternativen über vollständige Enumeration testen. Innerhalb größerer constraintlogischer Systeme ist eine derartige Vorgehensweise jedoch höchst ineffizient, da sich ein exponentieller Aufwand ergibt.

Eine Übergabe der Disjunktion als geschlossenen Ausdruck an den Constraint-Solver erscheint sinnvoller, da der Suchraum hierbei bereits erheblich eingeschränkt werden kann. Studien auf dem Gebiet der Produktionsplanung belegen dies (siehe [Breitinger94a] und [Nareyek95]). Die wenigsten verfügbaren Constraint-Solver lassen jedoch die disjunktive Verknüpfung von Constraints zu. In ECLiPSe können dafür beispielsweise sogenannte *Constraint Handling Rules* verwendet werden (siehe [Frühwirth95]). Auch Oz bietet inzwischen spezielle Konstrukte einer *guarded disjunction* (siehe [Smolka94]). All diese Ansätze sind sehr speziell und eignen sich nur sehr eingeschränkt zur Darstellung expliziter (Meta-)Relationen.

Eine weitere Herangehensweise zur Bewältigung des Problems der Disjunktion ist die der linearen Relaxation (siehe z.B. [Beringer93]). Diese Ansätze, wie z.B. Variationen des Algorithmus von Balas, führen jedoch innerhalb der ganzzahligen Programmierung wegen der gwaltigen Anzahl zusätzlich abzusetzender Constraints zu einem schlechten Laufzeitverhalten ([Balas85], [Beringer93]).

Der in dieser Arbeit verfolgte Ansatz fällt in die Kategorie der konstruktiven Lösungen. Bei diesem Ansatz wird die Problembeschreibung in ein konjunktiv verknüpftes Constraint-System transformiert, welches von gewöhnlichen Constraint-Solvern abgearbeitet werden kann. Eine in [Mitra94] entwickelte Lösung beschäftigt sich z.B. in diesem Rahmen mit der Darstellung logischer Meta-Constraints. Dabei muß die Problembeschreibung allerdings in einer speziellen *Logical Constraint in the Implication Form* (LCIF) vorliegen. Auch läßt die Modellierung nur implizite Relationen zu, wodurch die Darstellung komplexerer Zusammenhänge stark beeinträchtigt wird. Die in dieser Arbeit vorgestellten Algorithmen weisen diesbezüglich keinerlei Einschränkungen auf.

1.2.3 Überblick

Die Bewältigung der Disjunktion basiert im wesentlichen auf dem Konzept sogenannter Schaltvariablen. Diese werden in Kapitel 2 eingeführt.

Die Transformation expliziter Alternativen einer gegebenen Relation in ein konjunktiv verknüpftes Constraint-System behandelt Kapitel 3. Es werden mehrere Algorithmen vorgestellt und anschließend miteinander verglichen.

Mittels der eingeführten Algorithmen können nicht nur einfache Relationen über

Variablen, sondern auch Meta-Constraints behandelt werden. Dabei ist es möglich, auch implizite Relationen einzubinden. Diesem Themenkomplex widmet sich Kapitel 4.

Die grundlegenden Konzepte dieser Arbeit sind ebenfalls in [Nareyek96a] und [Nareyek96b] zu finden.

Kapitel 2

Schaltvariablen

Als Schlüsselkonzept zur Darstellung von Disjunktionen können in der ganzzahligen Constraint-Programmierung Schaltvariablen eingesetzt werden.

Schaltvariablen besitzen einen Wertebereich von Null und Eins. Sie werden in bestehende Constraints integriert, und erlauben über die Festlegung auf Null oder Eins die Steuerung der Gültigkeit der Constraints.

Es kann zwischen ein- und mehrdeutigen Schaltvariablen unterschieden werden. Eindeutige Schaltvariablen besitzen **genau dann** einen Wert von Eins, wenn das zugrundeliegende Constraint gültig ist. Mehrdeutige Schaltvariablen nehmen **nur in einem Fall** (Gültigkeit oder Ungültigkeit) einen eindeutigen Wert an.

2.1 Eindeutige Schaltvariblen

Abschnitt 2.1.1 stellt zunächst das in [Breitinger94a] und [Breitinger94b] eingeführte Verfahren der „Disjunction as Constraints“ vor. Anschließend wird in Abschnitt 2.1.2 eine Verallgemeinerung dieses Ansatzes vorgenommen, die die Steuerung der Gültigkeit allgemeiner Constraints ermöglicht.

2.1.1 „Disjunction as Constraints“

Dieses Verfahren wurde für die Behandlung von Disjunktionen im Rahmen von Scheduling-Problemen entwickelt. Dabei geht darum, daß zwei Arbeitsaufgaben nicht gleichzeitig auf einer Maschine bearbeitet werden dürfen. $Beginn_1$ und $Beginn_2$ geben die Startpunkte, $Dauer_1$ und $Dauer_2$ die Ausführungszeiten der Arbeitsaufgaben an. Entweder wird Arbeitsaufgabe 1 vor Arbeitsaufgabe 2 bearbeitet, oder es gilt die umgekehrte Reihenfolge:

$$Beginn_2 \geq Beginn_1 + Dauer_1 \quad \vee \quad Beginn_1 \geq Beginn_2 + Dauer_2$$

Die beiden Ungleichungen bzw. Constraints werden so erweitert, daß jeweils nur eines wahr sein kann. Hierfür ist eine weitere Variable notwendig: $Bool \in \{0, 1\}$ dient als Schaltvariable, die das eine Constraint außer Kraft setzt (trivial erfüllt), falls sich das andere Constraint als wahr erweist. Dies geschieht über eine Konstante Max, die ein nicht überschreitbares Maximum darstellt. Die folgenden Constraints realisieren die Vorgehensweise:

$$\begin{aligned} Beginn_2 + Bool \times Max &\geq Beginn_1 + Dauer_1 \\ Beginn_1 + (1 - Bool) \times Max &\geq Beginn_2 + Dauer_2 \end{aligned}$$

Das Problem der Disjunktion wird somit auf den Constraint-Solver übertragen. Genau dann, wenn eine Ausgangs-Ungleichung gilt, existiert eine eindeutige Bele-

gung der Variable *Bool*. Zur Verdeutlichung dieser bidirektionalen Funktionsweise nachfolgende Fallunterscheidung:

- Lösung über Arbeitsaufgaben initiiert:
  - **Arbeitsaufgabe 1 vor Arbeitsaufgabe 2** bzw. $Beginn_2 \geq Beginn_1 + Dauer_1$ erweist sich als wahr:

    Zur Wahrung der Konsistenz des Constraints $Beginn_1 + (1 - Bool) \times Max \geq Beginn_2 + Dauer_2$ muß die Variable *Bool* mit 0 belegt werden:

    1. Im Falle $Bool = 0$:
       $Beginn_1 + (1 - 0) \times Max \geq Beginn_2 + Dauer_2$
       $\Rightarrow$ $Beginn_1 + Max \geq Beginn_2 + Dauer_2$
       $\Rightarrow$ wahr
    2. Im Falle $Bool = 1$:
       $Beginn_1 + (1 - 1) \times Max \geq Beginn_2 + Dauer_2$
       $\Rightarrow$ $Beginn_1 \geq Beginn_2 + Dauer_2$
       $\Rightarrow$ Widerspruch
  - **Arbeitsaufgabe 2 vor Arbeitsaufgabe 1** bzw. $Beginn_1 \geq Beginn_2 + Dauer_2$ erweist sich als wahr:

    Zur Wahrung der Konsistenz des Constraints $Beginn_2 + Bool \times Max \geq Beginn_1 + Dauer_1$ muß die Variable *Bool* mit 1 belegt werden:

    1. Im Falle $Bool = 0$:
       $Beginn_2 + 0 \times Max \geq Beginn_1 + Dauer_1$
       $\Rightarrow$ $Beginn_2 \geq Beginn_1 + Dauer_1$
       $\Rightarrow$ Widerspruch
    2. Im Falle $Bool = 1$:
       $Beginn_2 + 1 \times Max \geq Beginn_1 + Dauer_1$
       $\Rightarrow$ $Beginn_2 + Max \geq Beginn_1 + Dauer_1$
       $\Rightarrow$ wahr

- Lösung über Schaltvariable initiiert:
  - **Bool = 1** erweist sich als wahr:

    $Beginn_2 + 1 \times Max \geq Beginn_1 + Dauer_1$
    $\Rightarrow$ $Beginn_2 + Max \geq Beginn_1 + Dauer_1$
    $\Rightarrow$ trivial erfüllt

    $Beginn_1 + (1 - 1) \times Max \geq Beginn_2 + Dauer_2$
    $\Rightarrow$ $Beginn_1 \geq Beginn_2 + Dauer_2$
    $\Rightarrow$ Arbeitsaufgabe 2 vor Arbeitsaufgabe 1
  - **Bool = 0** erweist sich als wahr:

$Beginn_2 + 0 \times Max \geq Beginn_1 + Dauer_1$
$\Rightarrow$ $Beginn_2 \geq Beginn_1 + Dauer_1$
$\Rightarrow$ Arbeitsaufgabe 1 vor Arbeitsaufgabe 2

$Beginn_1 + (1 - 0) \times Max \geq Beginn_2 + Dauer_2$
$\Rightarrow$ $Beginn_1 + Max \geq Beginn_2 + Dauer_2$
$\Rightarrow$ trivial erfüllt

2.1.2 Verallgemeinerung

Der Ansatz läßt sich für eine Ungleichung $L \geq R$ verallgemeinern. L und R stellen dabei beliebige Linearkombinationen dar. Statt der Disjunktion der Bearbeitungsreihenfolgen gilt hier:

$$L \geq R \quad \vee \quad L < R$$

Das Prinzip des Schalters $Bool$ kann auf die gleiche Weise wie im vorigen Abschnitt angewendet werden. Er soll genau dann einen Wert von Eins haben, wenn die ursprüngliche Ungleichung $L \geq R$ erfüllt ist. Das allgemeine Maximum Max ist durch die Maxima der linken und rechten Seite L_{max} und R_{max} präzisiert:

$$L + R_{max} \geq R + Bool \times R_{max} \tag{2.1}$$
$$L < R + Bool \times (L_{max} + 1) \tag{2.2}$$

Für eine Ungleichung $L > R$ ist eine Anpassung einfach durchführbar:

$$L + R_{max} + 1 > R + Bool \times (R_{max} + 1) \tag{2.3}$$
$$L \leq R + Bool \times L_{max} \tag{2.4}$$

Eine Gleichung $L = R$ kann durch die Beziehung $L \geq R \ \wedge \ R \geq L$ ausgedrückt werden. Die Formeln 2.1 und 2.2 kommen hier zur Anwendung. Allerdings benötigen beide Teile unterschiedliche Schaltvariablen $Bool^1$ und $Bool^2$, welche anschließend zur eigentlichen Schaltvariable $Bool$ kombiniert werden:

$$L + R_{max} \geq R + Bool^1 \times R_{max} \tag{2.5}$$
$$L < R + Bool^1 \times (L_{max} + 1) \tag{2.6}$$
$$R + L_{max} \geq L + Bool^2 \times L_{max} \tag{2.7}$$
$$R < L + Bool^2 \times (R_{max} + 1) \tag{2.8}$$
$$Bool + 1 = Bool^1 + Bool^2 \tag{2.9}$$

Bei der Formulierung einer Ungleichung $L \neq R$ können die Formeln 2.5 bis 2.8 unverändert übernommen werden. Nur die Ableitung der eigentlichen Schaltvariable $Bool$ von $Bool^1$ und $Bool^2$ ändert sich:

$$\begin{aligned} L + R_{max} &\geq R + Bool^1 \times R_{max} && (2.10)\\ L &< R + Bool^1 \times (L_{max} + 1) && (2.11)\\ R + L_{max} &\geq L + Bool^2 \times L_{max} && (2.12)\\ R &< L + Bool^2 \times (R_{max} + 1) && (2.13)\\ Bool + Bool^1 + Bool^2 &= 2 && (2.14) \end{aligned}$$

2.2 Mehrdeutige Schaltvariablen

In vielen Fällen reicht es aus, wenn die Schaltvariablen nur im Fall der Gültigkeit oder Ungültigkeit einen eindeutigen Wert annehmen. Im weiteren wird deshalb davon ausgegangen, daß nur die **Ungültigkeit** eines Constraints von Interesse ist.

2.2.1 Reduziertes Absetzen

Für die Darstellung einer Ungleichung $L \geq R$ bzw. $L > R$ reicht dann das Absetzen des jeweils ersten Constraints. Entsprechendes gilt für die darauf aufsetzenden Relationen $L = R$ und $L \neq R$:

- $L \geq R$:

$$L + R_{max} \geq R + Bool \times R_{max} \quad (2.15)$$

- $L > R$:

$$L + R_{max} + 1 > R + Bool \times (R_{max} + 1) \quad (2.16)$$

- $L = R$:

$$\begin{aligned} L + R_{max} &\geq R + Bool^1 \times R_{max} && (2.17)\\ R + L_{max} &\geq L + Bool^2 \times L_{max} && (2.18)\\ Bool + 1 &= Bool^1 + Bool^2 && (2.19) \end{aligned}$$

- $L \neq R$:

$$\begin{aligned} L + R_{max} &\geq R + Bool^1 \times R_{max} && (2.20)\\ R + L_{max} &\geq L + Bool^2 \times L_{max} && (2.21)\\ Bool + Bool^1 + Bool^2 &= 2 && (2.22) \end{aligned}$$

2.2.2 Grafischer Ansatz

Bei einer speziellen Struktur, bei der eine linke Seite L mit einer Konstanten r gleichgesetzt wird[1], kann die Konstruktion der Schaltvariablen verbessert werden.

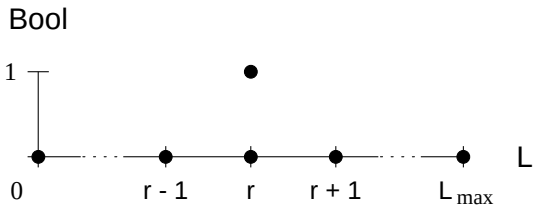


Abbildung 2.1: Abhängigkeit der Schaltvariable *Bool*

Da die möglichen Lösungen einen konvexen Raum bilden, wäre es einfach, das Verhalten der Schaltvariablen über zwei weitere Constraints zu beschreiben, die die ungültigen oberen Bereiche in Abbildung 2.1 abschneiden. Doch bei einem Constraint-Solver, der auf dem Konzept der Propagation von Domäneneinschränkungen basiert, würde eine derartige Modellierung fehlschlagen. Wenn die Domäne der linken Seite L z.B. auf Werte kleiner r eingeschränkt würde, hätte dies keine einschränkenden Auswirkungen auf die zulässigen Werte der Schaltvariablen *Bool*.

Deshalb werden zwei weitere binäre Hilfsvariablen $Bool^1$ und $Bool^2$ eingeführt, die der Abbildung 2.2 entsprechende Ausprägungen haben.

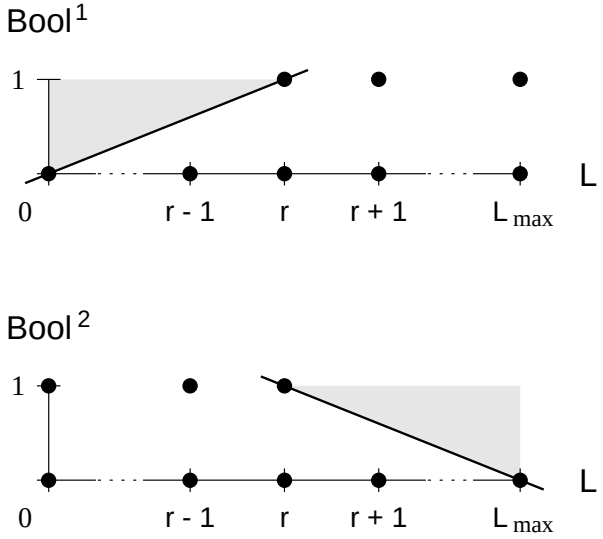


Abbildung 2.2: Verhalten der Hilfsvariablen $Bool^1$ und $Bool^2$

[1] Diese Struktur ist in einigen späteren Abschnitten von Interesse.

Die Kombination zur eigentlichen Schaltvariablen erfolgt wie in den vorigen Abschnitten:

$$Bool + 1 \quad = \quad Bool^1 + Bool^2 \tag{2.23}$$

Für die Hilfsvariablen müssen die Wertebereiche oberhalb der in den Abbildungen dargestellten Trenngeraden aus dem Wertebereich entfernt werden. Dies geschieht durch die folgenden Bedingungen:

$$Bool^1 \times r \quad \leq \quad L \tag{2.24}$$

$$Bool^2 \times (L_{max} - r) + L \quad \leq \quad L_{max} \tag{2.25}$$

Die Terme, mit denen die Variablen $Bool^1$ und $Bool^2$ multipliziert werden, können in Ausnahmefällen einen Wert von Null annehmen. Dies geschieht in den Fällen senkrechter Trenngeraden. In einer derartigen Situation lassen sich die Formeln 2.23 bis 2.25 wesentlich vereinfachen. Gilt $r = 0$, dann reicht das Absetzen des folgenden Constraints mit nur noch einer Schaltvariablen aus:

$$Bool \times L_{max} + L \quad \leq \quad L_{max} \tag{2.26}$$

Bei $L_{max} = r$ verbleibt ebenfalls nur ein Constraint:

$$Bool \times L_{max} \quad \leq \quad L \tag{2.27}$$

Insgesamt führt der grafische Ansatz zu einem ähnlichen Ergebnis, wie die Verarbeitung von $L = R$ im vorigen Abschnitt. Es ergibt sich allerdings der Vorteil, daß bei den geschilderten Ausnahmen die Formeln erheblich vereinfacht werden können. Beim Verfahren des vorigen Abschnittes ist dies nicht möglich.

Kapitel 3

Explizite Alternativen

In diesem Kapitel werden Algorithmen vorgestellt, die die Verarbeitung expliziter Alternativen ermöglichen. Zum besseren Verständnis der Algorithmen ist in Abschnitt 3.1 zunächst eine spezielle Problemstellung dargestellt, die bei der Einführung der Algorithmen beispielhaft behandelt wird.

Abschnitt 3.2 beschäftigt sich mit Möglichkeiten der rein Logischen Programmierung. Die weiteren Methoden der Abschnitte 3.4, 3.5 und 3.6 arbeiten dagegen mit Mitteln der Constraint-Programmierung. Die constraint-basierten Methoden benötigen eine binäre Darstellung des Problems, weshalb Abschnitt 3.3 zunächst die Transformation in eine binäre Darstellung behandelt.

Ein Vergleich der Algorithmen wird in Abschnitt 3.7 durchgeführt.

3.1 Problemstellung

Zur Verdeutlichung der Funktionsweise der Verfahren werden diese anhand einer konkreten Problemstellung vorgestellt.

Dabei soll im Rahmen einer Untersuchung zur Durchführbarkeit eines Auftrags ein Transportunternehmen zur Auslieferung einer Ware bestimmt werden. Es stehen die fünf Unternehmen U_1 bis U_5 zur Auswahl.

Da hierfür die Einbeziehung quantitativer und qualitativer Kriterien nötig ist, bietet sich die Anwendung eines Punktbewertungsverfahrens an (vgl. [Bruhn90]). Die Unternehmen werden zunächst in den Kategorien *Relevante Kosten* R, *Transportzeit* T, *Qualitative Eignung* Q und *Zuverlässigkeit* Z mit null bis zehn Punkten bewertet:

	Relevante Kosten R	Transportzeit T	Qualitative Eignung Q	Zuverlässigkeit Z
U_1	7	5	8	7
U_2	6	7	6	9
U_3	4	8	4	8
U_4	8	3	1	7
U_5	8	9	1	8

Tabelle 3.1: Punktwertungen der Unternehmen

Die Auswahl eines Unternehmens ist gleichbedeutend mit der Belegung der Variablen R, T, Q und Z durch die Werte der entsprechenden Zeile. Die Zeilen der Tabelle stellen somit explizite Alternativen dar.

3.2 Vollständige Enumeration

Mit den Mechanismen der Logischen Programmierung läßt sich die Suche nach der optimalen Lösung auch ohne Anwendung von Constraint-Techniken relativ einfach bewerkstelligen. Bei einem derartigen Vorgehen müssen allerdings **sämtliche** Lösungen einzeln untersucht werden.

3.2.1 Umsetzung durch Iteration

Durch die Logische Programmierung kann beispielsweise das Verfahren der Iteration realisiert werden. Zur Nutzung dieses Verfahrens werden die verschiedenen Belegungsalternativen zunächst als Fakten abgelegt. Für die in Abschnitt 3.1 dargestellte Problembeschreibung kann dies folgendermaßen geschehen:

```
alternative([7, 5, 8, 7]).
alternative([6, 7, 6, 9]).
alternative([4, 8, 4, 8]).
alternative([8, 3, 1, 7]).
alternative([8, 9, 1, 8]).
```

Über den Mechanismus der Unifikation kann den Entscheidungsvariablen dann eine Belegung zugewiesen werden. Bei einem Optimierungs-Problem reicht dies jedoch nicht aus, denn unter den gültigen Lösungen muß die **optimale** (bezüglich eines Optimierungskriteriums) bestimmt werden.

Nach der Auswertung einer Belegung kann durch ein `fail` ein Backtracking zur nächsten Alternative erzwungen werden. Dieser iterative Prozeß wird so lange durchgeführt, bis alle Alternativen geprüft sind. Innerhalb des Suchprozesses wird jede gültige Lösung mit der bisherig besten verglichen, und bei Überlegenheit als neue beste Lösung weitergeführt.

Während der Suche muß sich die derzeit beste Lösung gemerkt werden. Da das Backtracking aber den Gesamtzustand inklusive der Infomation über die beste Lösung zurücksetzt, muß dies als Seiteneffekt über den *assert/retract*-Mechanismus realisiert werden.

Mit einer Bewertungsfunktion `bewerte_alternative` und einer Funktion zum Vergleich dieser Bewertungen `besser_als` geschieht eine iterative Optimierung folgendermaßen:

```
loesung_aktualisieren(BesteAlternative, BesteBewertung) :-
```

```
      retract(bisher_beste_loesung(_, _)),
      assert(bisher_beste_loesung(BesteAlternative,
        BesteBewertung)),

      !.


optimiere(AktuelleAlternative) :-

      assert(bisher_beste_loesung(dummy, dummy)),

      alternative(AktuelleAlternative),
      bewerte_alternative(AktuelleAlternative, AktuelleBewertung),

      bisher_beste_loesung(_, BesteBewertung),

      (besser_als(AktuelleBewertung, BesteBewertung) ->

        loesung_aktualisieren(AktuelleAlternative,
          AktuelleBewertung);

        fail),

      fail.

optimiere(OptimaleAlternative) :-

      bisher_beste_loesung(OptimaleAlternative, _).


:- optimiere([R, T, Q, Z]).
```

Zum besseren Verständnis sind nachfolgend Auszüge eines Tracings des Programmablaufes dargestellt. Als Bewertungsfunktion (`bewerte_alternative`) wurde die Summation der Werte einer Alternative und als Vergleich von Bewertungen (`besser_als`) die >-Funktion verwendet:

```
CALL   optimiere([R, T, Q, Z])
EXIT   assert(bisher_beste_loesung(dummy, dummy))
CALL   alternative([R, T, Q, Z])
EXIT   alternative([7, 5, 8, 7])
EXIT   bewerte_alternative([7, 5, 8, 7], 27)
```

```
EXIT   bisher_beste_loesung(dummy, dummy)
EXIT   besser_als(27, dummy)
CALL   loesung_aktualisieren([7, 5, 8, 7], 27)
EXIT   retract(bisher_beste_loesung(dummy, dummy))
EXIT   assert(bisher_beste_loesung([7, 5, 8, 7], 27))
EXIT   loesung_aktualisieren([7, 5, 8, 7], 27)
FAIL   fail
RETRY  alternative([R, T, Q, Z])
EXIT   alternative([6, 7, 6, 9])
EXIT   bewerte_alternative([6, 7, 6, 9], 28)
EXIT   bisher_beste_loesung([7, 5, 8, 7], 27)
EXIT   besser_als(28, 27)
CALL   loesung_aktualisieren([6, 7, 6, 9], 28)
EXIT   retract(bisher_beste_loesung([7, 5, 8, 7], 27))
EXIT   assert(bisher_beste_loesung([6, 7, 6, 9], 28))
EXIT   loesung_aktualisieren([6, 7, 6, 9], 28)
FAIL   fail
RETRY  alternative([R, T, Q, Z])
EXIT   alternative([4, 8, 4, 8])
EXIT   bewerte_alternative([4, 8, 4, 8], 24)
EXIT   bisher_beste_loesung([6, 7, 6, 9], 28)
FAIL   besser_als(24, 28)
FAIL   fail
RETRY  alternative([R, T, Q, Z])
EXIT   alternative([8, 3, 1, 7])
EXIT   bewerte_alternative([8, 3, 1, 7], 19)
EXIT   bisher_beste_loesung([6, 7, 6, 9], 28)
FAIL   besser_als(19, 28)
FAIL   fail
RETRY  alternative([R, T, Q, Z])
EXIT   alternative([8, 9, 1, 8])
EXIT   bewerte_alternative([8, 9, 1, 8], 26)
EXIT   bisher_beste_loesung([6, 7, 6, 9], 28)
FAIL   besser_als(26, 28)
FAIL   fail
RETRY  optimiere([R, T, Q, Z])
EXIT   bisher_beste_loesung([6, 7, 6, 9], 28)
EXIT   optimiere([6, 7, 6, 9])

yes
```

In der Regel wird die Auswahl einer Alternative nicht die einzig zu optimierende Aufgabe sein. Bei der eingeführten Untersuchung zur Durchführbarkeit eines

Auftrags (Abschnitt 3.1) kann beispielsweise ebenfalls eine Entscheidung über die genaue Produktgestaltung zu fällen sein.

Die zu treffenden Entscheidungen sind voneinander abhängig, da eine andere Produktgestaltung schnellere Transportzeiten oder eine bessere qualitative Eignung des Transportunternehmens bedingen kann. Die Entscheidungen dürfen dann nicht lokal optimiert werden, da dies ein **globales Optimum** verhindern kann.

Die Einzelalternativen werden deshalb zu einer Gesamtalternative zusammengefaßt, welche zu optimieren ist. Die dargestellte Funktion zur Optimierung kann erhalten bleiben, das Prädikat `alternative` muß nun allerdings statt einer Transportalternative eine Gesamtalternative liefern, in der eine Transportalternative mit weiteren Alternativen, wie z.B. der Auswahl der Produktgestaltung, gekoppelt ist:

```
alternative([Transportalternative|WeitereAlternativen]) :-

      transportalternative(Transportalternative),

      weitere_alternativen(WeitereAlternativen).
```

3.2.2 Umsetzung durch Rekursion

Statt einem Durchlaufen der Alternativen mittels Iteration kann der Mechanismus der Rekursion verwendet werden. Die noch nicht bearbeiteten Alternativen werden zusammen mit der bisher besten Alternative und deren Bewertung jeweils rekursiv übergeben. Über eine weitere Variable im Regelkopf wird die optimale Alternative am Ende zurückgegeben.

```
optimiere([AktuelleAlternative|RestlicheAlternativen],
        BisherBesteAlternative, BisherBesteBewertung,
        OptimaleAlternative) :-

      bewerte_alternative(AktuelleAlternative, AktuelleBewertung),

      bessere_alternative(AktuelleAlternative, AktuelleBewertung,
        BisherBesteAlternative, BisherBesteBewertung,
        JetztBesteAlternative, JetztBesteBewertung),

      optimiere(RestlicheAlternativen, JetztBesteAlternative,
        JetztBesteBewertung, OptimaleAlternative).

optimiere([], OptimaleAlternative, _, OptimaleAlternative).
```

Der Ansatz der Rekursion erscheint zunächst eleganter, da keine Seiteneffekte auftreten. Gehen neben der Auswahl des Transportunternehmens jedoch noch **weitere Entscheidungen** in die Optimierung ein, wie z.B. die Auswahl der genauen Produktgestaltung, so bereitet der Ansatz über Rekursion erhebliche Probleme.

In jedem Rekursionsschritt müssen nun alle noch nicht bearbeiteten Gesamtalternativen übergeben werden. Die Anzahl der Gesamtalternativen berechnet sich aber aus der Multiplikation der Anzahlen der Einzelalternativen. Selbst bei kleineren Problemstellungen ergibt sich somit schnell ein unbeherrschbarer Verarbeitungsaufwand. Bei der Umsetzung durch Iteration wird dagegen jeweils nur eine einzige Gesamtalternative betrachtet.

Ein weiteres Problem stellen **Einschränkungen durch den Kontext** dar. Beispielsweise könnte sich aus Nebenbedingungen ergeben, daß die Bewertung der relevanten Kosten mindestens acht Punkte betragen muß. Die Wahl des unter dem Optimierungskriterium der höchsten Summe von Punkten als optimal bezeichneten zweiten Transportunternehmens (Alternative [6, 7, 6, 9]) wäre dann ungültig.

Das Vorgehen der Rekursion bedingt, daß die Unifikation einer Belegung mit den Entscheidungsvariablen erst nach dem Finden des Optimums vorgenommen werden kann[1]. Mit diesen Variablen verknüpfte Nebenbedingungen können dann erst greifen. Bei einer daraus entstehenden Inkonsistenz kann dies ein Backtracking der gesamten Optimierung bewirken, was den Aufwand des Verfahrens erheblich steigert. Bei der im vorigen Abschnitt vorgeschlagenene Umsetzung durch Iteration erfolgt die Unifikation direkt mit der Wahl einer Gesamtalternative, was ein sofortiges Ausscheiden von ungültigen Lösung bewirkt.

Trotz der auf den ersten Blick deklarativeren Methode der Rekursion ist deshalb der Ansatz der Iteration vorzuziehen.

[1] Andere Lösungen sind denkbar, würden aber den deklarativen Charakter der Rekusions-Lösung beeinträchtigen.

3.3 Transformation in eine binäre Darstellung

Die in den folgenden Abschnitten vorgestellten Algorithmen basieren auf Alternativen binärer Variablen[2]. Da diese Technik im Operations Research sehr verbreitet ist, existieren bereits mannigfaltige Verfahren zur Transformation und Kompression (siehe z.B. [Syslo83]). Dazu werden die natürlichen Variablen zunächst auf eine Linearkombination binärer Variablen abgebildet:

$$n = \sum_{i=0}^{m} 2^i b_i \tag{3.1}$$

Hierdurch werden viele eigentlich unnötige Variablen erzeugt. Mit Verfahren zur Elimination von Variablen kann die Menge der Variablen reduziert werden. Dabei ist darauf zu achten, daß die Abbildung auf die ursprüngliche Zahl n linearer Natur sein muß. Abbildung 3.1 zeigt ein Beispiel für die Transformation der relevanten Kosten:

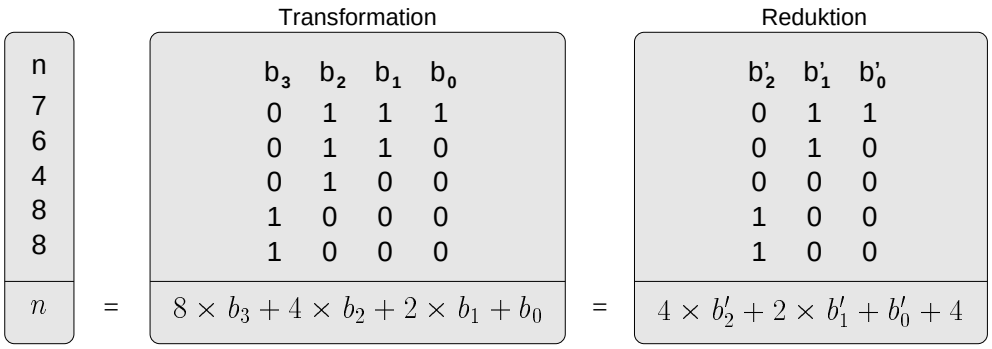


Abbildung 3.1: Binäre Darstellung der relevanten Kosten

Im folgenden sind einige einfache Regeln zur Durchführung der Reduktion aufgeführt (Regel 4 wurde zur Reduktion in Abbildung 3.1 verwendet):

1. **Alle Elemente einer Spalte sind null**:
   Die Ausprägung der zugehörigen binären Variable hat keinen Einfluß auf den Wert der Zielvariable n. Die Variable kann somit entfallen.

2. **Alle Elemente einer Spalte sind eins**:
   Der Koeffizient der zugehörigen Variable geht in jedem Fall in den Wert der Zielvariable ein. Statt der Variable kann deshalb eine Konstante verwendet werden.

[2]Die Einschränkung auf diesen Wertebereich läßt die Anwendung speziell optimierter Mechanismen zu (siehe z.B. [Barth96]).

3. **Gleiche Spalten**:
   Die Koeffizienten der zugehörigen Variablen gehen in denselben Alternativen in den Wert der Zielvariable ein. Es ist deshalb die Verwendung nur einer Variable ausreichend, deren Koeffizient gleich der Summe der Koeffizienten der betreffenden Spalten ist.

4. **Einsen mehrerer Spalten ergänzen sich zu einer Spalte mit ausschließlich Einsen**:
   Die Variable mit dem kleinsten Koeffizienten kann entfallen, der Koeffizient wird in eine Konstante umgewandelt. Von den Koeffizienten der anderen betroffenen Spalten wird diese Konstante abgezogen.

5. **Variablen mit gleichen Koeffizienten**:
   Treten die Einsen der zugehörigen Spalten in unterschiedlichen Alternativen auf, ist die Verwendung nur einer Variable ausreichend. In die zugehörige Spalte werden alle Einsen der betreffenden Spalten übernommen. Der Koeffizient bleibt unverändert.

Entsprechend werden auch die anderen Punktwertungen umgesetzt. Das Resultat bildet eine aus binären Werten bestehende Alternativen-Tabelle:

	Rel. Kosten			Transportzeit				Qual. Eignung			Zuverl.	
	B^1	B^2	B^3	B^4	B^5	B^6	B^7	B^8	B^9	B^{10}	B^{11}	B^{12}
A_1	0	1	1	0	1	0	0	1	0	0	0	0
A_2	0	1	0	0	1	1	0	0	1	0	1	1
A_3	0	0	0	1	0	0	0	0	0	0	0	1
A_4	1	0	0	0	0	0	1	0	0	1	0	0
A_5	1	0	0	1	0	0	1	0	0	1	0	1

Tabelle 3.2: Alternativen-Tabelle der binär transformierten Punktwertungen

Dabei gelten die folgenden Zusammenhänge:

$$\begin{aligned} \text{Relevante Kosten} &= 4 \times B^1 + 2 \times B^2 + B^3 + 4 \\ \text{Transportzeit} &= 6 \times B^4 + 3 \times B^5 + 2 \times B^6 + B^7 + 2 \\ \text{Qualitative Eignung} &= 4 \times B^8 + 2 \times B^9 + B^{10} + 4 \\ \text{Zuverlässigkeit} &= B^{11} + B^{12} + 7 \end{aligned}$$

Die Menge von Variablen B besteht nun aus den 12 Variablen B^1 bis B^{12}. Ein Element der Alternativen-Tabelle von Variable (bzw. Spalte) i und Alternative (bzw. Zeile) a wird im weiteren durch B^i_a referenziert.

3.4 Umsetzung durch Ungleichungen

Nachdem die Alternativen in einer binären Form vorliegen, kann eine effiziente constraint-basierte Umsetzung erfolgen. Die allgemeine Vorgehensweise dieser Umsetzung zeigt die folgende Grafik:

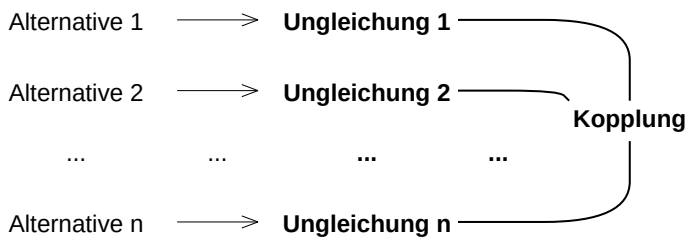


Abbildung 3.2: Vorgehensweise bei einer Umsetzung durch Ungleichungen

Jede Alternative wird zunächst durch eine Ungleichung beschrieben. Das Absetzen einer solchen Ungleichung erzwingt die Belegung der Entscheidungsvariablen mit den Werten der korrespondierenden Alternative. Das gleichzeitige Absetzen aller Ungleichungen ist nun aber nicht möglich, da das Gelten mehrerer Ungleichungen widersprechende Belegungen der Entscheidungsvariablen verlangen würde. Über einen zusätzlichen Mechanismus werden die Ungleichungen deshalb so gekoppelt, daß nur genau eine von ihnen gültig ist.

3.4.1 Umsetzung einer Alternative

Dieser Abschnitt erläutert die Umsetzung einer Alternative in eine Ungleichung. Für die erste Alternative des Beispiels müssen den Variablen B^1 bis B^{12} die Werte B_1^1 bis B_1^{12} zugewiesen werden.

	B^1	B^2	B^3	B^4	B^5	B^6	B^7	B^8	B^9	B^{10}	B^{11}	B^{12}
A_1	0	1	1	0	1	0	0	1	0	0	0	0

Die Summe aller mit Null zu belegenden Variablen soll Null betragen:

$$B^1 + B^4 + B^6 + B^7 + B^9 + B^{10} + B^{11} + B^{12} \quad = \quad 0$$

Ebenso kann die Summe der Variablen vorgegeben werden, denen eine Eins zugewiesen werden soll:

$$B^2 + B^3 + B^5 + B^8 \quad = \quad 4$$

Die beiden Gleichungen können durch Addition zu einer einzigen zusammengefaßt werden. Durch die einfach Addition ergibt sich:

$$\begin{aligned} B^1 + B^4 + B^6 + B^7 + B^9 + B^{10} + B^{11} + B^{12} &= 0 \\ B^2 + B^3 + B^5 + B^8 &= 4 \\ \hline B^1 + B^2 + B^3 + B^4 + B^5 + B^6 + B^7 + B^8 + B^9 + B^{10} + B^{11} + B^{12} &= 4 \end{aligned}$$

Die Zuordnung der Belegungen ist nicht mehr eindeutig. Jede Belegung, bei der vier Variablen eine Eins erhalten, ist zulässig. Wird vor der Addition bei einer Gleichung die linke Seite mit der rechten vertauscht, ergibt sich jedoch eine eindeutige Lösung:

$$\begin{aligned} 0 &= B^1 + B^4 + B^6 + B^7 + B^9 + B^{10} + B^{11} + B^{12} \\ B^2 + B^3 + B^5 + B^8 &= 4 \\ \hline B^2 + B^3 + B^5 + B^8 &= 4 + B^1 + B^4 + B^6 + B^7 + B^9 + B^{10} + B^{11} + B^{12} \end{aligned}$$

Die Konstante der rechten Seite muß auf jeden Fall durch die linke Seite abgedeckt werden. Dies kann nur erreicht werden, wenn allen Variablen der linken Seite eine Eins zugewiesen wird. Da die rechte Seite den Wert der linken Seite nicht überschreiten darf, müssen die Variablen der rechten Seite mit Null belegt werden.

Zur effizienten Kopplung wird allerdings statt der Gleichung eine Ungleichung benötigt. Die Umwandlung in eine Ungleichung kann ohne Verlust der Eindeutigkeit vollzogen werden:

$$B^2 + B^3 + B^5 + B^8 \geq 4 + B^1 + B^4 + B^6 + B^7 + B^9 + B^{10} + B^{11} + B^{12}$$

Allgemein geschieht die Umsetzung einer Alternative a folgendermaßen:

$$\sum_{B_a^i=1} B^i \geq \sum B_a^i + \sum_{B_a^i=0} B^i \tag{3.2}$$

Dies ist für sämtliche Alternativen durchzuführen.

3.4.2 Kopplung durch Schaltvariablen

Wie bereits erwähnt würde das Gelten aller Ungleichungen eine Inkonsistenz ergeben, da jede Ungleichung eine andere Belegung der Variablen voraussetzt.

Deshalb wird jede Ungleichung mit einer Schaltvariable verbunden, die angibt, ob die jeweilige Ungleichung erfüllt ist. Dafür wird der in Abschnitt 2.1.2 vorgestellte Ansatz verwendet. Die durch Formel 3.2 gebildete Ungleichung einer Alternative a wird zunächst aufgespalten:

$$\begin{aligned} L_a &= \sum_{B_a^i=1} B^i \\ R_a &= \sum B_a^i + \sum_{B_a^i=0} B^i \end{aligned}$$

Der Wert einer linken Seite L kann maximal gleich der Anzahl der Entscheidungsvariablen B^i sein. Der Wert einer rechten Seite R erreicht maximal die verdoppelte Anzahl von Entscheidungsvariablen. Es lassen sich zwar niedrigere Schranken für diese Werte finden, doch diese Maxima sind ohne größeren Aufwand ermittelbar:

$$\begin{aligned} L_{max} &= |B^i| \\ R_{max} &= 2 \times |B^i| \end{aligned}$$

In Weiterführung des Beispiels gilt somit $L_{max} = 12$ und $R_{max} = 24$.

Unter Verwendung von S_a als Schaltvariable führt die Anwendung der Formeln 2.1 und 2.2 auf $L_a \geq R_a$ zur Aufstellung folgender Constraints:

$$\sum_{B_a^i=1} B^i + 2 \times |B^i| \;\geq\; \sum B_a^i + \sum_{B_a^i=0} B^i + S_a \times 2 \times |B^i| \tag{3.3}$$

$$\sum_{B_a^i=1} B^i \;<\; \sum B_a^i + \sum_{B_a^i=0} B^i + S_a \times \Big(|B^i| + 1\Big) \tag{3.4}$$

Für die erste Alternative des Beispiels ergibt sich:

$$B^2 + B^3 + B^5 + B^8 + 20 \;\geq$$
$$B^1 + B^4 + B^6 + B^7 + B^9 + B^{10} + B^{11} + B^{12} + 24 \times S_1$$

$$B^2 + B^3 + B^5 + B^8 \;<$$
$$4 + B^1 + B^4 + B^6 + B^7 + B^9 + B^{10} + B^{11} + B^{12} + 13 \times S_1$$

Nachdem Schaltvariablen für alle Ungleichungen erzeugt wurden, kann die Gültigkeit genau einer Alternative durch das folgende Constraint gesichert werden:

$$\sum_a S_a \quad = \quad 1 \tag{3.5}$$

Diese Gleichung eröffnet die Möglichkeit, mehrdeutige Schaltvariablen zu verwenden. Bei Ungültigkeit nehmen die mehrdeutigen Schaltvariablen einen Wert von Null an, nur der Fall der Gültigkeit bleibt ungeklärt. Sind alle Schaltvariablen bis auf eine mit Null belegt, so wird für diese letzte Schaltvariable wegen Gleichung 3.5 ein eindeutiger Wert von Eins erzwungen. Für die Repräsentation der erste Alternative des Beispiels reicht deshalb folgendes Constraint[3]:

$$B^2 + B^3 + B^5 + B^8 + 20 \;\geq$$
$$B^1 + B^4 + B^6 + B^7 + B^9 + B^{10} + B^{11} + B^{12} + 24 \times S_1$$

[3]Ein Vergleich der Laufzeiten beider Versionen zeigt deutliche Laufzeitvorteile zugunsten mehrdeutiger Schaltvariablen.

3.5 Umsetzung durch Gleichungen

Die grobe Vorgehensweise bei einer Umsetzung durch Gleichungen ähnelt der des Abschnittes 3.4:

Abbildung 3.3: Vorgehensweise bei einer Umsetzung durch Gleichungen

Die Alternative werden nun durch Gleichungen statt Ungleichungen beschrieben. Ebenso wie im vorigen Abschnitt bewirkt das Gelten einer solchen Gleichung die Belegung der Entscheidungsvariablen mit den Werten der korrespondierenden Alternative. Da das gleichzeitige Absetzen aller Gleichungen nicht möglich ist, muß anschließend ebenfalls eine Kopplung durchgeführt werden.

3.5.1 Umsetzung einer Alternative

Theoretisch wäre eine Umsetzung entsprechend des vorigen Abschnittes möglich, bei dem in Formel 3.2 das $\geq$ einfach durch ein $=$ ersetzt wird. Die eindeutige Belegung der Entscheidungsvariablen wird auch hier erzwungen. Im Vergleich zur Umsetzung durch eine Ungleichung schneidet diese Lösung aber schlechter ab, da die Kopplung von Gleichungen einen höheren Aufwand erfordert, als die von Ungleichungen (siehe Abschnitt 2.1.2).

Bei Verwendung einer linearen Gleichung zur Darstellung einer Alternative a können die Entscheidungsvariablen mit Koeffizienten multipliziert, und einer Konstanten gleichgesetzt werden:

$$\sum_{B^i} \left(\alpha_a^i \times B^i \right) \quad = \quad \Sigma_a \tag{3.6}$$

Die Koeffizienten α_a^i und die Konstante Σ_a müssen im folgenden so bestimmt werden, daß Gleichung 3.6 genau dann gilt, wenn die Variablen B^i die der Alternative entsprechenden Werte annehmen.

Wenn der von der Alternative vorgegebene Wert für eine Variable Null ist, muß die Zuweisung einer Eins unterbunden werde. Dies kann durch die Wahl des zur Variable zugehörigen Koeffizienten geschehen. Er muß einen Wert erhalten,

der größer ist, als die Konstante Σ_a. Eine Aktivierung[4] der Variablen würde die linke Seite der Gleichung bereits größer als die rechte machen, und somit die Gültigkeit verhindern. Voraussetzung dafür ist allerdings, daß alle Koeffizienten Werte größer Null erhalten.

Andererseits muß eine vorgegebene Belegung einer Variablen von Eins durchgesetzt werden. Dazu kann die Kombinationssumme Σ_a mit der Summe der Koeffizienten aller Variablen gleichgesetzt werden, die eine Belegung mit Eins erfordern. Die genauen Werte der Koeffizienten spielen in diesem Fall keine Rolle und können der Einfachheit halber auf Eins gesetzt werden.

Σ_a und die Koeffizienten α_a^i sind somit folgendermaßen zu wählen:

$$\Sigma_a = \sum_{B^i} B_a^i \tag{3.7}$$

$$\alpha_a^i = \left(1 - B_a^i\right) \times \Sigma_a + 1 \tag{3.8}$$

Wird die erste Alternative des Beispiels umgesetzt, so ergibt sich die folgende Gleichung:

$$B^1 + B^4 + B^6 + B^7 + B^9 + B^{10} + B^{11} + B^{12} + 5 \times \left(B^2 + B^3 + B^5 + B^8\right) = 4$$

Im Vergleich zur Umsetzung durch Ungleichungen erreicht diese Lösung eine sehr viel schlechtere Propagation. Wird in Formel 3.2 nur eine Variable falsch belegt, hat dies sofort die Ungültigkeit zur Folge. Bei der hier angebotenen Lösung kann die Ungültigkeit nur bei einer Fehlbelegung von Variablen festgestellt werden, deren vorgegebener Wert eine Null ist, bzw. falls alle Variablen mit vorgegebenen Werten von Eins belegt sind.

3.5.2 Kopplung durch Schaltvariablen

Zur Kopplung der Gleichungen werden wieder Schaltvariablen benutzt. Die Einfügung verläuft entsprechend Abschnitt 2.1.2. Dafür werden zunächst die linke und rechte Seite einer Gleichung getrennt:

$$L_a = \sum_{B^i} \left(\alpha_a^i \times B^i\right)$$

$$R_a = \Sigma_a$$

Das Maximum der linken Seite entspricht der Summe aller Koeffizienten, die rechte Seite ist bereits eine Konstante:

$$L_{max} = \sum_{B^i} \alpha_a^i$$

[4] Der Begriff der *Aktivierung* wird im folgenden synonym zur Belegung einer Variablen mit Eins verwendet.

$$R_{max} \;=\; \Sigma_a$$

Nun können die Formeln 2.5 bis 2.5 zum Einsatz kommen. Als Basis-Schaltvariablen dienen S_a^1 und S_a^2, welche dann zur eigentlichen Schaltvariable S_a kombiniert werden:

$$\sum_{B^i} \left(\alpha_a^i \times B^i\right) \;\geq\; S_a^1 \times \Sigma_a \tag{3.9}$$

$$\sum_{B^i} \left(\alpha_a^i \times B^i\right) \;<\; \Sigma_a + S_a^1 \times \left(\sum_{B^i} \alpha_a^i + 1\right) \tag{3.10}$$

$$\Sigma_a + \sum_{B^i} \alpha_a^i \;\geq\; \sum_{B^i} \left(\alpha_a^i \times B^i\right) + S_a^2 \times \sum_{B^i} \alpha_a^i \tag{3.11}$$

$$\Sigma_a \;<\; \sum_{B^i} \left(\alpha_a^i \times B^i\right) + S_a^2 \times (\Sigma_a + 1) \tag{3.12}$$

$$S_a + 1 \;=\; S_a^1 + S_a^2 \tag{3.13}$$

Die Kopplung der Schaltvariablen verläuft analog zu Formel 3.5. Entsprechend reicht die Verwendung mehrdeutiger Schaltvariablen aus. Formeln 3.10 und 3.12 können somit entfallen[5].

Die Umsetzung der ersten Alternative des Beispiels resultiert in folgenden Constraints:

$$\begin{aligned} B^1 + B^4 + B^6 + B^7 + B^9 + B^{10} + B^{11} + B^{12} + 5 \times (B^2 + B^3 + B^5 + B^8) \\ \geq\; 4 \times S_1^1 \end{aligned}$$

$$\begin{aligned} 32 \;\geq\; B^1 + B^4 + B^6 + B^7 + B^9 + B^{10} + B^{11} + B^{12} + \\ 5 \times (B^2 + B^3 + B^5 + B^8) + 28 \times S_1^2 \end{aligned}$$

$$S_1 + 1 \;=\; S_1^1 + S_1^2$$

[5]Die Verwendung mehrdeutiger Schaltvariablen nach Abschnitt 2.2.2 wäre allerdings wegen der Vereinfachung in Ausnahmefällen effizienter.

3.6 Umsetzung durch Gruppenbildung

Im Gegensatz zu einer Umsetzung durch Ungleichungen oder Gleichungen werden bei einer Umsetzung durch Gruppenbildung die Alternativen vorerst in Gruppen aufgeteilt. Für jede dieser Gruppen wird eine Gleichung erstellt, deren mögliche Lösungen genau den durch die zugehörigen Alternativen vorgegebenen Belegungen entsprechen. Diese Gleichungen werden anschließend analog zu Abschnitt 3.4.2 gekoppelt.

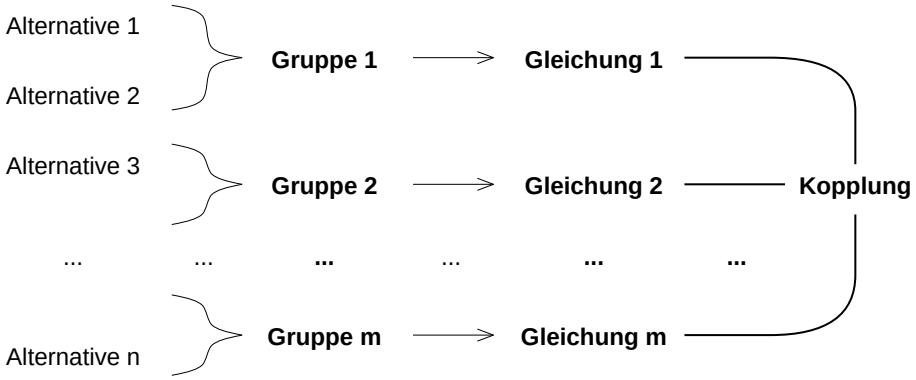


Abbildung 3.4: Vorgehensweise bei einer Umsetzung durch Gruppenbildung

Die Gruppenbildung ermöglicht eine komprimierte Darstellung, d.h. eine Reduktion der benötigten Constraints und Variablen. Es existieren Verfahren zur Aggregation von Gleichungen, die an dieser Stelle für eine Zusammenfassung eingesetzt werden könnten (siehe z.B. [Syslo83]), doch diese sind wegen hoher Laufzeiten und extremem Wachstum der benötigten Koeffizienten praktisch nicht einsetzbar.

3.6.1 Umsetzung einer Gruppe

Im Rahmen des Beispiels können die beiden Gruppen von Alternativen $G_1 = \{A_1, A_2, A_3, A_4\}$ und $G_2 = \{A_5\}$ gebildet werden[6]. Im folgenden wird die Bildung einer linearen Gleichung für eine Gruppe von Alternativen anhand der ersten Gruppe des Beispiels G_1 verdeutlicht.

Den Ausgangspunkt für die Gültigkeit genau einer Alternative einer Gruppe G_g bildet wie im vorigen Abschnitt eine lineare Gleichung:

$$\sum_{B^i} \left(\alpha_g^i \times B^i \right) \quad = \quad \Sigma_g \tag{3.14}$$

[6]Die Restriktionen für die Aufteilung in Gruppen werden später erläutert.

Diese Gleichung muß für die Belegungen aller Alternativen der Gruppe erfüllt sein:

$$\forall a \in G_g \; : \quad \sum_{B^i} \left(\alpha_g^i \times B_a^i \right) \; = \; \Sigma_g$$

Andere Belegungen als die der Alternativen aus G_g dürfen die Gleichung 3.14 nicht erfüllen.

Für die Gleichung der Gruppe G_1 des Beispiels sind die 12 Koeffizienten α_1^1 bis α_1^{12} und die Konstante Σ_1 zu bestimmen.

Das Vorgehen zur Bestimmung dieser Werte gliedert sich in mehrere Schritte (siehe Abbildung 3.5). Zunächst findet eine Vorverarbeitung statt, bei der die Spalten der Tabelle umsortiert und einige abgespalten werden. Die Koeffizienten des einen Teils können anschließend bestimmt werden. Aus diesen Koeffizienten läßt sich die Konstante Σ_g der Gleichung ableiten. Danach ist es möglich, die restlichen Koeffizienten zu bestimmen.

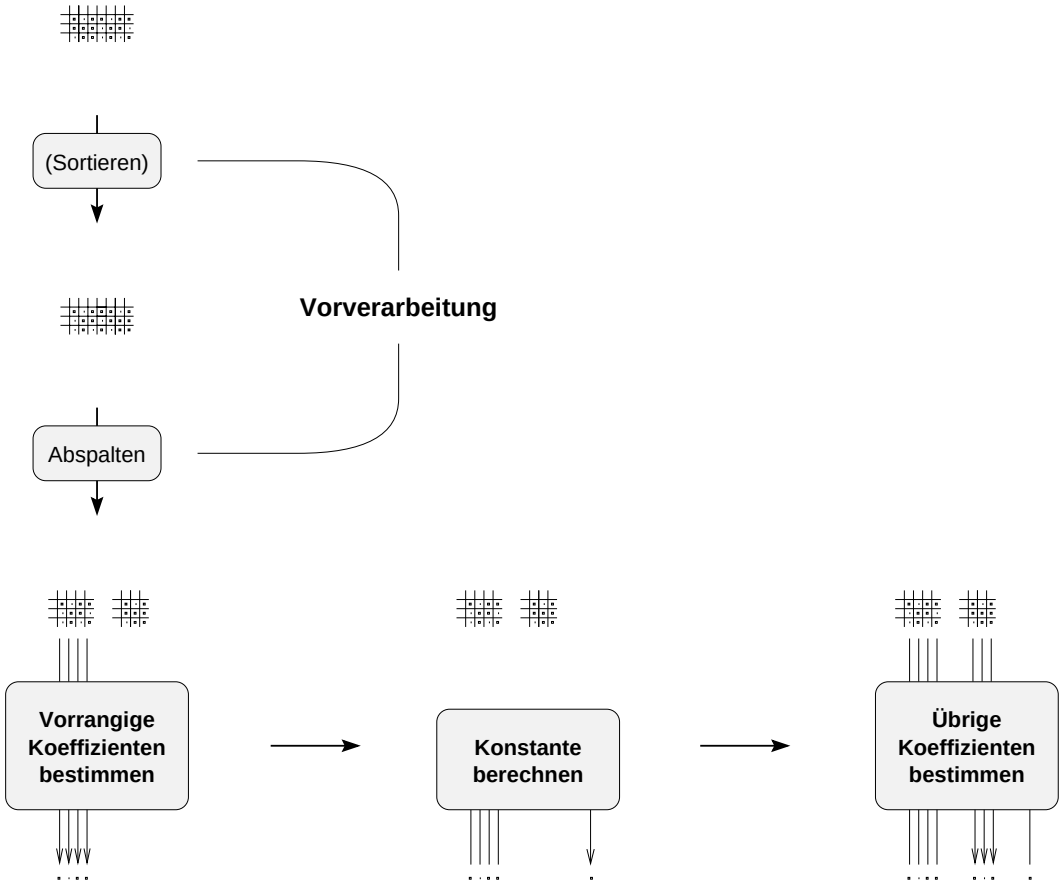


Abbildung 3.5: Umsetzung einer Gruppe

3.6.1.1 Vorverarbeitung

Aus den in der Tabelle der Alternativen vorgegebenen möglichen Werten für die Variablen B^i wird pro Variable und Gruppe G_g ein Vektor $\vec{B}_g^i$ gebildet, der einer Spalte der Tabelle entspricht. Dabei werden allerdings nur die Werte der Alternativen der Gruppe berücksichtigt.

In den beiden folgenden Tabellen sind die Vektoren der Gruppe G_1 des Beispiels aufgeführt. Dabei wird zwischen Vektoren unterschieden, deren Betrag größer oder gleich Eins ist. Gleiche Vektoren werden nebeneinander geordnet. Diese Ordnung ist nicht zwingend, ermöglicht aber eine bessere Übersicht.

	$\vec{B}_1^2$	$\vec{B}_1^5$	$\vec{B}_1^{12}$
A_1	$\begin{pmatrix}1\\1\\0\\0\end{pmatrix}$	$\begin{pmatrix}1\\1\\0\\0\end{pmatrix}$	$\begin{pmatrix}0\\1\\1\\0\end{pmatrix}$
A_2			
A_3			
A_4			

Tabelle 3.3: Vektoren mit einem Betrag größer Eins

	$\vec{B}_1^3$	$\vec{B}_1^8$	$\vec{B}_1^9$	$\vec{B}_1^6$	$\vec{B}_1^{11}$	$\vec{B}_1^4$	$\vec{B}_1^1$	$\vec{B}_1^7$	$\vec{B}_1^{10}$
A_1	$\begin{pmatrix}1\\0\\0\\0\end{pmatrix}$	$\begin{pmatrix}1\\0\\0\\0\end{pmatrix}$	$\begin{pmatrix}0\\1\\0\\0\end{pmatrix}$	$\begin{pmatrix}0\\1\\0\\0\end{pmatrix}$	$\begin{pmatrix}0\\1\\0\\0\end{pmatrix}$	$\begin{pmatrix}0\\0\\1\\0\end{pmatrix}$	$\begin{pmatrix}0\\0\\0\\1\end{pmatrix}$	$\begin{pmatrix}0\\0\\0\\1\end{pmatrix}$	$\begin{pmatrix}0\\0\\0\\1\end{pmatrix}$
A_2									
A_3									
A_4									

Tabelle 3.4: Vektoren mit einem Betrag von Eins

Von den Vektoren mit einem Betrag von Eins wird jeweils ein Repräsentant gleicher Vektoren abgespalten (Tabelle 3.5). Die restlichen Vektoren mit einem Betrag von Eins werden mit den Vektoren mit einem Betrag größer Eins zusammengelegt (Tabelle 3.6).

Zur Vereinfachung der weiteren Ausführungen wird zunächst ein Operator $\lceil\ \rceil$ eingeführt. Ohne Index bezeichnet er die Menge aller natürlichen Zahlen, die kleiner oder gleich der Kardinalität einer Menge sind:

$$\lceil M \rceil \quad = \quad \{\, i \,:\, 1 \leq i \leq |M| \,\} \tag{3.15}$$

$\vec{B}_1^3$	$\vec{B}_1^9$	$\vec{B}_1^4$	$\vec{B}_1^1$
$\begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$	$\begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}$	$\begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}$	$\begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$

Tabelle 3.5: Repräsentative Vektoren mit einem Betrag von Eins

$\vec{B}_1^2$	$\vec{B}_1^5$	$\vec{B}_1^{12}$	$\vec{B}_1^8$	$\vec{B}_1^6$	$\vec{B}_1^{11}$	$\vec{B}_1^7$	$\vec{B}_1^{10}$
$\begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \end{pmatrix}$	$\begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \end{pmatrix}$	$\begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \end{pmatrix}$	$\begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$	$\begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}$	$\begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}$	$\begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$	$\begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$

Tabelle 3.6: Restliche Vektoren

Beispielsweise ist $\lceil \{A, B, C\} \rceil = \{1, 2, 3\}$.

Durch oberes Indizieren kann eindeutig auf die Elemente der Menge zugegriffen werden:

$$\forall i \in \lceil M \rceil : \lceil M \rceil^i \in M \tag{3.16}$$

$$\forall i, j \in \lceil M \rceil : (\lceil M \rceil^i = \lceil M \rceil^j) \rightarrow (i = j) \tag{3.17}$$

Die könnte z.B. zu $\lceil \{A, B, C\} \rceil^1 = B$, $\lceil \{A, B, C\} \rceil^2 = A$ und $\lceil \{A, B, C\} \rceil^3 = C$ führen.

Die Bildung der Vektoren geschieht dann folgendermaßen:

$$i \in \lceil B \rceil \iff \vec{B}_g^i = \begin{pmatrix} B^i_{\lceil G_g \rceil^1} \\ \vdots \\ B^i_{\lceil G_g \rceil^{|G_g|}} \end{pmatrix} \tag{3.18}$$

Die Vektoren werden anschließend unter den Mengen Ψ_g, Δ_g und Θ_g aufgeteilt. Ψ_g ist dabei eine Menge von Untermengen. Die Mengen (bzw. ihre Untermengen) enthalten nicht die Vektoren $\vec{B}_g^i$ selbst, sondern nur deren Indizes i.

$$i \in \lceil B \rceil \iff \exists \psi \in \Psi_g : i \in \psi \cup \Delta_g \cup \Theta_g \tag{3.19}$$

Die Menge Θ_g erhält die Indizes aller Vektoren, die nur Nullen als Elemente besitzen. In der ersten Gruppe des Beispiel kommt dies nicht vor, Θ_1 wäre entsprechend leer.

$$\left|\vec{B_g^i}\right| = 0 \quad \Longleftrightarrow \quad i \in \Theta_g \tag{3.20}$$

In Δ_g werden die Indizes von Repräsentanten gleicher Vektoren mit einem Betrag von Eins aufgenommen. Dies entspricht Tabelle 3.5. Δ_1 kann somit die Menge $\{1, 3, 4, 9\}$ zugewiesen werden.

$$\left|\vec{B_g^i}\right| = 1 \quad \Longleftrightarrow \quad \exists j \in \Delta_g : \vec{B_g^i} = \vec{B_g^j} \tag{3.21}$$

$$\forall i, j \in \Delta_g : \left(\vec{B_g^i} = \vec{B_g^j}\right) \rightarrow (i = j) \tag{3.22}$$

Die Indizes der restlichen Vektoren gehen in die Menge Ψ_g ein. Dabei wird für jede Gruppe gleicher Vektoren eine Untermenge angelegt.

$$\forall \psi \in \Psi_g : \forall i, j \in \psi : \vec{B_g^i} = \vec{B_g^j} \tag{3.23}$$

$$\forall \psi, \psi' \in \Psi_g : \exists\, i \in \psi, j \in \psi' : \left(\vec{B_g^i} = \vec{B_g^j}\right) \rightarrow (\psi = \psi') \tag{3.24}$$

Tabelle 3.6 korrespondiert zur Menge $\Psi_1 = \{\{2, 5\}, \{6, 11\}, \{7, 10\}, \{8\}, \{12\}\}$.

Durch die Aufspaltung ist Formel 3.14 gleichbedeutend mit:

$$\sum_{\psi \in \Psi_g, i \in \psi} \alpha_g^i \times B^i \; + \; \sum_{i \in \Delta_g} \alpha_g^i \times B^i \; + \; \sum_{i \in \Theta_g} \alpha_g^i \times B^i \;\; = \;\; \Sigma_g$$

3.6.1.2 Vorrangige Koeffizienten bestimmen

Die Koeffizienten der zu den in Ψ_g gehörigen Indizes können nun gebildet werden. Zur Bestimmung der Koeffizienten muß die folgende Bedingung gewahrt bleiben:

- Die Koeffizienten unterschiedlicher Vektoren dürfen nicht untereinander substituierbar sein.

Werden bei den drei Vektoren aus Ψ_1 des Beispiels

$$\vec{B_1^5} = \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \end{pmatrix}, \quad \vec{B_1^6} = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} \quad \text{und} \quad \vec{B_1^7} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

die Koeffizienten α_1^6 und α_1^7 trotz $\vec{B}_1^6 \neq \vec{B}_1^7$ gleichgesetzt, so gilt nach Formel 3.14:

$$\ldots + \alpha_1^6 \times \left(B^6 + B^7\right) + \ldots = \Sigma_1$$

Die vorgegebenen Belegungen können zwar erfüllt werden, doch die Eindeutigkeit ist verletzt. Die Aktivierung von B^6 und B^7 kann beliebig vertauscht werden, obwohl entsprechende Alternativen eventuell gar nicht existieren. In diesem Fall wäre z.B. eine nicht vorgesehene Belegung von $B^5 = 1$, $B^6 = 0$ und $B^7 = 1$ ebenfalls möglich.

Diese Verletzung der Eindeutigkeit beschränkt sich nicht nur auf die Gleichsetzung von Koeffizienten, sondern gilt allgemein bei Substituierbarkeit von Koeffizienten.

Damit die Koeffizienten nicht unnötig groß werden, soll außerdem gelten:

- Gleiche Vektoren führen zu gleichen Koeffizienten.
- Der kleinste Koeffizient wird mit Eins belegt.

Um den Anforderungen gerecht zu werden, erfolgt deshalb eine sukzessive Bildung der Koeffizienten, beginnend mit Eins. Bei gleichen Vektoren erhalten die entsprechenden Koeffizienten dieselben Werte. Zur Umgehung einer möglichen Substituierbarkeit wird bei einem Vektor, der nicht gleich seinem Vorgänger ist, der Koeffizient durch die um Eins erhöhte Summe der bereits berechneten Koeffizienten gebildet.

$$\Omega_g^0 = 0 \tag{3.25}$$

$$\forall d \in \lceil \Psi_g \rceil : \Omega_g^d = \left(\Omega_g^{d-1} + 1\right) \times \left|\lceil \Psi_g \rceil^d\right| + \Omega_g^{d-1} \tag{3.26}$$

$$\forall d \in \lceil \Psi_g \rceil : \forall i \in \lceil \Psi_g \rceil^d : \alpha_g^i = \Omega_g^{d-1} + 1 \tag{3.27}$$

Die Anwendung auf das Beispiel führt zu folgender Belegung:

$$\begin{array}{rclcrcl}
\alpha_1^2 & = & \alpha_1^5 & = & 0+1 & = & 1 \\
\alpha_1^6 & = & \alpha_1^{11} & = & (0+1) \times 2 + 0 + 1 & = & 3 \\
\alpha_1^7 & = & \alpha_1^{10} & = & (2+1) \times 2 + 2 + 1 & = & 9 \\
 & & \alpha_1^8 & = & (8+1) \times 2 + 8 + 1 & = & 27 \\
 & & \alpha_1^{12} & = & (26+1) \times 1 + 26 + 1 & = & 54
\end{array}$$

3.6.1.3 Konstante errechnen

Mittels der bereits bestimmten Koeffizienten wird die Konstante Σ_g auf folgende Weise berechnet:

$$\Sigma_g = \begin{cases} 0 & : \quad |\Psi_g| + |\Delta_g| = 0 \\ 2 \times \Omega_g^{|\Psi_g|} + 1 & : \quad |\Psi_g| + |\Delta_g| > 0 \end{cases} \tag{3.28}$$

Der Wert der Konstante des Beispiels Σ_1 beträgt somit 215.

Die Höhe der Konstanten läßt sich nicht durch die Reihenfolge der Bildung der Koeffizienten aus Ψ_g beeinflussen. Egal in welcher Reihenfolge vorgegangen wird, ergibt sich immer derselbe Wert für die Konstante (siehe z.B. Abschnitt 3.6.1.5).

3.6.1.4 Übrige Koeffizienten bestimmen

Damit Gleichung 3.14 bei einer Belegung der Variablen entsprechend einer Alternative erfüllt ist, muß die Summe der aktivierten Koeffizienten gleich der Konstanten Σ_g sein. Bei der ersten Alternative muß beispielsweise $\alpha_1^2 + \alpha_1^3 + \alpha_1^5 + \alpha_1^8 = \Sigma_1$ gelten.

Durch das bisherige Vorgehen sind für jede Alternative alle Koeffizienten bis auf einen bestimmt (α_1^1, α_1^3, α_1^4 bzw. α_1^9). Dies entspricht den Indizes der Menge Δ_g. Die noch nicht bestimmten Koeffizienten können dafür benutzt werden, jeweils den Wert der linken Seite der Gleichung auf die Konstante Σ_g anzuheben:

$$\forall \psi \in \Psi_g : \forall i \in \Delta_g, j \in \psi : \left(\vec{v}_g^{\,i} \cdot \vec{v}_g^{\,j} = 1\right) \iff \left(j \in \Delta_g^i\right) \tag{3.29}$$

$$\forall i \in \Delta_g : \alpha_g^i = \Sigma_g - \sum_{j=1}^{|\Delta_g^i|} \alpha_g^{\lceil \Delta_g^i \rceil j} \tag{3.30}$$

Damit wäre ebenso die Voraussetzung für die Aufteilung von Alternativen in Gruppen geklärt: Für jede in einer Gruppe G_g befindliche Alternative muß ein Vektor mit einem Betrag von Eins existieren, der die Eins in der Zeile der betreffenden Alternativen hat. Dadurch kann für jede Alternative die Erreichung der Konstanten Σ_g gewährleistet werden.

Der Rest der Koeffizienten der ersten Gruppe des Beispiels kann nun errechnet werden:

$$\begin{aligned} \alpha_1^1 &= 215 - (9 + 9) &&= 197 \\ \alpha_1^3 &= 215 - (1 + 1 + 27) &&= 186 \\ \alpha_1^4 &= 215 - 54 &&= 161 \\ \alpha_1^9 &= 215 - (1 + 1 + 54 + 3 + 3) &&= 153 \end{aligned}$$

Die zu den Indizes der Menge Θ_g gehörigen Koeffizienten dürfen nicht aktiviert werden. Dies geschieht ebenso wie in Abschnitt 3.5.1 durch die Belegung mit einen Wert größer Σ_g:

$$\forall i \in \Theta_g : \quad \alpha_g^i = \Sigma_g + 1 \tag{3.31}$$

Da Θ_1 der ersten Gruppe des Beispiels leer ist, braucht hier keine weitere Berechnung stattzufinden. Alle benötigten Elemente sind bereits vollständig bestimmt. Die resultierende Gleichung lautet wie folgt:

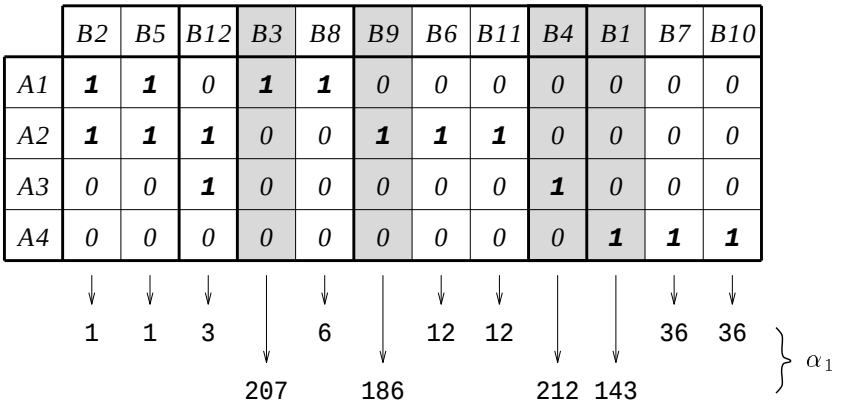
$$197 \times B^1 + B^2 + 186 \times B^3 + 161 \times B^4 + B^5 + 3 \times B^6 + 9 \times B^7 + 27 \times B^8 + 153 \times B^9 + 9 \times B^{10} + 3 \times B^{11} + 54 \times B^{12} = 215$$

Die Erstellung der Gleichung für die zweite Gruppe G_2 liefert folgendes Ergebnis:

$$5 \times B^1 + 10 \times B^2 + 10 \times B^3 + B^4 + 10 \times B^5 + 10 \times B^6 + B^7 + 10 \times B^8 + 10 \times B^9 + B^{10} + 10 \times B^{11} + B^{12} = 9$$

3.6.1.5 Zusammenfassung

Die vollzogenen Schritte zur Berechnung der Konstanten Σ_1 und den Koeffizienten α_1^i faßt Abbildung 3.6 zusammen. Im Gegensatz zu den vorigen Abschnitten wurde dabei die Reihenfolge der Belegung der zu den Indizes aus Ψ_g gehörigen Koeffizienten variiert.



$$\Sigma_1 = (1 + 1 + 3 + 6 + 12 + 12 + 36 + 36) \times 2 + 1 = 215$$

Abbildung 3.6: Zusammenfassung der Berechnungen

Ausgehend von der spaltenweise sortierten Tabelle der Gruppe G_1 werden die zur Menge Δ_g gehörigen Spalten abgespalten (grau unterlegt). Hierauf folgt die Bestimmung der zu Ψ_g gehörigen Koeffizienten nach dem in Abschnitt 3.6.1.2 geschilderten Prinzip (kurze Pfeile). Nach der Errechnung der Konstanten Σ_1 können auch die restlichen Koeffizienten bestimmt werden (längere Pfeile).

Anhand einer anderen Problemstellung wird die Umsetzung durch Gruppen in Abschnitt 4.4 nochmals ausführlicher dargestellt.

3.6.2 Korrektheit

Es ist bereits klar, daß eine Belegung der Variablen mit den Werten einer Alternative die erstellte Gleichung erfüllt. Es muß jedoch noch gezeigt werden, daß jede andere Belegung die Gleichung ungültig werden läßt. Dies geschieht im folgenden.

Ist durch die Alternativen bei einer Variablen ausschließlich die Belegung von Null zugelassen, wird die Aktivierung der Variabeln durch Formel 3.31 verhindert. Im weiteren müssen deshalb nur noch die zu den Mengen Psi_g und Δ_g gehörigen Koeffizenten in Betracht gezogen werden.

Es muß immer[7] genau ein zu einem Index aus Δ_g gehöriger Koeffizient aktiviert werden, um die Gleichung zu erfüllen. Wenn zwei derartige Koeffizienten aktiviert würden, wäre die Konstante Σ_g auf jeden Fall überschritten. Mit der Aktivation keines dieser Koeffizienten kann der Wert der Konstanten hingegen nicht erreicht werden. Dies wird im folgenden gezeigt:

$$\sum_{\psi\in\Psi, i\in\psi} \alpha_g^i \quad < \quad \Sigma_g \quad < \quad \alpha_g^a + \alpha_g^b \quad \forall a, b \in \Delta_g, a \neq b \tag{3.32}$$

(Anwendung von Gleichung 3.30,
Abschätzung der oberen Schranke über Formel 3.29)

$$\sum_{\psi\in\Psi, i\in\psi} \alpha_g^i \quad < \quad \Sigma_g \quad < \quad 2 \times \left(\Sigma_g - \sum_{\psi\in\Psi, i\in\psi} \alpha_g^i \right) \tag{3.33}$$

(Anwendung von Gleichung 3.28 für den Fall: $|\Psi_g| + |\Delta_g| > 0$)

$$\sum_{\psi\in\Psi, i\in\psi} \alpha_g^i \quad < \quad 2 \times \Omega_g^{|\Psi_g|} + 1 \quad < \quad 2 \times \left(2 \times \Omega_g^{|\Psi_g|} + 1 - \sum_{\psi\in\Psi, i\in\psi} \alpha_g^i \right) \tag{3.34}$$

[7] Eine Ausnahme bildet die Situation, wenn die Gruppe nur eine Alternative beinhaltet, die ausschließlich aus Nullen besteht. Dieser Fall wird im Anschluß behandelt.

Die Omegas Ω_g beschreiben die Summe einer wachsenden Untermenge von Koeffizienten (Formeln 3.25 bis 3.27). Das letzte Omega beinhaltet die Gesamtsumme der durch die Indizes der Menge Ψ_g beschriebenen Koeffizienten:

$$\sum_{\psi \in \Psi, i \in \psi} \alpha_g^i \quad = \quad \Omega_g^{|\Psi_g|} \tag{3.35}$$

Die Gültigkeit der Formel 3.34 kann nun durch Substitution auf Basis der Gleichung 3.35 gezeigt werden:

$$\Omega_g^{|\Psi_g|} \quad < \quad 2 \times \Omega_g^{|\Psi_g|} + 1 \quad < \quad 2 \times \Omega_g^{|\Psi_g|} + 2 \tag{3.36}$$

Es wurde gezeigt, daß genau ein zu einem Index aus Δ_g gehöriger Koeffizient aktiviert werden muß, um die Gleichung erfüllbar werden zu lassen. Ist dies geschehen, müssen weitere zu Indizes der Menge Ψ_g gehörige Koeffizienten aktiviert werden, um die Gleichheit zu etablieren. Dafür existiert eine eindeutige Auswahl, denn diese Koeffizienten sind bereits mit der Zielsetzung der Eindeutigkeit gebildet worden (Formeln 3.25 bis 3.27). Als Konsequenz entspricht die Belegung der Variablen genau einer Alternative.

Dies gilt auch bei der bereits erwähnten Ausnahme, bei der nur eine Alternative in der Gruppe vertreten ist, die ausschließlich aus Nullen besteht. In diesem Fall werden alle Indizes der Menge Θ_g zugeordnet. Die Konstante Σ_g muß dann nach Formel 3.28 auf Null gesetzt werden. Die erstellt Gleichung ist wie gewünscht nur mit einer Belegung von Nullen erfüllbar.

3.6.3 Kopplung durch Schaltvariablen

Die Kopplung der Gleichungen der Gruppen erfolgt wie in Abschnitt 3.4.2 durch die Einfügung von Schaltvariablen und deren Kopplung entsprechend Formel 3.5:

$$\sum_{G_g} S_g \quad = \quad 1 \tag{3.37}$$

3.7 Vergleich der Algorithmen

Für einen empirischen Vergleich der Laufzeiten der vorgestellten Algorithmen müssen die Alternativen in ein größeres Constraint-System eingebettet werden, denn erst bei höherdimensionierten Gesamtproblemen werden die Unterschiede deutlich.

3.7.1 Vervielfältigung

Zur Ausweitung der Problemstellung bietet es sich an, ein System von Alternativen zu vervielfältigen und über ein Optimierungsziel zu verbinden. Das gesamte Problem besteht dann aus entsprechend der Vervielfältigung vielen Teilproblemen. Im folgenden wird die Anzahl der Vervielfältigungen als *Dimension* bezeichnete. Abbildung 3.7 zeigt ein Beispiel mit fünf Dimensionen und jeweils vier Alternativen.

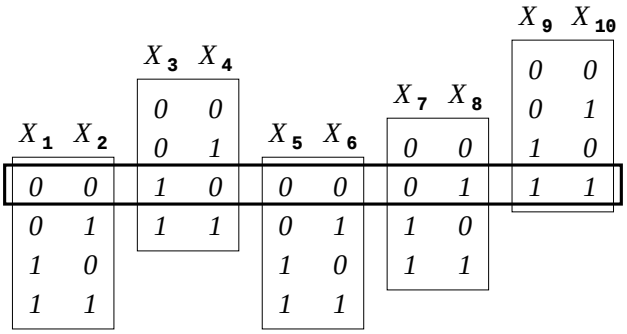


Abbildung 3.7: Beispiel für die Kombination vervielfältigter Alternativen

3.7.2 Verbindung der Teilprobleme

Die Verbindung der Teilprobleme geschieht durch ein Optimierungsziel. Es soll erreicht sein, wenn die einzelnen Alternativen insgesamt eine vorgegebene Binärzahl bilden (umrandet in Abbildung 3.7). Diese Optimierung ist einfach zu beschreiben. Für das Beispiel von Abbildung 3.7 kann sie folgendermaßen ausgedrückt werden:

$$\begin{aligned} X^3 + X^8 + X^9 + X^{10} + K &= 4 + X^1 + X^2 + X^4 + X^5 + X^6 + X^7 \\ K &\rightarrow min \end{aligned}$$

Allgemein erfolgt die Optimierung für eine Binärzahl mit Komponenten B^i durch:

$$\begin{aligned} \sum_{B^i=1} X^i + K &\geq \sum B^i + \sum_{B^i=0} X^i \\ K &\rightarrow min \end{aligned}$$

3.7.3 Test sämtlicher Binärzahlen

Mit a Alternativen und d Dimensionen ergeben sich a^d Möglichkeiten, eine Binärzahl vorzugeben. Für jede dieser Möglichkeiten wird ermittelt, wie lange ein Algorithmus zur Bestimmung des Optimums benötigt[8]. In Abbildung 3.8 sind die Methoden bezüglich dieses Aspektes verglichen.

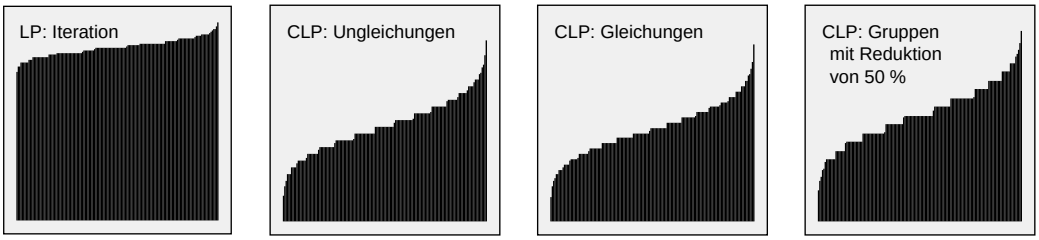


Abbildung 3.8: Nach Länge sortierte Laufzeiten für alle möglichen Binärzahlen

Die kleineren Differenzen der Laufzeiten bei der Umsetzung durch Iteration ergeben sich durch unterschiedliche Anzahlen von Verbesserungen der aktuellen Lösung.

Die unregelmäßigere Struktur der Umsetzung durch Gruppen liegt an einer unterschiedlich schnellen Propagation für die einzelnen Alternativen. Alternativen einer kleinen Gruppe werden schneller berücksichtigt als die einer größeren Gruppe. Die Gruppenbildung kann somit zu einer Steuerung der Propagation verwendet werden.

3.7.4 Durchschnittliche Laufzeiten

Aus diesen Testläufen wird anschließend ein Durchschnitt gebildet. Abbildung 3.9 zeigt die berechneten Durchschnitte für Lösungen durch Iteration, Ungleichungen und Gleichungen.

[8]Für die constraint-basierten Verfahren wird hierzu das CHIP-Prädikat `min_max` verwendet. Alle Programm-Texte sind mit weiteren Erläuterungen in Anhang A zu finden.

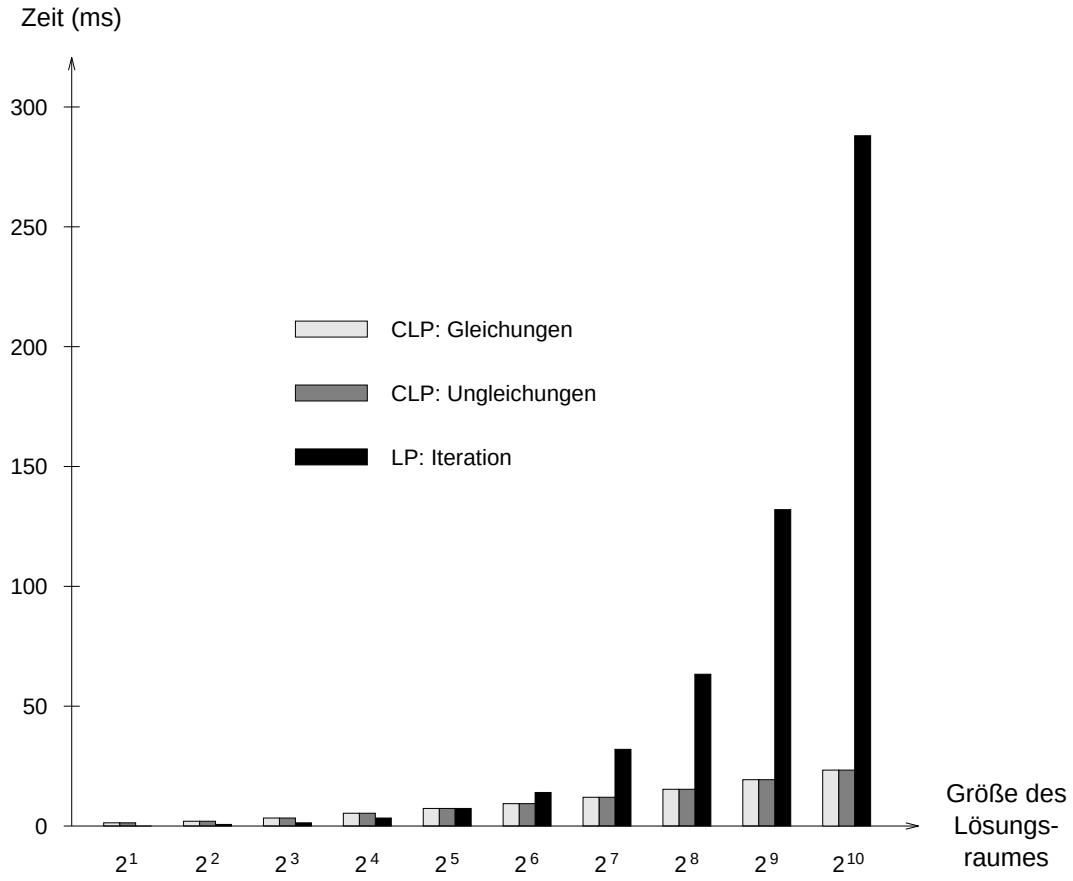


Abbildung 3.9: Vergleich bei zwei Alternativen pro Dimension (2^1 bis 2^{10})

Die Durchschnittszeit beim Ansatz der Iteration verdoppelt sich ungefähr pro Dimension. Dies ist verständlich, da sich die Anzahl der möglichen Lösungen ebenfalls verdoppelt. Die Laufzeiten einer Umsetzung durch Ungleichungen sind identisch mit denen einer Umsetzung durch Gleichungen. Ist der gesamte Lösungsraum des Suchproblems größer als $2^5 = 32$, was bei relevanten Anwendungen auf jeden Fall gegeben ist, ergibt sich ein Vorteil gegenüber der Iteration. Dieser Vorteil wird umso deutlicher, desto größer der gesamte Lösungsraum ist.

Um das Wachstum der vorgestellten Algorithmen besser untersuchen zu können, müssen höhere Dimensionen getestet werden. Allerdings ist in diesen Dimensionen das Testen sämtlicher Binärzahlen nicht mehr möglich. Abbildung 3.10 beinhaltet somit nur repräsentative Testläufe[9].

Die Überlegenheit der Constraint-Lösungen ist offensichtlich. Dem **exponentielle Wachstum** der Iteration steht ein **polynomisches Wachstum** entgegen

[9] Zur Auswahl der repräsentativen Testläufe siehe Anhang A.2.3.

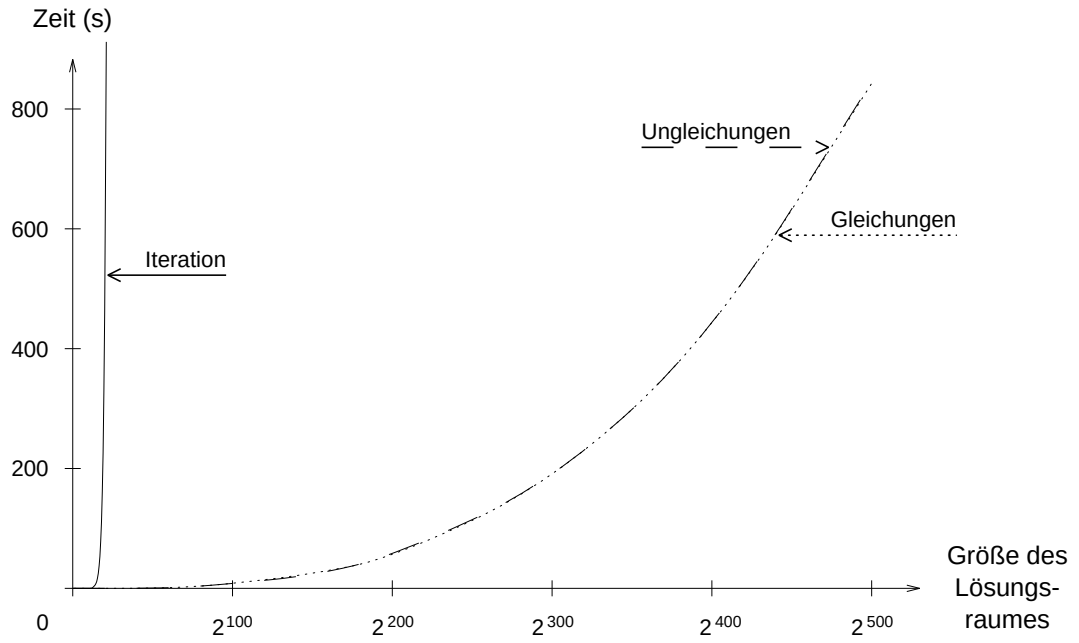


Abbildung 3.10: Vergleich bei zwei Alternativen pro Dimension (2^1 bis 2^{500})

(fallende zweite Ableitungen). Die Umsetzung durch Ungleichungen zeigt kaum Unterschiede zu einer Umsetzung durch Gleichungen. Minimale Abweichungen lassen sich auf variierenden Belastungen der CPU während der Ausführung der Testläufe zurückführen.

Die Vorteile einer Umsetzung durch Gruppen werden in Abbildung 3.11 deutlich. Als Grundlage wurde hier ein System von 16 Alternativen gewählt, welche sich bei einer Umsetzung durch Gruppen lediglich um 50% reduzieren lassen, d.h. es entstehen acht Gruppen. Obwohl die Gruppen-Reduktion nur 50% beträgt, ergibt sich bereits eine mehrfache Beschleunigung gegenüber den anderen Verfahren.

Ebenso werden nun Unterschiede zwischen Gleichungs- und Ungleichungs-Ansatz deutlich. Eine Lösung durch Ungleichungen ist mindestens so schnell wie eine Lösung durch Gleichungen. Mit steigender Anzahl von Alternativen ergeben sich jedoch Vorteile zugunsten der Umsetzung durch Ungleichungen.

Diese Erkenntnisse bestätigen sich bei Verwendung einer noch größeren Anzahl von Alternativen. Abbildung 3.12 liegt ein System von 32 Alternativen zugrunde, welche bei einer Umsetzung durch Gruppen auf 12 Gruppen verteilt werden (Reduktion von 62.5%).

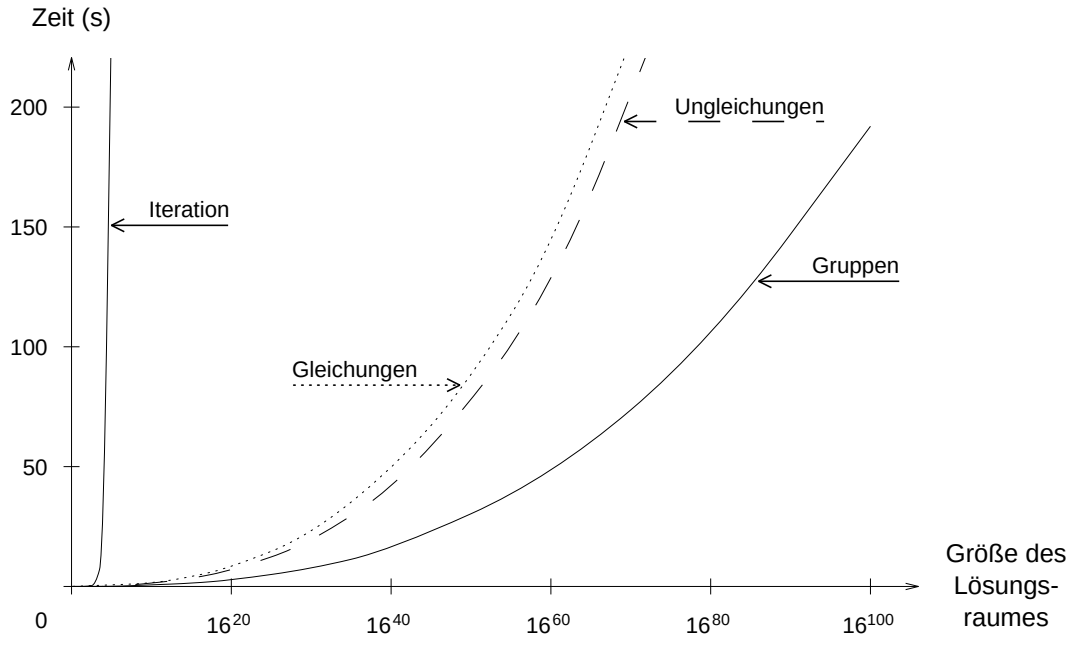


Abbildung 3.11: Vergleich bei 16 Alternativen pro Dimension

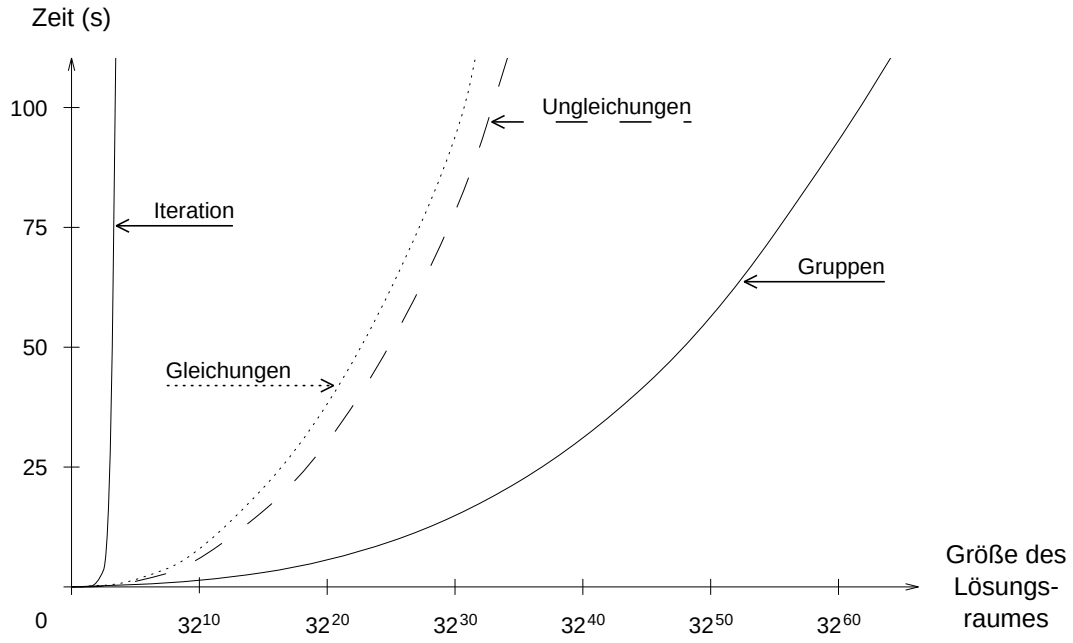


Abbildung 3.12: Vergleich bei 32 Alternativen pro Dimension

Kapitel 4

Meta-Constraints

Das allgemeine Vorgehen zur Umsetzung von Meta-Constraints beschreibt Abbildung 4.1. Zunächst müssen Schaltvariablen für die Basis-Constraints[1] (**1**) entsprechend Abschnitt 2.1.2 gebildet werden. Ein Meta-Constraint wird in explizite Alternativen umgesetzt (**2**). Aus Schaltvariablen und expliziten Alternativen wird eine Alternativen-Tabelle gebildet (**3**). Anschließend erfolgt eine Umsetzung durch Gruppen, deren resultierende Gleichungen über Gleichung 3.37 gekoppelt werden (**4**).

In diese Gleichung wird eine Schaltvariable eingeführt (**5**), mittels derer dann die Gültigkeit des jeweiligen Meta-Constraints gesteuert werden kann. Diese Schaltvariable kann im weiteren genauso wie die Schaltvariablen der Basis-Constraints für übergeordnete Meta-Constraints verwendet werden.

In einigen Fällen ist es möglich, einfache Meta-Constraints durch spezialisierte Formeln auszudrücken (**6**), in die direkt eine Schaltvariable eingefügt wird (**7**).

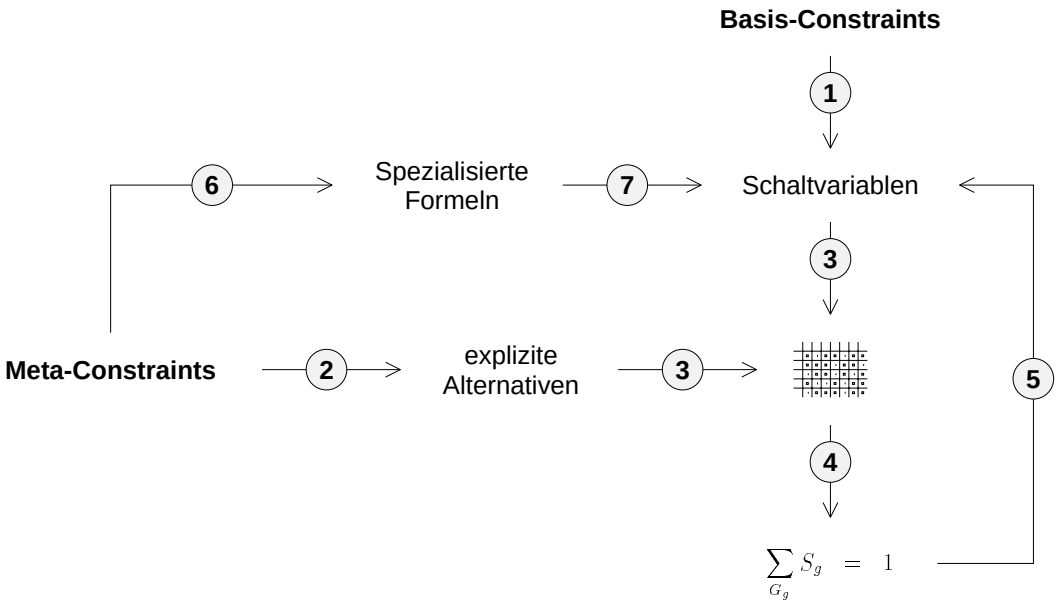


Abbildung 4.1: Vorgehensweise bei einer Umsetzung von Meta-Constraints

Das Vorgehen wird im weiteren anhand der in Abbildung 4.2 dargestellten stückweise linearen Funktion erläutert.

[1] Basis-Constraints sind Constraints, deren Gültigkeit durch die Meta-Constraints geregelt werden soll.

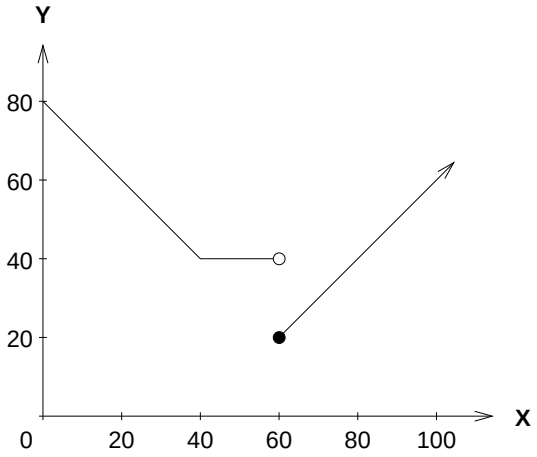


Abbildung 4.2: Beispiel einer stückweise linearen Funktion

4.1 Behandlung der Basis-Constraints

Die Funktion setzt sich aus drei unterschiedlichen Segmenten zusammen. Diese können durch sieben Basis-Constraints beschrieben werden:

Erstes Segment:	$C^1: X < 40$	$C^2: X + Y = 80$	
Zweites Segment:	$C^3: 40 \leq X$	$C^4: X < 60$	$C^5: Y = 40$
Drittes Segment:	$C^6: 60 \leq X$	$C^7: 2 \times X = Y + 40$	

Die Constraints sind nur innerhalb ihres Segmentes gültig. Deshalb werden in alle Basis-Constraints Schaltvariablen eingeführt (S^1 bis S^7), die Constraints nicht zutreffender Segmente außer Kraft setzen können.

Diese Schaltvariablen müssen **eindeutig** sein, da die Möglichkeit der Propagation in verschiedenen Richtung erhalten bleiben soll. Im Fall von C^1 ergibt sich nach Abschnitt 2.1.2:

$$\begin{aligned} 41 + X_{max} &> X + S^1 \times (X_{max} + 1) \\ 40 &\leq X + S^1 \times 40 \end{aligned}$$

4.2 Generierung von Meta-Constraints

Die Relationen zwischen den Basis-Constraints werden über vier Meta-Constraints M^1 bis M^4 beschrieben:

M^1 :		C^1	`AND`	C^2	
M^2 :	C^3	`AND`	C^4	`AND`	C^5
M^3 :		C^6	`AND`	C^7	
M^4 :	M^1	`XOR`	M^2	`XOR`	M^3

Tabelle 4.1: Meta-Constraints

Die Meta-Constraints können in explizite Alternativen umgesetzt werden. Dazu werden die Alternativen für ein zweiwertiges `AND` und dreiwertiges `AND` und `XOR` benötigt:

Zweiwertiges `AND`:

A_1	1	1

Dreiwertiges `AND`:

A_1	1	1	1

Dreiwertiges `XOR`:

A_1	1	0	0
A_2	0	1	0
A_3	0	0	1

Tabelle 4.2: Alternativen der Meta-Constraints

Mit den im vorigen Abschnitt erstellten Schaltvariablen für die Basis-Constraints lassen sich anschließend Alternativen-Tabellen bilden (siehe Tabelle 4.3), deren Behandlung in Abschnitt 3.6 beschrieben ist.

In die die Gruppen verbindende Gleichung 3.37 kann wiederum eine Schaltvariable (M^1 bis M^3) eingeführt werden, durch die die Gültigkeit des Meta-Constraints gesteuert bzw. auf die sie abgebildet wird. Lassen sich alle Alternativen des Meta-Constraints in nur einer Gruppe zusammenfassen, so kann Gleichung 3.37 entfallen und die Schaltvariable direkt in die Gruppen-Gleichung eingeführt werden.

Die Schaltvariablen der Meta-Constraints können im weiteren auf die gleiche Weise wie die Schaltvariablen der Basis-Constraints für übergeordnete Meta-Constraints verwendet werden (z.B. für M^4). In Constraints an der Spitze der Hierarchie werden keine Schaltvariablen eingeführt, diese müssen gelten.

M^1	C^1	C^2
A_1	1	1

M^2	C^3	C^4	C^5
A_1	1	1	1

M^3	C^6	C^7
A_1	1	1

M^4	M^1	M^2	M^3
A_1	1	0	0
A_2	0	1	0
A_3	0	0	1

Tabelle 4.3: Alternativen-Tabellen der Meta-Constraints

4.3 Spezialisierte Formeln

Bei der sehr einfachen Struktur der Meta-Constraints des Beispiels lassen sich statt der dargelegten Umsetzung spezialisierte Formeln verwenden[2]. Die Schaltvariablen für die Meta-Constraints könnten in diesem Fall in folgende Formeln eingefügt werden:

$$\begin{aligned} M^1 &: C^1 + C^2 \geq 2 \\ M^2 &: C^3 + C^4 + C^5 \geq 3 \\ M^3 &: C^6 + C^7 \geq 2 \\ M^4 &: M^1 + M^2 + M^3 = 1 \end{aligned}$$

Bei der Beispiel-Funktion muß auch bei einer Umsetzung durch Gruppen jeweils nur eine Formel behandelt werden, da sich die Alternativen in einer Gruppen zusammenfassen lassen. In dieser Beziehung sind die beiden Lösungen gleichwertig. Allerdings sind die spezialisierten Formeln für die Meta-Constraints M^1 bis M^3 Ungleichungen, in die sich Schaltvariablen einfacher einbinden lassen.

Der eingeschränkte aber effiziente Ansatz spezialisierter Formeln steht dem der generelleren expliziten Alternativen nicht entgegen, sinnvoll ist eine Kombination der Ansätze. Bei Meta-Constraints, für die spezialisierte Formeln bekannt sind,

[2]Eine Sammlung spezialisierter Formeln ist in [Mitra94] zu finden.

sollten diese verwendet werden, da sie in der Regel zu einer effizienteren Abarbeitung führen. Eine Umsetzung durch explizite Alternativen läßt sich bei allen anderen Meta-Constraints durchführen.

4.4 Einebnen

Es ist möglich, das Netzwerk von Meta-Constraints durch eine Vorverarbeitung teilweise einzuebnen (siehe Abbildung 4.3). Im Extrem bleibt nur ein Meta-Constraint übrig, dessen Alternativen eine disjunktive Normalform bilden.

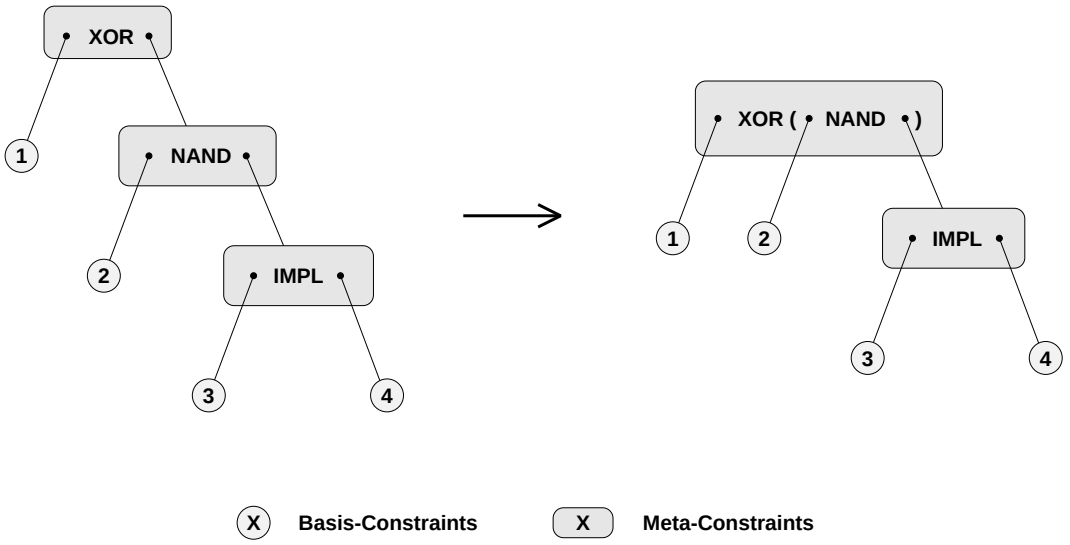


Abbildung 4.3: Einebnung eines Meta-Constraints

Die Hoffnung ist, daß der Vorteil der danach effizienteren Lösungssuche den anfänglichen Mehraufwand überwiegt. Bei größer dimensionierten Problemen wird ein solches Vorgehen kaum vermeidbar sein. Auch kann eine eingeebnete Darstellung zusätzlich abgesetzt werden, um eine bessere Propagation durch Ausnutzung der Redundanzen zu erzielen[3].

In schwach strukturierten Bereichen kann die Anzahl von benötigten Alternativen durch das Einebnen stark zunehmen, da dort bei der Kombination der Alternativen der Meta-Constraints sehr wenig Alternativen entfallen. Trotz der durch den erweiterten Zustandsraum sehr viel besseren Kompression bei der Gruppenbildung (siehe Abschnitt A.2.1) wird der zu erwartende Laufzeitgewinn bei der Lösungssuche deshalb nicht sehr groß sein.

Wenn das Netzwerk von Meta-Constraints soweit eingeebnet wurde, daß nur eine

[3] Zu Vorteilen redundanter Modellierung siehe [Cheng96].

disjunktive Normalform von Alternativen bleibt, können mehrdeutige statt eindeutiger Schaltvariablen verwendet werden. Dies bringt einen zusätzlichen Laufzeitgewinn.

Anhand der Beispiel-Funktion von Abbildung 4.2 wird im folgenden ein Einebnen bis zur disjunktiven Normalform vorgeführt. Das Einebnen kann entweder durch Berechnungen oder auf analytischem Wege erfolgen. Beim Einebnen durch Berechnung werden die einzuebnenden Teile abgesetzt, deren mögliche Lösungen bestimmt und anschließend als neues Meta-Constraint verwendet.

Einfache Systeme von Meta-Constraints lassen eine analytische Vorgehensweise zu, die hier Anwendung finden soll. Da in jeder Alternative der disjunktiven Normalform alle Schaltvariabelen der Basis-Constraints enthalten sind, muß jeweils die Ausprägung sämtlicher Schaltvariablen betrachtet werden.

Für das erste Segment der Funktion wird eine Alternative A_1 arrangiert, bei der C^1, C^2 und C^4 wahr sind, während der Rest der Schaltvariablen der Basis-Constraints falsch ist. Das zweite Segment benötigt zwei Alternativen: eine für den Kontaktpunkt (A_2), an dem C^2 ebenfalls noch gilt, und eine für den Rest (A_3). Auch für das dritte Segment reicht eine Alternative nicht aus, da in einem Punkt die durch C^5 bestimmte Gerade geschnitten wird:

	C^1	C^2	C^3	C^4	C^5	C^6	C^7
A_1	1	1	0	1	0	0	0
A_2	0	1	1	1	1	0	0
A_3	0	0	1	1	1	0	0
A_4	0	0	1	0	1	1	1
A_5	0	0	1	0	0	1	1

Tabelle 4.4: Alternativen-Tabelle der eingeebneten Meta-Constraints

Die Umsetzung der Alternativen-Tabelle geschieht entsprechend Abschnitt 3.6. Abbildung 4.4 stellt die dazu nötige Berechnung der Gruppen-Gleichungen dar.

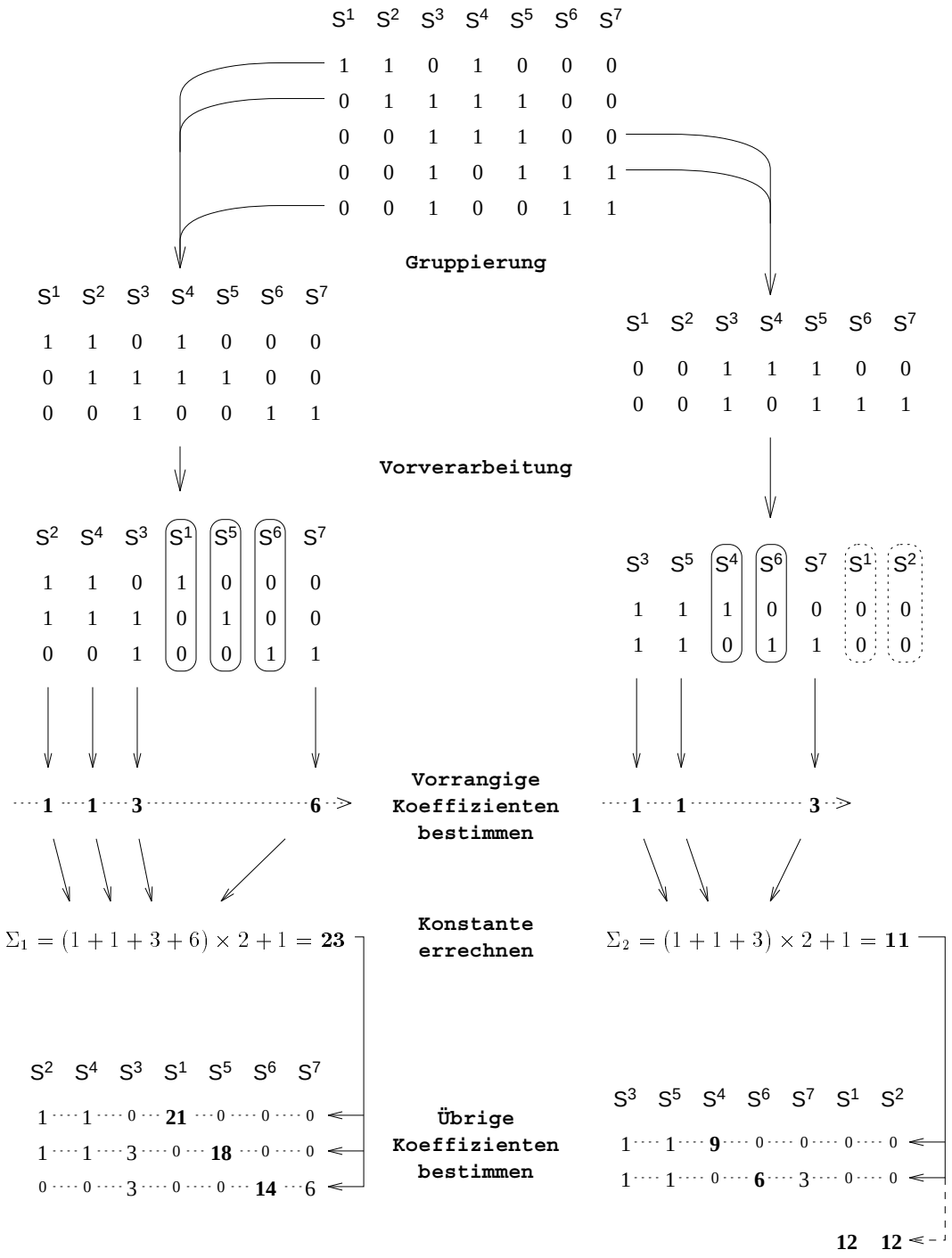


Abbildung 4.4: Berechnung der Gruppen-Gleichungen der Beispiel-Funktion

Kapitel 5

Resümee

Die eingeführten Algorithmen erlauben die Formulierung expliziter Relationen. Diese können einfache Variablen bis hin zu Constraints und Meta-Constraints in Beziehung setzen. Neben der Darstellung expliziter Relationen ist die Einbindung impliziter (Meta-)Relationen möglich, wie sie z.B. in [Mitra94] vorgeschlagen werden.

Eine Lösung über den Mechanismus des Backtrackings der Logischen Programmierung ist möglich, den aufgezeigten Constraint-Lösungen jedoch weit unterlegen. Das Problem der Disjunktion wird bei den Constraint-Lösungen durch Schaltvariablen bzw. über die unterschiedlichen Lösungen von linearen Gleichungen gelöst. Die expliziten Relationen lassen sich hiermit schnell und einfach in ein effizientes System von konjunktiven Constraints umsetzen, welches von gewöhnlichen Constraint-Solvern abgearbeitet werden kann.

Unter den Constraint-Lösungen zeigt wiederum eine Umsetzung durch Gruppenbildung das beste Laufzeitverhalten. Mit Zunahme der Größe des Gesamtsystems und der Komplexität der Relationen verstärkt sich diese Überlegenheit. Testläufe belegen eine effiziente Abarbeitung selbst in hohen Dimensionen und weisen damit auf die Tauglichkeit für Probleme der Praxis hin.

Derzeit werden die Algorithmen in Programme zur Konfiguration von Leitsystemen beim GMD-FIRST[1] und in einem Theorembeweiser des SICS[2] integriert.

[1]GMD – Forschungszentrum Informationstechnik GmbH, Forschungsinstitut für Rechnerarchitektur und Softwaretechnik

[2]Swedish Institute of Computer Science, Logic Programming Technology and Applications

Anhang A

Programmtexte

A.1 Programm-Module

Die aufgeführten Programm-Module realisieren die in den vorigen Abschnitten vorgestellten Algorithmen. Sie wurden zur empirischen Auswertung der Algorithmen in Abschnitt 3.7 verwendet.

A.1.1 `tools.pl`

Dieses Modul stellt einige Basis-Funktionalitäten bereit, die von den anderen Modulen genuzt wird.

```
%
% TOOL.PL
%
% Allgemeine Hilfspraedikate
%


build_addition([First|Other], First + AdditionOther) :-

      build_addition(Other, AdditionOther).

build_addition(_, 0).


add_list(List, Sum) :-

      build_addition(List, Addition),

      Sum is Addition.


sum_is_one([SwitchVariable|SwitchVariables], Sum) :-

      sum_is_one(SwitchVariables, Sum + SwitchVariable).

sum_is_one(_, Sum) :-

      Sum #= 1.


build_linear_combination([Variable|Variables], [Coefficient|Coefficients],
  Accu, LinearCombination) :-

      build_linear_combination(Variables, Coefficients,
        Accu + Coefficient * Variable, LinearCombination).

build_linear_combination(_, _, LinearCombination, LinearCombination).
```

A.1.2 umsetzung_ungleichungen.pl

Dieses Modul stellt Informationen zur Umsetzung durch Ungleichungen nach Abschnitt 3.4 bereit. Die Kopplung und das Absetzen der Constraints erfolgen in Modul kopplung_ungleichungen.pl.

```
%
% UMSETZUNG_UNGLEICHUNGEN.PL
%
% Berechnung der Koeffizienten und der Konstanten fuer eine
% Umsetzung durch Ungleichungen:
%
%   prepare_values(ungleichungen, Alternativen, Ergebnis)
%


compute_positions([0|OtherBools], [r|OtherPositions], Sum, MaxLeft) :-

      Sum is OldSum + 1,

      compute_positions(OtherBools, OtherPositions, OldSum, MaxLeft).

compute_positions([1|OtherBools], [l|OtherPositions], Sum, MaxLeft) :-

      Sum is OldSum + 1,
      MaxLeft is OldMaxLeft + 1,

      compute_positions(OtherBools, OtherPositions, OldSum, OldMaxLeft).

compute_positions(_, [], 0, 0).


prepare_values(ungleichungen, [Alternative|Alternatives],
  [[Positions, Sum, MaxLeft]|OtherValues]) :-

      compute_positions(Alternative, Positions, Sum, MaxLeft),

      prepare_values(ungleichungen, Alternatives, OtherValues).

prepare_values(ungleichungen, _, []).
```

A.1.3 kopplung_ungleichungen.pl

In diesem Modul erfolgen die Kopplung und das Absetzen der Constraints, welche die Umsetzung durch Ungleichungen entsprechend Abschnitt 3.4 realisieren.

```
%
```

```
% KOPPLUNG_UNGLEICHUNGEN.PL
%
% Kopplung der Alternativen durch Ungleichungen mit Schaltvariablen:
%
%   establish_alternatives(ungleichungen, Werte, Entscheidungs-Variablen,
%     Schaltvariablen)
%


:- [-tools].


build_sides([Variable|Variables], [l|OtherPositions], Variable + LeftSide,
  RightSide) :-

      build_sides(Variables, OtherPositions, LeftSide, RightSide).

build_sides([Variable|Variables], [r|OtherPositions], LeftSide,
  Variable + RightSide) :-

      build_sides(Variables, OtherPositions, LeftSide, RightSide).

build_sides(_, _, 0, 0).


process_ungleichungen(Variables, MaxRight, [[Positions, Sum, MaxLeft]|Values],
  [SwitchVariable|SwitchVariables]) :-

      build_sides(Variables, Positions, LeftSide, RightSide),

      SwitchVariable :: [0, 1],

      LeftSide + MaxRight #>= MaxLeft + RightSide + MaxRight * SwitchVariable,

      process_ungleichungen(Variables, MaxRight, Values, SwitchVariables).

process_ungleichungen(_, _, _, []).


establish_alternatives(ungleichungen, Variables, Values,
  SwitchVariables, []) :-

      length(Variables, VariableNumber),
      DoubledVariableNumber is 2 * VariableNumber,

      process_ungleichungen(Variables, DoubledVariableNumber, Values,
        SwitchVariables),

      sum_is_one(SwitchVariables, 0),

      !.
```

A.1.4 umsetzung_gleichungen.pl

Dieses Modul stellt Informationen zur Umsetzung durch Gleichungen nach Abschnitt 3.5 bereit. Die Kopplung und das Absetzen der Constraints erfolgen in Modul kopplung_gleichungen.pl.

```
%
% UMSETZUNG_GLEICHUNGEN.PL
%
% Berechnung der Koeffizienten und der Konstanten fuer eine
% Umsetzung durch Gleichungen:
%
%   prepare_values(gleichungen, Alternativen, Ergebnis)
%


:- [-tools].


build_coefficients([Bool|Bools], Constant, [Coefficient|Coefficients],
  Accu, MaxSum) :-

      Coefficient is (1 - Bool) * Constant + 1,

      NewAccu is Accu + Coefficient,

      build_coefficients(Bools, Constant, Coefficients, NewAccu, MaxSum).

build_coefficients(_, _, [], MaxSum, MaxSum).


prepare_values(gleichungen, [Alternative|Alternatives],
  [[Coefficients, Constant, MaxSum]|OtherValues]) :-

      add_list(Alternative, Constant),

      build_coefficients(Alternative, Constant, Coefficients, 0, MaxSum),

      prepare_values(gleichungen, Alternatives, OtherValues).

prepare_values(gleichungen, _, []).
```

A.1.5 kopplung_gleichungen.pl

In diesem Modul erfolgen die Kopplung und das Absetzen der Constraints, welche die Umsetzung durch Gleichungen (entsprechend Abschnitt 3.5) wie auch durch Gruppen (entsprechend Abschnitt 3.6) realisieren.

```
%
% KOPPLUNG_GLEICHUNGEN.PL
%
% Kopplung der Alternativen durch Gleichungen mit Schaltvariablen:
%
%   establish_alternatives(gleichungen/gruppen, Werte,
%     Entscheidungs-Variablen, Schalt-Variablen, Hilfs-Schalt-Variablen)
%


:- [-tools].


establish_equality_ambiguous(LinearCombination, MaxSum, MaxSum,
  SwitchVariable, []) :-

      SwitchVariable :: [0, 1],

      MaxSum * SwitchVariable #<= LinearCombination.

establish_equality_ambiguous(LinearCombination, 0, MaxSum,
  SwitchVariable, []) :-

      SwitchVariable :: [0, 1],

      MaxSum * SwitchVariable + LinearCombination #<= MaxSum.

establish_equality_ambiguous(LinearCombination, Sigma, MaxSum,
  SwitchVariable, [IncSwitchVariable, DecSwitchVariable]) :-

      [SwitchVariable, IncSwitchVariable, DecSwitchVariable] :: [0, 1],

      Sigma * IncSwitchVariable #<= LinearCombination,

      Distance is MaxSum - Sigma,

      Distance * DecSwitchVariable + LinearCombination #<= MaxSum,

      SwitchVariable + 1 #= IncSwitchVariable + DecSwitchVariable.


process_gleichungen(Variables, [[Coefficients, Sigma, MaxSum]|Values],
  [SwitchVariable|SwitchVariables], HelpSwitchVariables) :-

      build_linear_combination(Variables, Coefficients, 0,
        LinearCombination),

      establish_equality_ambiguous(LinearCombination, Sigma, MaxSum,
        SwitchVariable, HelpSwitchVariablePart1),

      process_gleichungen(Variables, Values, SwitchVariables,
```

```
        HelpSwitchVariablePart2),

      append(HelpSwitchVariablePart1, HelpSwitchVariablePart2,
        HelpSwitchVariables).

process_gleichungen(_, _, [], []).


establish_alternatives(gleichungen, Variables, Values, SwitchVariables,
  HelpSwitchVariables) :-

      process_gleichungen(Variables, Values, SwitchVariables,
        HelpSwitchVariables),

      sum_is_one(SwitchVariables, 0),

      !.

establish_alternatives(gruppen, Variables, Values, SwitchVariables,
  HelpSwitchVariables) :-

      establish_alternatives(gleichungen, Variables, Values, SwitchVariables,
        HelpSwitchVariables).
```

A.1.6 umsetzung_gruppen.pl

Dieses Modul stellt Informationen zur Umsetzung durch Gruppen nach Abschnitt 3.6 bereit. Die Kopplung und das Absetzen der Constraints erfolgen durch das Modul `kopplung_gleichungen.pl` wie bei einer Realisierung durch Gleichungen.

```
%
% UMSETZUNG_GRUPPEN.PL
%
% Berechnung der Koeffizienten und der Konstanten fuer eine
% Umsetzung durch Gruppen:
%
%   prepare_values(gruppen, Gruppierte Alternativen, Ergebnis)
%


:- [-tools].


%
% Vorverarbeitung
%

get_first_elements([[First|List]|Other], [First|Values], [List|Lists]) :-
```

```
      get_first_elements(Other, Values, Lists).

get_first_elements(_, [], []).


build_vectors(CGroup, [Vector|Vectors]) :-

      get_first_elements(CGroup, Vector, NewCGroup),

      build_vectors(NewCGroup, Vectors).

build_vectors(_, _).


decision_smt([_], 0, Vector, [], [], [Vector]).

decision_smt([_], _, Vector, [Vector], [], []).

decision_smt([0|List], Counter, Vector, Single, Multiple, Theta) :-

      decision_smt(List, Counter, Vector, Single, Multiple, Theta).

decision_smt([1|List], 0, Vector, Single, Multiple, Theta) :-

      decision_smt(List, 1, Vector, Single, Multiple, Theta).

decision_smt([1|List], 1, Vector, [], [Vector], []).


split_by_number([Vector|Vectors], Singles, Multiples, Thetas) :-

      decision_smt(Vector, 0, Vector, Single, Multiple, Theta),

      split_by_number(Vectors, OtherSingles, OtherMultiples, OtherThetas),

      append(Single, OtherSingles, Singles),
      append(Multiple, OtherMultiples, Multiples),
      append(Theta, OtherThetas, Thetas).

split_by_number(_, [], [], []).


split_deltas([Single|OtherSingles], Last, Deltas,
  [Single|RemainingSingles]) :-

      append(Last, [_], Single),

      split_deltas(OtherSingles, Last, Deltas, RemainingSingles).

split_deltas([Single|OtherSingles], _, [Single|Deltas], RemainingSingles) :-
```

```
      append(Last, [_], Single),

      split_deltas(OtherSingles, Last, Deltas, RemainingSingles).

split_deltas(_, _, [], []).


%
% Zu Psi gehoerige Koeffizienten berechnen
%

calc_psis([Last|Psis], Last, Counter, SumPsi) :-

      append(Bools, [Var], Last),

      NewCounter is Counter + Var,

      calc_psis(Psis, Last, NewCounter, SumPsi).

calc_psis([Psi|Psis], _, Counter, SumPsi) :-

      append(_, [Var], Psi),

      Var is Counter + 1,

      NewCounter is Counter + Var,

      calc_psis(Psis, Psi, NewCounter, SumPsi).

calc_psis(_, _, SumPsi, SumPsi).


%
% Konstante berechnen
%

calc_sigma(0, 0, Sum, 0).

calc_sigma(_, _, Sum, Sigma) :-

      Sigma is 2 * Sum + 1.


%
% Zu Delta gehoerige Koeffizienten berechnen
%

overlap([_], [_]) :-

      !, fail.
```

```
overlap([1|_], [1|_]).

overlap([_|Other1], [_|Other2]) :-

      overlap(Other1, Other2).


sum_of_relevant_psis(Delta, [Psi|Psis], Coefficient + Sum) :-

      overlap(Delta, Psi),

      append(_, [Coefficient], Psi),

      sum_of_relevant_psis(Delta, Psis, Sum).

sum_of_relevant_psis(Delta, [_|Psis], Sum) :-

      sum_of_relevant_psis(Delta, Psis, Sum).

sum_of_relevant_psis(_, _, 0).


calc_deltas([Delta|Deltas], Psis, Sigma) :-

      sum_of_relevant_psis(Delta, Psis, Sum),

      append(_, [Coefficient], Delta),

      Coefficient is Sigma - Sum,

      calc_deltas(Deltas, Psis, Sigma).

calc_deltas(_, _, _).


%
% Zu Theta gehoerige Koeffizienten bestimmen
%

set_thetas([Theta|Thetas], SigmaAndOne) :-

      append(_, [SigmaAndOne], Theta),

      set_thetas(Thetas, SigmaAndOne).

set_thetas(_, _).


%
% Ablaufsteuerung
```

```
%

prepare_values(gruppen, [Group|Groups],
  [[Coefficients, Sigma, MaxSum]|OtherValues]) :-

      member(DummyAlternative, Group),

      length(DummyAlternative, AlternativeLength),
      length(Coefficients, AlternativeLength),
      length(Vectors, AlternativeLength),

      append(Group, [Coefficients], CGroup),

      build_vectors(CGroup, Vectors),

      split_by_number(Vectors, Singles, Multiples, Thetas),

      msort(Singles, SortedSingles),
      msort(Multiples, SortedMultiples),

      split_deltas(SortedSingles, [], Deltas, RemainingSingles),

      append(SortedMultiples, RemainingSingles, Psis),

      calc_psis(Psis, [], 0, SumPsis),

      length(Psis, PsiLength),
      length(Deltas, DeltaLength),

      calc_sigma(PsiLength, DeltaLength, SumPsis, Sigma),

      calc_deltas(Deltas, Psis, Sigma),

      SigmaAndOne is Sigma + 1,
      set_thetas(Thetas, SigmaAndOne),

      add_list(Coefficients, MaxSum),

      prepare_values(gruppen, Groups, OtherValues),

      !.

prepare_values(gruppen, _, []).


% prepare_values(gruppen, [[[0, 1, 0, 1], [1, 0, 1, 0]]], X).
```

A.2 Testprogramm

Die hier dargestellten Programmtexte wurden zur empirischen Auswertung der Algorithmen in Abschnitt 3.7 verwendet.

A.2.1 `test.data` und `test_gruppen.data`

Um nicht generalisierbare Laufzeiten aufgrund einer speziellen Struktur der Alternativen auszuschließen, werden jeweils nur Systeme von Alternativen untersucht, die den gesamten Lösungsraum abdecken. Wegen des binären Aufbaus der Alternativen werden somit nur Systeme von 2^n Alternativen mit n Variablen betrachtet.

Für eine Umsetzung durch Gruppen ist dies bezüglich einer möglichen Reduktion der Worst case. Die folgende Tabelle stellt die erreichbare Reduktion dar[1]:

Alternativen	Gruppen	Reduktion
2	2	0%
4	3	25%
8	5	37.5%
16	8	50%
32	12	62.5%

Tabelle A.1: Erreichbare Gruppen-Reduktion im Worst case

Die beobachteten Daten lassen folgenden Zusammenhang zwischen Anzahlen von Alternativen 2^n und Gruppen g vermuten:

$$g = \frac{1}{2}n^2 - \frac{1}{2}n + 2 \tag{A.1}$$

Da diese Untersuchungen den Worst case behandeln, sollte die Anwendung der Umsetzung durch Gruppen auf reale Probleme zu einer sehr viel besseren Performanz führen.

Bei den in Abschnitt 3.7 durchgeführten Testläufen müssen die Alternativen über die Datei `test.data` bereitgestellt werden. Die gruppierten Alternativen enthält die Datei `test_gruppen.data`.

In der Datei `test.data` werden die expliziten Alternativen als Fakten abgelegt. Für einen Testlauf mit vier Alternativen pro Dimension würde dies beispielsweise folgendermaßen aussehen:

[1] Die Ergebnisse wurden mittels vollständiger Suche bestimmt.

```
alternative([0, 0]).
alternative([0, 1]).
alternative([1, 0]).
alternative([1, 1]).
```

Durch das Prädikat `combination` in der Datei `test_gruppen.data` wird die Gruppenaufteilung bestimmt. Für die in Abbildung 3.11 gezeigten Testläufe mit 16 Alternativen wurde folgende Aufteilung verwendet:

```
combination(8,
  [[[0, 0, 0, 0]],
  [[1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 1, 0], [0, 0, 0, 1]],
  [[1, 0, 0, 1], [0, 1, 0, 1], [0, 0, 1, 1]],
  [[1, 0, 1, 0], [0, 1, 1, 0]],
  [[1, 0, 1, 1], [0, 1, 1, 1]],
  [[1, 1, 0, 0]],
  [[1, 1, 1, 0], [1, 1, 0, 1]],
  [[1, 1, 1, 1]]]).
```

Für die in Abbildung 3.12 gezeigten Testläufe mit 32 Alternativen wurde die folgende Aufteilung verwendet:

```
combination(12,
  [[[0, 0, 0, 0, 0]],
  [[1, 0, 0, 0, 0], [0, 1, 0, 0, 0], [0, 0, 1, 0, 0],
    [0, 0, 0, 1, 0], [0, 0, 0, 0, 1]],
  [[1, 0, 0, 0, 1], [0, 1, 0, 0, 1], [0, 0, 1, 0, 1], [0, 0, 0, 1, 1]],
  [[1, 0, 0, 1, 0], [0, 1, 0, 1, 0], [0, 0, 1, 1, 0]],
  [[1, 0, 0, 1, 1], [0, 1, 0, 1, 1], [0, 0, 1, 1, 1]],
  [[1, 0, 1, 1, 0], [1, 0, 1, 0, 1], [0, 1, 1, 0, 0]],
  [[1, 0, 1, 0, 0], [0, 1, 1, 1, 0], [0, 1, 1, 0, 1]],
  [[1, 1, 0, 0, 0], [0, 1, 1, 1, 1]],
  [[1, 1, 0, 1, 1], [1, 0, 1, 1, 1]],
  [[1, 1, 1, 0, 0], [1, 1, 0, 1, 0], [1, 1, 0, 0, 1]],
  [[1, 1, 1, 1, 0], [1, 1, 1, 0, 1]],
  [[1, 1, 1, 1, 1]]]).
```

A.2.2 `test.pl`

Der nachfolgende Programmtext realisiert die empirische Auswertung der vorgestellten Algorithmen. Die Ergebnisse sind in Abschnitt 3.7 beschrieben.

```
%
% TEST.PL
%
```

```
% Auswertung der verschiedenen Algorithmen:
%
%   run(Algorithmus, Level, Wiederholungen)
%


:- [-tools].


:- wflags(200).


:- dynamic time/1,
:- dynamic best/1.


load_algorithm(backtracking) :-

      [-'test.data'],

      retractall(best(_)).

load_algorithm(ungleichungen) :-

      [-'test.data'],

      [-umsetzung_ungleichungen],
      [-kopplung_ungleichungen].

load_algorithm(gleichungen) :-

      [-'test.data'],

      [-umsetzung_gleichungen],
      [-kopplung_gleichungen].

load_algorithm(gruppen) :-

      [-'test_gruppen.data'],

      [-umsetzung_gruppen],
      [-kopplung_gleichungen].


get_alternatives(gruppen, GroupedAlternatives, AlternativeLength) :-

      combination(_, GroupedAlternatives),

      member(Group, GroupedAlternatives),
      member(Alternative, Group),
```

```
      length(Alternative, AlternativeLength).

get_alternatives(Algorithm, Alternatives, AlternativeLength) :-

      findall(Alternative, alternative(Alternative), Alternatives),

      alternative(DummyAlternative),

      length(DummyAlternative, AlternativeLength).


group_variables(_, [], []).

group_variables(GroupLength, FlatVariables, [GroupedVariables|Other]) :-

      length(GroupedVariables, GroupLength),

      append(GroupedVariables, OtherFlatVariables, FlatVariables),

      group_variables(GroupLength, OtherFlatVariables, Other).


prepare_variables(AlternativeLength, Level, VariableNumber, Variables,
  GivenVariables, GroupedVariables, GroupedGivenVariables) :-

      VariableNumber is Level * AlternativeLength,

      length(Variables, VariableNumber),
      length(GivenVariables, VariableNumber),

      Variables :: [0, 1],
      GivenVariables :: [0, 1],

      group_variables(AlternativeLength, Variables, GroupedVariables),
      group_variables(AlternativeLength, GivenVariables,
        GroupedGivenVariables).


split_variables([1|OtherValues], [Variable|OtherVariables],
  [Variable|OtherToBeOne], OtherToBeZero) :-

      split_variables(OtherValues, OtherVariables,
        OtherToBeOne, OtherToBeZero).

split_variables([0|OtherValues], [Variable|OtherVariables],
  OtherToBeOne, [Variable|OtherToBeZero]) :-

      split_variables(OtherValues, OtherVariables, OtherToBeOne,
        OtherToBeZero).

split_variables(_, _, [], []).
```

```
connect_problems(GivenVariables, Variables, VariableNumber, ToMinimize) :-

      split_variables(GivenVariables, Variables, ToBeOne, ToBeZero),

      length(ToBeOne, ToBeOneLength),

      ToMinimize :: 0..VariableNumber,

      build_addition(ToBeOne, AddedOne),
      build_addition(ToBeZero, AddedZero),

      AddedOne + ToMinimize #= ToBeOneLength + AddedZero,

      !.


choose_alternative([First|Other]) :-

      alternative(First),

      choose_alternative(Other).

choose_alternative([]).


get_actual_best(Minimum) :-

      best(Minimum),

      !.


actualize_solution(Minimum) :-

      retract(best(_)),
      assert(best(Minimum)),

      !.


optimize_by_iteration(Variables, ToMinimize) :-

      choose_alternative(Variables),

      get_actual_best(Minimum),

      ToMinimize #< Minimum,

      actualize_solution(ToMinimize),
```

```
      fail.

optimize_by_iteration(_, _).


all_sublists_equal([Sublist|Other], Sublist) :-

      all_sublists_equal(Other, Sublist).

all_sublists_equal([Sublist|[]], Sublist).


store_time(GroupedVariables, Time) :-

      all_sublists_equal(GroupedVariables, _),

      assert(time(Time, special)),

      !.

store_time(_, Time) :-

      assert(time(Time, normal)).


establishment(Algorithm, Values, [GroupedVariables|OtherGroupedVariables],
  SwitchVariables) :-

      establish_alternatives(Algorithm, GroupedVariables, Values,
        NewSwitchVariables, _),

      establishment(Algorithm, Values, OtherGroupedVariables,
        OtherSwitchVariables),

      append(NewSwitchVariables, OtherSwitchVariables, SwitchVariables).

establishment(_, _, _, []).


check(backtracking, Level) :-

      alternative(DummyAlternative),

      length(DummyAlternative, AlternativeLength),

      prepare_variables(AlternativeLength, Level, VariableNumber, Variables,
        GivenVariables, GroupedVariables, GroupedGivenVariables),

% Zu eingeschraenktem Test aktivieren:
all_sublists_equal(GroupedGivenVariables, _),
```

```
      !,

      labeling(GivenVariables, 0, most_constrained, indomain),

      connect_problems(GivenVariables, Variables, VariableNumber, ToMinimize),

      retractall(best(_)),
      assert(best(100000)),

      cputime(TimeA),
      optimize_by_iteration(GroupedVariables, ToMinimize),
      cputime(TimeB),

      Time is TimeB - TimeA,

      store_time(GroupedGivenVariables, Time),

      write('.'),

      fail.

check(Algorithm, Level) :-

      Algorithm \= backtracking,

      get_alternatives(Algorithm, Alternatives, AlternativeLength),

      prepare_values(Algorithm, Alternatives, Values),

      prepare_variables(AlternativeLength, Level, VariableNumber, Variables,
        GivenVariables, GroupedVariables, GroupedGivenVariables),

      establishment(Algorithm, Values, GroupedVariables, SwitchVariables),

      append([ToMinimize], SwitchVariables, AllVariables),

% Zu eingeschraenktem Test aktivieren:
all_sublists_equal(GroupedGivenVariables, _),

      !,

      labeling(GivenVariables, 0, most_constrained, indomain),

      connect_problems(GivenVariables, Variables, VariableNumber, ToMinimize),

      cputime(TimeA),
      min_max(labeling(AllVariables, 0, most_constrained, indomain),
        ToMinimize),
      cputime(TimeB),
```

```
      Time is TimeB - TimeA,

      store_time(GroupedGivenVariables, Time),

      write('.'),

      fail.


prepare_evaluation(backtracking, CombiLength, 'BT', 'Alternatives') :-

      findall(A, alternative(A), Alternatives),

      length(Alternatives, CombiLength).

prepare_evaluation(ungleichungen, CombiLength, 'NEQ', 'Alternatives') :-

      findall(A, alternative(A), Alternatives),

      length(Alternatives, CombiLength).

prepare_evaluation(gleichungen, CombiLength, 'EQ', 'Alternatives') :-

      findall(A, alternative(A), Alternatives),

      length(Alternatives, CombiLength).

prepare_evaluation(gruppen, CombiLength, 'GRP', 'Groups') :-

      combination(_, Combination),

      length(Combination, CombiLength).


write_list([[Time, Status]|Other]) :-

      write_list(Other),

      write(Time),
      write(' '),
      writeln(Status).

write_list(_).


repeat_checks(Algorithm, Level, _) :-

      check(Algorithm, Level).

repeat_checks(Algorithm, Level, Counter) :-
```

```
      Counter > 1,

      NewCounter is Counter - 1,

      repeat_checks(Algorithm, Level, NewCounter).

repeat_checks(Algorithm, Level, Counter) :-

      prepare_evaluation(Algorithm, CombiLength, AlgorithmName,
        AlgorithmProc),

      findall([T, S], time(T, S), Times),

      msort(Times, SortedTimes),

      length(SortedTimes, TimeListLength),

      Index :: 1..100000,

      indomain(Index),

      sprintf(FileName, "out%s_%dx%d_%d",
        [AlgorithmName, CombiLength, Level, Index]),

      not(exists(FileName)),

      tell(FileName),

      write('Used Algorithm: '), writeln(AlgorithmName),
      write(AlgorithmProc), write(': '), writeln(CombiLength),
      write('Dimension: '), writeln(Level),
      writeln(TimeListLength),

      write_list(Times),
      write_list(SortedTimes),

      told,

      findall(Time, time(Time, _), OnlyTimes),

      add_list(OnlyTimes, Sum),
      Average is Sum / TimeListLength,

      nl, nl, write(Average), writeln(' ms.').


run(Algorithm, Level, Counter) :-

      retractall(time(_)),

      load_algorithm(Algorithm),
```

```
      repeat_checks(Algorithm, Level, Counter),

      nl,

      halt.

run(Algorithm, Level) :-

      run(Algorithm, Level, 1).


:- nl,
   writeln('Start with -> run(ALGORITHM, LEVEL, REPETATIONS)'),
   writeln('        or -> run(ALGORITHM, LEVEL)'),
   nl.
```

A.2.3 paint

Durch das folgende Skript in TCL/TK[2] wurden die Bilder in Abbildung 3.8 unter Verwendung der Dokumentations-Dateien von `test.pl` erstellt. Der Name der von `test.pl` erzeugten Datei muß beim Aufruf als Argument mitgegeben werden.

So führt der Aufruf von

```
> paint outGRP_8x3_1
```

zu der in Abbildung A.1 gezeigten Darstellung. Die Dokumentations-Datei enthält dabei einen Testlauf einer Problemstellung mit acht Alternativen und drei Dimensionen, die mittels einer Umsetzung durch Gruppen gelöst wurde.

Die beiden Grafiken dokumentieren die Laufzeiten zur Bestimmung des Optimums. In der linken Grafik sind diese nach den untersuchten Binärzahlen geordnet. In der rechten Grafik sind die Zeiten dagegen der Länge der Zeit nach sortiert. Über die beiden Buttons unter den Grafiken können die Grafiken als Postscript-Dateien abgelegt werden.

Das `paint`-Tool enthält weiterhin die Möglichkeit, speziell ausgezeichnete Laufzeiten der Dokumentations-Datei hervorzuheben. Dies geschieht durch die Aktivierung des „Show Specials“-Buttons.

Dies wird genutzt, um die Tauglichkeit von Repräsentanten für die höherdimensionierten Testläufe des Abschnitts 3.7.4 zu bestimmen. Als Repräsentanten werden solche Binärzahlen gewählt, bei denen die Belegungen aller Dimensionen

[2] Eine Einführung in TCL/TK liefert [Ousterhout94].

identisch sind. Im Beispiel von Abbildung 3.7 würden nur folgende Binärzahlen getestet:

X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8	X_9	X_{10}
0	0	0	0	0	0	0	0	0	0
0	1	0	1	0	1	0	1	0	1
1	0	1	0	1	0	1	0	1	0
1	1	1	1	1	1	1	1	1	1

Tabelle A.2: Repräsentanten für Abbildung 3.7

Abbildung A.2 zeigt Repräsentanten innerhalb des Testlaufes von Abbildung A.1. Der Durchschnitt der Zeiten bei den Repräsentanten ist in allen getesteten Fällen ungefähr gleich dem Durchschnitt aller Zeiten. Gegenüber Lösungen durch vollständige Enumeration ergeben sich bei den Constraint-Lösungen allerdings Laufzeiten, die etwas über dem eigentlichen Durchschnitt liegen.

Nachfolgend ist der Programmtext des Tools aufgeführt:

```
#!/usr/bin/wish -f


if {$argc != 1} {

    puts "Bitte Filenamen als Option angeben!"
    exit 1

}


set Specials 0


. configure -bg #677


frame .f -bg #677

pack .f -side left -padx 10m -pady 2m

frame .f1 -relief groove -bd 2 -bg #677
frame .f2 -relief groove -bd 2 -bg #677

pack .f1 .f2 -side left -padx 2m -pady 2m
```

```
message .f.m -width 4c -justify center -relief sunken -borderwidth 2 \
  -bg #fff -fg #000
checkbutton .f.special -text "  Show Specials" -bg #677 -fg #dff \
  -selector #ff0 -variable Specials -command special_change
button .f.exit -text "Exit" -bg #677 -fg #dff -command "exit 0"

pack .f.m .f.special .f.exit -ipadx 5m -ipady 2m -pady 3m -side top


canvas .f1.c1 -width 190 -height 190 -bg #fff
button .f1.postscript -text "Make Time-EPS" -bg #677 -fg #dff \
  -command "ps .f1.c1"

pack .f1.c1 .f1.postscript -side top -fill x -pady 1m -padx 2m


canvas .f2.c2 -width 190 -height 190 -bg #fff
button .f2.postscript -text "Make Value-EPS" -bg #677 -fg #dff \
  -command "ps .f2.c2"

pack .f2.c2 .f2.postscript -side top -fill x -pady 1m -padx 2m


proc special_change {} {

    global R1
    global R2
    global Specials
    global SpecialLines1
    global SpecialLines2

    if {$Specials == 1} {
        set BG #999
        set FG #ff0
    } {
        set BG #fff
        set FG #000
    }

    .f1.c1 itemconfigure $R1 -fill $BG
    .f2.c2 itemconfigure $R2 -fill $BG

    foreach LineID $SpecialLines1 {
        .f1.c1 itemconfigure $LineID -fill $FG
    }

    foreach LineID $SpecialLines2 {
        .f2.c2 itemconfigure $LineID -fill $FG
    }

}
```

```
proc ps {Window} {

    global argv

    if {$Window == ".f1.c1"} {
        set FileId [open "[lindex $argv 0]_time.eps" w]
    } {
        set FileId [open "[lindex $argv 0]_value.eps" w]
    }

    puts $FileId [$Window postscript]

    close $FileId

}


set R1 [.f1.c1 create rectangle 10 10 180 180 -fill #fff]
set R2 [.f2.c2 create rectangle 10 10 180 180 -fill #fff]


for {set i 0} {$i <= 150} {incr i} {

    set Counter($i) 0
    set Color($i) normal
    set Sum($i) 0

}


set SpecialLines1 {}
set SpecialLines2 {}

set FileId [open [lindex $argv 0] r]

gets $FileId Desc1
gets $FileId Desc2
gets $FileId Desc3

.f.m configure -text "$Desc1\n$Desc2\n$Desc3"

gets $FileId Length

set XCompress [expr 150.0 / ($Length - 1)]

set Max 0
set Number 0
set X 0
```

```
while {$Number < $Length} {

    gets $FileId Input

    set Value [lindex $Input 0]
    set Status [lindex $Input 1]

    if {$Status == "special"} {
        set Color($X) special
    }

    if {$Value > $Max} {
        set Max $Value
    }

    set X [expr round($Number * $XCompress)]

    incr Counter($X)

    set Sum($X) [expr $Sum($X) + $Value]

    incr Number

}

set YCompress [expr 150.0 / $Max]

for {set i 0} {$i <= 150} {incr i} {

    if {$Counter($i) > 0} {

        set Y [expr round(170 - $YCompress * $Sum($i) / $Counter($i))]
        set X [expr 150 - $i + 20]

        set LineID [.f1.c1 create line $X 170 $X $Y -fill #000]

        if {$Color($i) == "special"} {
            lappend SpecialLines1 $LineID
        }

    }

}

for {set i 0} {$i <= 150} {incr i} {

    set Counter($i) 0
    set Color($i) normal
    set Sum($i) 0

}
```

```
set Number 0
set X 0

while {$Number < $Length} {

    gets $FileId Input

    set Value [lindex $Input 0]
    set Status [lindex $Input 1]

    if {$Status == "special"} {
        set Color($X) special
    }

    set X [expr round($Number * $XCompress)]

    incr Counter($X)

    set Sum($X) [expr $Sum($X) + $Value]

    incr Number

}

for {set i 0} {$i <= 150} {incr i} {

    if {$Counter($i) > 0} {

        set Y [expr round(170 - $YCompress * $Sum($i) / $Counter($i))]
        set X [expr 150 - $i + 20]

        set LineID [.f2.c2 create line $X 170 $X $Y -fill #000]

        if {$Color($i) == "special"} {
            lappend SpecialLines2 $LineID
        }

    }

}

close $FileId
```

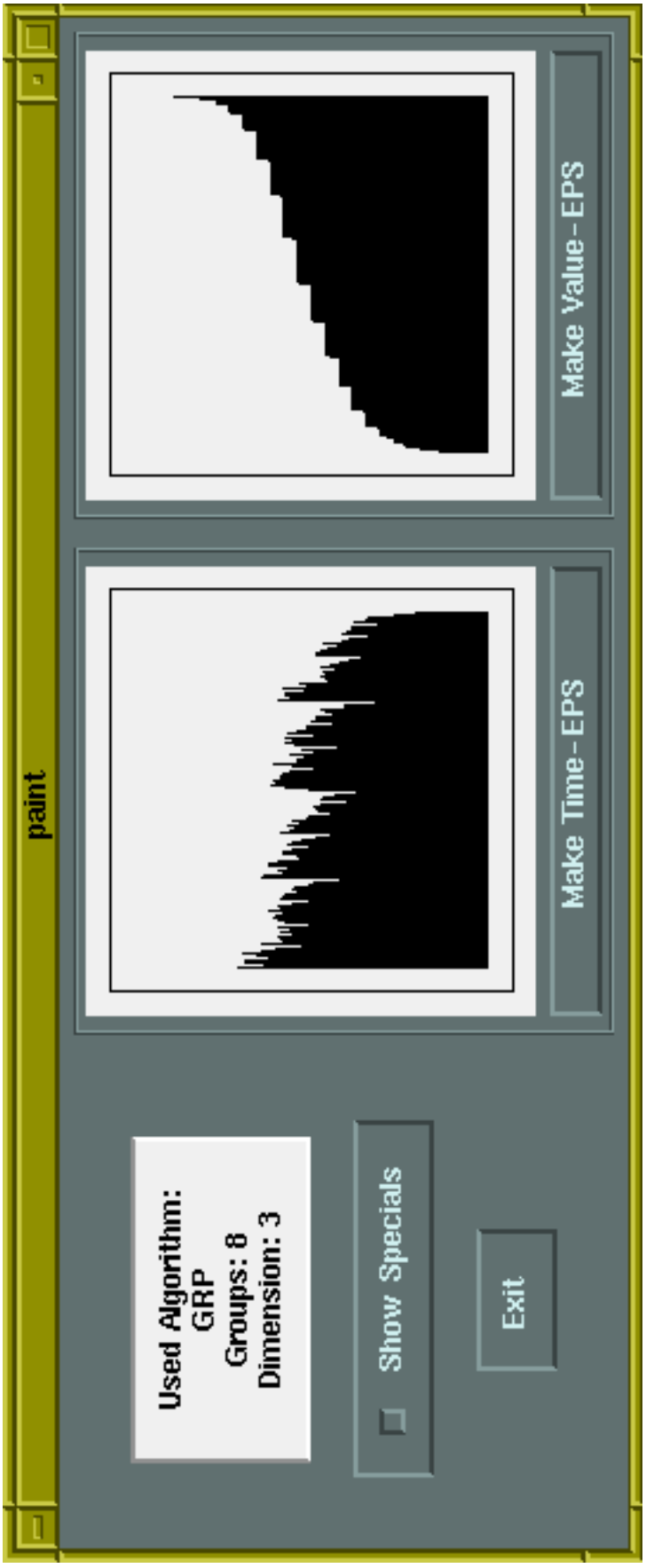


Abbildung A.1: Beispiel einer Visualisierung mittels `paint`

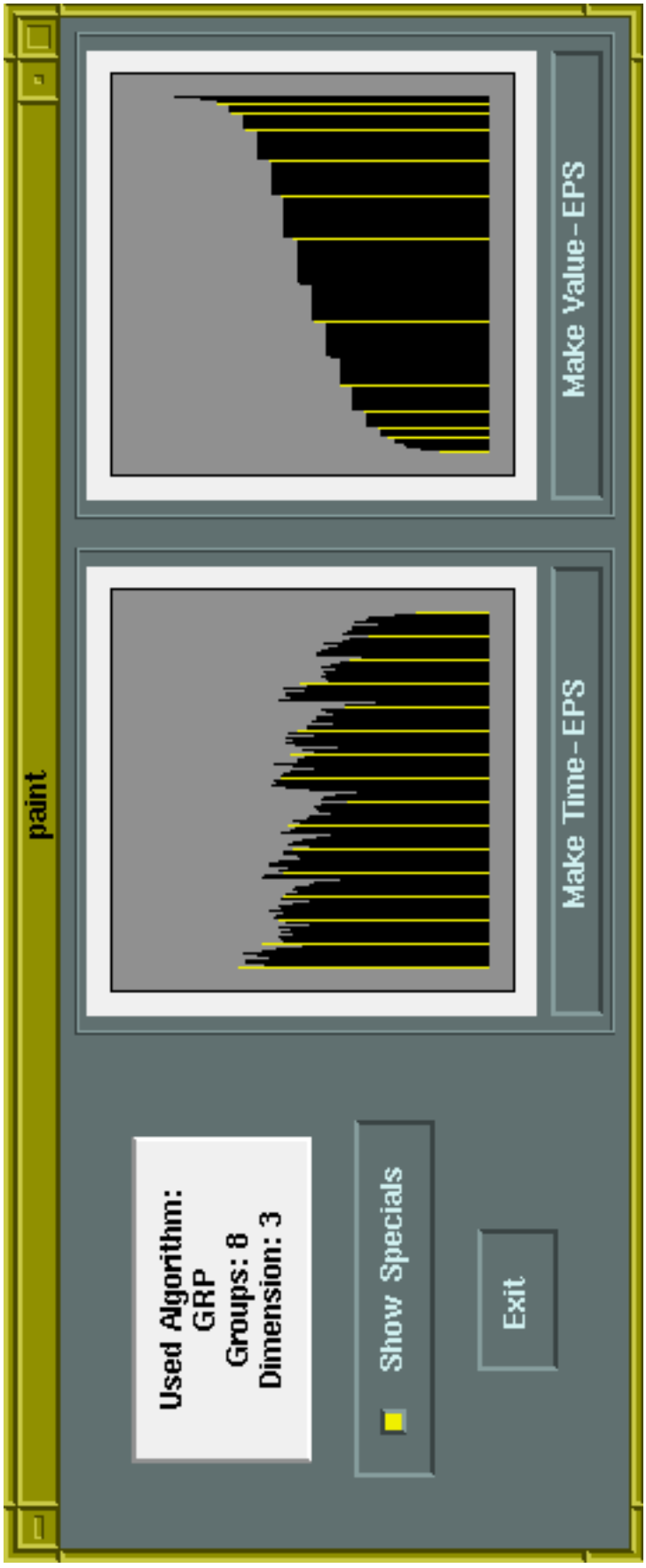


Abbildung A.2: Visualisierung repräsentativer Testläufe

Literaturverzeichnis

[Balas85] Egon Balas. *Disjunctive programming and a hierarchy of relaxation for discrete optimization problems.* SIAM journal on algebraic and discrete methods, 6(3), 1985.

[Barth96] Peter Barth, Alexander Bockmayr und Thomas Kasper. *Pseudo-Boolean Constraint Logic Programming.* Workshop Deklarative Constraint-Programmierung zur KI'96, GMD-Studien, Nr. 297, 1996.

[Beringer93] Henri Beringer und Bruno de Backer. *Satisfiability of Boolean formulas over linear constraints.* Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence, 1993.

[Beierle93] C. Beierle und G. Meyer. *Handbook for the Logic Programming Language PROTOS-L.* WISPRO Report 2/93, IBM Informationssysteme GmbH, WZH, 1993.

[Bessière95] Christian Bessière, Eugene C. Freuder und Jean-Charles Régin. *Using Inference to Reduce Arc Consistency Computation.* Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence, 1995.

[Bruhn90] Manfred Bruhn. *Marketing: Grundlage für Studium und Praxis.* Sonderausgabe, Gabler Verlag, Wiesbaden, 1990.

[Breitinger94a] Silvia Breitinger und Hendrik Lock. *Improving Search for Job-Shop Scheduling with CLP(FD).* Programming Language Implementation and Logic Programming, Springer-Verlag, 1994.

[Breitinger94b] Silvia Breitinger und Hendrik Lock. *Modelling and Scheduling in CLP(FD).* Proceedings of the Second International Conference on the Practical Application of Prolog, 1994.

[Cheng96] B. M. W. Cheng, J. H. M. Lee und J. C. K. Wu. *Speeding Up Constraint Propagation By Redundant Modeling.* Proceedings of

the Second International Conference on Principles and Practice of Constraint Programming — CP96, 1996.

[Dechter90] Rina Dechter. *Enhancement Schemes for Constraint Processing: Backjumping, Learning, and Cutset Decomposition.* Artificial Intelligence, Volume 41, 1990.

[ECRC93] ECRC. *Eclipse User Manual.* International Computers Limited and ECRC GmbH, 1993.

[Freuder96] Eugene C. Freuder. *Constraint Programming Position Paper.* ACM Workshop on Strategic Directions in Computing Research — Constraint Programming, 1996.

[Frühwirth95] Thom Frühwirth. *Constraint Handling Rules.* Constraint Programming: Basics and Trends, Springer LNCS 910, 1995.

[Geske93] Ulrich Geske. *Prolog: Grundlagen — Programmiermethoden — Standards.* Akademie Verlag, 1993.

[Guesgen92] Hans Werner Guesgen und Joachim Hertzberg. *A Perspective of Constraint-Based Reasoning.* Springer-Verlag, 1992.

[Heintze92] Nevin C. Heintze, Joxan Jaffar, Spiro Michaylov, Peter J Stuckey und Roland H.C. Yap. *The CLP(R) Programmer's Manual*, 1992.

[Hentenryck89] Pascal van Hentenryck. *Constraint Satisfaction in Logic Programming*, MIT Press, 1989.

[Jaffar94] Joxan Jaffar und Michael J. Mahler. *Constraint Logic Programming: A Survey.* Journal of Logic Programming, Volume 19/20, 1994.

[Mitra94] G. Mitra, C. Lucas, S. Moody und E. Hadjiconstantinou. *Tools for reformulating logical forms into zero-one mixed integer programs.* European Journal of Operational Research 72, 1994.

[Nareyek95] Alexander Nareyek. *Modellierung der Disjunktion in der constraintlogischen Programmierung.* Arbeitspapiere der GMD, Nummer 932, 1995.

[Nareyek96a] Alexander Nareyek und Ulrich Geske. *Representation of Relations over Linear Constraints.* Second International Conference on Principles and Practice of Constraint Programming — CP96, Workshop on Constraint Programming Applications, 1996.

[Nareyek96b] Alexander Nareyek und Ulrich Geske. *Behandlung expliziter Alternativen in CLP(FD).* Workshop Deklarative Constraint-Programmierung zur KI'96, GMD-Studien, Nummer 297, 1996.

[Ousterhout94] John K. Ousterhout. *Tcl and the Tk toolkit.* Addison-Wesley, 1994.

[Pesant96] Gilles Pesant und Michel Gendreau. *A View of Local Search in Constraint Programming.* Proceedings of the Second International Conference on Principles and Practice of Constraint Programming — CP96, 1996.

[Simonis95] Helmut Simonis, *The CHIP System and Its Applications*, Proceedings of the First International Conference on Principles and Practice of Constraint Programming — CP95, 1995.

[Smolka94] Gert Smolka. *A Calculus for Higher-Order Concurrent Constraint Programming with Deep Guards.* DFKI Research Report RR-94-03, 1994.

[Syslo83] Maciej M. Syslo, Narsingh Deo und Janusz S. Kowalik, *Discrete Optimization Algorithms*, Prentice-Hall, 1983.