Albert-Ludwigs-Universität Freiburg

Institut für Informatik

Lehrstuhl für Rechnerarchitektur

Prof. Dr. Bernd Becker

Ganzzahlige Lineare Optimierung und Separierung mit binären Entscheidungsdiagrammen

Diplomarbeit

Ralf Wimmer

22. November 2004

Danksagung

Ich möchte mich an dieser Stelle bedanken bei

- meinen Eltern, die mir auf meinem gesamten Lebensweg beigestanden haben und mir dieses Studium erst möglich gemacht haben.
- meiner Lerngruppe, die mich seit dem ersten Semester auch durch schwierige Phasen des Studiums begleitet hat. Ohne sie wäre mein Studium sicherlich nicht so erfolgreich verlaufen.
- Prof. Becker und dem ganzen LRA-Team für die gute Zusammenarbeit während der letzten vier Jahre.
- Marc Herbstritt für die vielen hilfreichen Tipps zu den Variablenordnungen für die BDDs.
- Markus Behle, dem die zahlreichen Bugs in meinem Code oft zu schaffen gemacht haben, und Fritz Eisenbrand vom Max-Planck-Institut Saarbrücken. Ein großer Teil dieser Arbeit ist in enger Zusammenarbeit mit ihnen entstanden.
- den Korrekturlesern Manuel Bitzer und Heiko Falk.

Erklärung

Hiermit erkläre ich, daß ich die vorliegende Arbeit selbständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt und alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder unveröffentlichten Schriften entnommen wurden, als solche kenntlich gemacht habe. Außerdem erkläre ich, daß diese Diplomarbeit nicht, auch nicht auszugsweise, bereits für eine andere Prüfung angefertigt wurde.

Ralf Wimmer
Freiburg im November 2004.

Inhaltsverzeichnis

1. Einleitung

Optimierungprobleme begegnen uns ständig in unserem täglichen Leben: sei es, daß wir uns überlegen, wie wir am schnellsten von einem Ort zum anderen kommen, oder daß ein Geschäftsmann den Gewinn seiner Fabrik maximieren will. Insbesondere der Informatiker hat Optimierungsprobleme zu lösen: was soll beim Entwurf eines eingebetteten Systems als Hardware, was als Software realisiert werden, so daß die maximale Ausführungszeit einer Anwendung nicht überschritten werden und die Kosten minimal sind?

Grob lassen sich die Optimierungsprobleme in zwei Klassen aufteilen: die einfachen (d. h. in polynomieller Zeit lösbaren) und die schwierigen (d. h. NP-vollständigen) Probleme. Viele der praktisch wichtigen Probleme gehören zur Klasse der schwierigen Probleme. Deshalb ist es wichtig, möglichst effiziente Algorithmen dafür zu finden.

Das Thema dieser Diplomarbeit ist die ganzzahlige lineare Optimierung. Dabei handelt es sich um ein komplexitätstheoretisch schwieriges Problem, auf das sich viele in der Praxis vorkommenden Probleme zurückführen lassen.

Das Ziel dieser Arbeit ist die Untersuchung, wie binäre Entscheidungsdiagramme verwendet werden können, um die Lösung von ganzzahligen linearen Programmen zu beschleunigen. Die Idee so vorzugehen, geht zurück auf einen Artikel von Lai et. al. [10] aus dem Jahre 1994, in dem ein Verfahren mit EVBDDs beschrieben wird. Der Schwerpunkt dieser Diplomarbeit liegt darin zu zeigen, wie man die klassischen Branch & Cut-Verfahren mit der BDD-Technologie kombinieren kann.

Diese Arbeit ist deshalb folgendermaßen aufgebaut:

- Im nächsten Kapitel gehen wir auf die Grundlagen der ganzzahligen linearen Optimierung ein. Nach der formalen Definition des Problems untersuchen wir dessen Komplexität und den Zusammenhang mit einem einfacheren Problem, das dadurch entsteht, daß wir die Forderung an die Ganzzahligkeit der Variablen fallenlassen. Außerdem werden wir den Fall untersuchen, wann ein ganzzahliges lineares Programm effizient gelöst werden kann.

- Das dritte Kapitel ist dem zweiten Stützpfeiler unseres Verfahrens, den binären Entscheidungsdiagrammen (BDDs), gewidmet. Wir zeigen, wie BDDs Boolesche Funktionen darstellen und präsentieren die für uns wichtigsten Algorithmen für BDDs.

- Ein erstes Lösungsverfahren, das ausschließlich auf BDDs beruht und die klassischen Verfahren noch außer Acht läßt, stellen wir im vierten Kapitel vor. Nach der Beschreibung des Verfahrens gehen wir auf die Probleme und einige Lösungsmöglichkeiten ein, bevor wir experimentelle Ergebnisse präsentieren.

- Das fünfte Kapitel behandelt die klassischen Verfahren zur Lösung von ganzzahligen linearen Programmen. Das verwendete Verfahren beruht auf der Kombination eines Branch & Bound-Algorithmus mit einem Verfahren, das auf der Erzeugung zusätzlicher Constraints beruht. Diese beiden Verfahren werden wir zuerst getrennt betrachten und dann zeigen, wie sie kombiniert werden können.

- Im sechsten Kapitel folgt schließlich das Verfahren, das unter Verwendung von BDDs solche zusätzlichen Constraints erzeugt, wie sie im fünften Kapitel benötigt werden. Wir beschreiben zuerst, wie die BDDs aufgebaut werden. Danach stellen wir zwei Verfahren vor, mit denen wir aus dem BDD einen zusätzlichen Constraint erzeugen können. Da die Constraints noch nicht so stark wie möglich sind, zeigen wir, wie man die Constraints verstärken kann. Zum Abschluß dieses Kapitels geben wir einige Hinweise, wie wir das Verfahren implementiert haben. Experimentelle Ergebnisse bilden den Abschluß des Kapitels.

- Im siebten Kapitel werden die Resultate dieser Arbeit nochmals kurz zusammengefaßt. Es werden Punkte angegeben, an denen noch Forschungsbedarf besteht, und Ideen präsentiert, die es sich zu verfolgen lohnen könnte.

2. ILP – Was ist das?

In diesem Kapitel werden die theoretischen Grundlagen der ganzzahligen linearen Programmierung vorgestellt. Am Anfang steht die Definition der Problemstellung. Nach der Untersuchung der Komplexität des Problems gehen wir in einem weiteren Abschnitt auf die geometrische Dimension des Problems und das Verhältnis zur sogenannten LP-Relaxierung ein. Eine besondere Rolle spielen später sogenannte total unimodulare Matrizen. Diesen wird ein weiterer Abschnitt gewidmet sein. Zum Abschluß des Kapitels untersuchen wir eine Methode namens Lagrange-Relaxierung zur Berechnung unterer Schranken der Lösung.

2.1. Die Problemstellung

Definition 2.1 *Ein **ganzzahliges lineares Programm** (ILP[1]) ist gegeben durch*

- *eine Matrix* $A \in \mathbb{Z}^{m \times n}$
- *einen Vektor* $b \in \mathbb{Z}^m$,
- *einen Vektor* $c \in \mathbb{Z}^n$.

Gesucht ist ein Vektor $x \in \mathbb{Z}^n$, *so daß gilt*

$$\begin{aligned} &Ax \leq b \\ &c^T x = \min\{c^T y \,|\, y \in \mathbb{Z}^{n \times 1}, Ay \leq b\} \end{aligned}$$

Die linearen Ungleichungen $Ax \leq b$ heißen **Constraints**, die lineare Funktion $g(x) = c^T x$ **Zielfunktion**. Unsere Aufgabe ist es also, einen ganzzahligen Vektor x zu finden, der alle Constraints erfüllt und die Zielfunktion minimiert.

Ein Vektor $\hat{x}$, der alle Constraints erfüllt, aber nicht notwendigerweise optimal ist, nennen wir eine **gültige Lösung**.

Wir beschränken uns hier auf den Spezialfall, daß $x \in \{0,1\}^n$ sein muß. Diese Variante wird als **0/1-ILP** bezeichnet. Sofern beim allgemeinen ILP-Problem alle

[1] ILP: integer linear program

Variablen beschränkt sind, stellt die Beschränkung auf $x \in \{0,1\}^n$ keine wesentliche Einschränkung dar. Wenn für eine Variable x_i gilt:

$$u \leq x_i \leq o,$$

dann läßt sich dies umformen zu

$$0 \leq x_i - u \leq o - u.$$

Sei $2^{k-1} \leq o - u < 2^k$. Dann ersetze überall x_i durch

$$\sum_{j=0}^{k-1} x_{ij} 2^j + u$$

und füge den Constraint

$$\sum_{j=0}^{k-1} x_{ij} 2^j \leq o - u$$

hinzu. Dadurch erhält man ein 0/1-ILP-Problem. Hat man eine Lösung $x_{i0}, \ldots, x_{i,k-1}$ berechnet, so erhält man den Wert für x_i des ursprünglichen Problems als

$$x_i = \sum_{j=0}^{k-1} x_{ij} 2^j + u$$

Beispiel 2.1 *Sei ein 0/1-ILP-Problem gegeben durch die Constraints*

$$\begin{aligned} x_1 - x_3 - 2x_4 &\leq 0 \\ -2x_2 + 3x_3 - 2x_4 &\leq -1 \\ x_1 + x_2 + x_3 + x_4 &\leq 2 \end{aligned}$$

und die Zielfunktion

$$g(x_1, \ldots, x_4) = 2x_1 - x_2 + 3x_3 - x_4$$

Dieses Problem besitzt die (optimale) Lösung

x_1	x_2	x_3	x_4
0	1	0	1

mit $g(0,1,0,1) = -2$.

Obwohl wir uns hauptsächlich mit 0/1-ILP-Problemen beschäftigen, werden wir in diesem Kapitel, wenn möglich, die Sätze und Algorithmen nicht für 0/1-ILP angeben, sondern für allgemeine ILP-Probleme. Notwendige Einschränkungen auf 0/1-ILP sind explizit gekennzeichnet.

2.2. Komplexität von 0/1-ILP

Daß es sich bei 0/1-ILP komplexitätstheoretisch um ein schwieriges Problem handelt, folgt aus folgendem Satz:

Satz 2.1 *0/1-ILP (und damit auch ILP) ist NP-hart.*

Beweis:
SAT läßt sich einfach auf 0/1-ILP reduzieren. Wir führen die Transformation an einem Beispiel durch. Den allgemeinen Beweis kann man in [13] nachlesen.
Betrachte also folgendes SAT-Problem:

$$(x_1 \vee \bar{x}_3 \vee x_4) \wedge (\bar{x}_2 \vee x_3 \vee \bar{x}_4) \wedge (x_1 \vee x_4)$$

Ein äquivalentes 0/1-ILP-Problem sieht folgendermaßen aus (da wir nichts minimieren müssen, wählen wir als Zielfunktion eine konstante Funktion):

$$\begin{aligned} x_1 + (1 - x_3) + x_4 &\geq 1 \\ (1 - x_2) + x_3 + (1 - x_4) &\geq 1 \\ x_1 + x_4 &\geq 1 \end{aligned}$$

Jede Klausel wird in einen Constraint umgewandelt: Aus dem logischen Oder wird ein Plus, eine negierte Variable $\overline{x}_i$ wird durch $(1 - x_i)$ ersetzt. Diese Summe muß größer oder gleich Eins sein, damit die Klausel erfüllt ist.

Bringt man dies in Normalform wie in Definition 2.1, erhält man

$$\begin{aligned} -x_1 + x_3 - x_4 &\leq 0 \\ x_2 - x_3 + x_4 &\leq 1 \\ -x_1 - x_4 &\leq -1 \end{aligned}$$

Die Funktionsweise und die Korrektheit der Transformation dürften damit klar sein. □

Ein wichtiger Begriff, der im Folgenden oft auftauchen wird, ist der der LP-Relaxierung:

Definition 2.2 *Sei*

$$\min\{c^T x \mid Ax \leq b, x \in \mathbb{Z}^n\}$$

ein ILP-Problem. Die zugehörige ***LP-Relaxierung*** *ist dann*

$$\min\{c^T x \mid Ax \leq b, x \in \mathbb{R}^n\}.$$

Ein Optimierungsproblem dieser Form heißt ***Lineares Programm*** *(LP[2]).*

[2]LP: linear program

Ein Vektor $\hat{x} \in \mathbb{R}^n$, der alle Constraints des LP-Programms erfüllt, nennen wir eine **gültige LP-Lösung**.

Verzichtet man auf die Bedingung, daß die Variablen nur die Werte 0 und 1 annehmen dürfen, und läßt reelle Werte aus $[0, 1]$ zu, so existieren polynomielle Lösungsverfahren (z. B. die Ellipsoid-Methode, siehe [9]).

Leider läßt sich aus der kontinuierlichen LP-Lösung im Allgemeinen nicht effizient auf die ILP-Lösung schließen, außer es ist $P = NP$. Könnten wir in polynomieller Zeit von der LP-Lösung auf die ILP-Lösung schließen, dann könnten wir – da sich LP-Programme in polynomieller Zeit lösen lassen – ILP polynomiell lösen und wir hätten $P = NP$ gezeigt.

Trotzdem wird uns die LP-Lösung als untere Schranke in Kapitel 5 von Nutzen sein.

2.3. Geometrische Interpretation

In diesem Abschnitt werden wir einige Begriffe der linearen Algebra einführen, die im Zusammenhang mit der ganzzahligen linearen Optimierung von Bedeutung sind. Mit deren Hilfe können wir das Verhältnis von LP und ILP veranschaulichen.

Definition 2.3 *Eine Menge der Form*

$$\{x \in \mathbb{R}^n \mid a^T x \leq b, a \in \mathbb{R}^n, b \in \mathbb{R}\}$$

heißt ***Halbraum****.*

Man sieht leicht, daß die einzelnen Constraints eines LP-Problems jeweils einen Halbraum definieren. Die Menge der gültigen Lösungen eines LP-Problems wird also durch den Schnitt von m Halbräumen definiert.

Definition 2.4 *Seien $H_1, \ldots, H_m$ Halbräume. Setze*

$$H := \bigcap_{i=1}^{m} H_i.$$

H heißt ***Polyeder****. Falls H beschränkt ist, wird H auch als* ***Polytop*** *bezeichnet.*

Da alle unsere Variablen nach unten durch 0 und nach oben durch 1 beschränkt sind, sind die Lösungsräume aller LP-Probleme, die wir betrachten, Polytope. Das Polytop, das alle gültigen Lösungen eines LP-Problems enthält, nennen wir **LP-Polytop**.

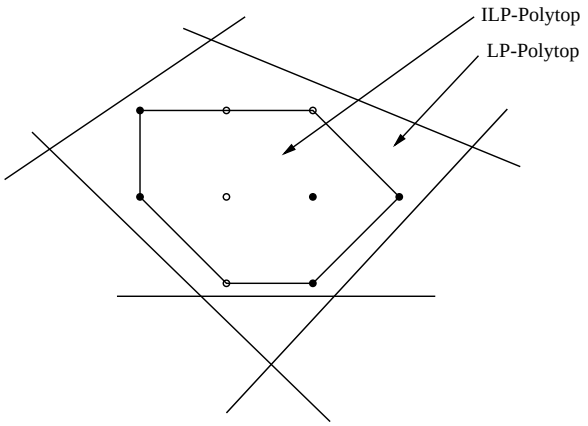


Abbildung 2.1.: Graphische Darstellung eines ILP-Problems

Definition 2.5 *Sei $P = \{p_1, \ldots, p_k\} \subset \mathbb{R}^n$ eine endliche Menge von Punkten. Die **konvexe Hülle** von P,* $\mathrm{conv}(P)$*, ist gegeben durch*

$$\mathrm{conv}(P) = \{\lambda_1 p_1 + \cdots + \lambda_k p_k \,|\, \lambda_i \geq 0, \lambda_1 + \cdots + \lambda_k = 1\}$$

Lemma 2.1 *Eine Menge P ist genau dann ein Polytop, wenn sie die konvexe Hülle einer endlichen Menge von Punkten ist.*

Beweis:
Siehe [9, 20] □

Weil Konvexkombinationen gültiger Lösungen alle Constraints erfüllen (aber natürlich nicht unbedingt ganzzahlig sind), wird für uns die konvexe Hülle der ganzzahligen Punkte im LP-Polytop wichtig sein. Nach Lemma 2.1 ist sie wieder ein Polytop, das vollständig im LP-Polytop enthalten ist. Wir bezeichnen es im Folgenden als **ILP-Polytop**. Graphisch stellt sich die Situation wie in Abbildung 2.1 dar.

Eine besondere Rolle spielt für uns der Rand des Polytops, der aus sogenannten Faces besteht. Wenn wir uns ein dreidimensionales Polytop vorstellen, dann besteht sein Rand aus „Flächen“, „Kanten“ und „Ecken“. Diese bilden die Faces des Polytops. Sie besitzen verschiedene Dimensionen: Die „Flächen“ haben die Dimension 2, die „Kanten“ die Dimesion 1 und die „Ecken“ sind 0-dimensional.

Wenn wir Polytope der Dimension n betrachten, besitzen diese im Allgemeinen Faces in allen Dimensionen i mit $0 \leq i < n$. Die beiden Extremfälle – Faces der Dimension $n-1$ und der Dimension 0 – spielen eine besondere Rolle. Deshalb nennt man sie Facetten bzw. Ecken.

Wir wollen die Begriffe formal definieren:

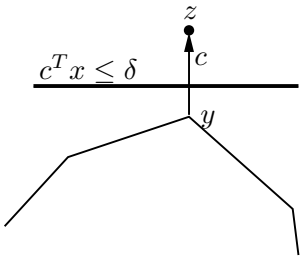


Abbildung 2.2.: zum Beweis von Satz 2.2

Definition 2.6 *Sei P ein Polytop. Eine Menge F heißt **Face** von P, wenn $F = P$ ist oder wenn es einen Vektor $a \neq 0$ gibt und $\delta = \max\{a^T y \mid y \in P\}$, so daß*

$$F = \{z \in P \mid a^T z = \delta\}.$$

*Wenn $\{x\}$ ein Face von P ist, dann ist x eine **Ecke** von P. Ein maximales Face, das verschieden von P ist, heißt **Facette**.*

Wenn P ein Polytop ist, dann ist die Menge der Punkte, in denen das Minimum $\min\{c^T x \mid x \in P\}$ angenommen wird, ein Face von P. Dies folgt unmittelbar aus der Definition eines Face'.

Dadurch, daß jedes nicht-leere Face eines Polytops eine Ecke enthält, bedeutet das auch, daß es immer eine Ecke gibt, in der das Minimum angenommen wird. Der Simplex-Algorithmus, der in der Praxis zur Lösung von LP-Problemen eingesetzt wird ([9, Kap. 3.2]), liefert – wenn das LP lösbar ist – immer eine solche Ecke des LP-Polytops.

Grundlegend für unser Verfahren, das wir in Kapitel 6 beschreiben werden, ist die Existenz von sogenannten Schnittebenen (siehe [21]):

Satz 2.2 *Sei P ein Polytop und $z \notin P$. Dann gibt es einen von Null verschiedenen Vektor $c \in \mathbb{R}^n$ und ein $\delta \in \mathbb{R}$ mit*

$$c^T z > \delta \quad \text{und} \quad \forall x \in P : c^T x \leq \delta.$$

*Der Constraint $c^T x \leq \delta$ wird als **Schnittebene** bezeichnet.*

Beweis:

Sei $P \neq \emptyset$ (ansonsten ist die Aussage trivial). Dann gibt es einen Vektor $y \in P$, der $\|z - y\|$ minimiert. Setze dann:

$$c := z - y \quad \text{und} \quad \delta := \tfrac{1}{2}(\|z\|^2 - \|y\|^2).$$

Graphisch stellt sich die Situation dann wie in Abbildung 2.2 dar. Wir müssen nachrechnen, daß die Ungleichung $c^T x \leq \delta$ die geforderten Eigenschaften besitzt:

Es ist

$$c^T z = (z-y)^T z > (z-y)^T z - \tfrac{1}{2}\|z-y\|^2 = \delta$$

Damit ist der erste Teil der Aussage bewiesen.
Angenommen, der zweite Teil wäre falsch, d. h. es gäbe ein $x \in P$ mit $c^T x > \delta$.
Wegen $c^T y < c^T y + \frac{1}{2}\|c\|^2 = \delta$ wissen wir, daß $c^T(x-y) = c^T x - c^T y > \delta - c^T y = \frac{1}{2}\|c\|^2 > 0$ ist. Folglich existiert ein $\lambda \in (0,1]$ mit

$$\lambda < \frac{2c^T(x-y)}{\|x-y\|^2}$$

Setze dann $w := \lambda x + (1-\lambda)y$. Da P konvex ist, ist $w \in P$. Außerdem gilt:

$$\begin{aligned}\|w-z\|^2 &= \|\lambda(x-y)+(y-z)\|^2 = \|\lambda(x-y)+c\|^2 \\ &= \lambda^2\|x-y\|^2 - 2\lambda c^T(x-y) + \|c\|^2 < \|c\|^2 = \|y-z\|^2\end{aligned}$$

Das $<$ folgt dabei aus der Bedingung an λ.
$\|w-z\|^2 < \|y-z\|^2$ stellt einen Widerspruch zur Definition von y dar. □

Das Problem, zu einem Polytop P und einem Punkt x zu entscheiden, ob $x \in P$ gilt, und, wenn $x \notin P$ ist, eine Schnittebene zu finden, die gültig für ganz P ist, aber nicht für x, heißt **Separierungsproblem**.

2.4. Dualität

Ein wichtiges Prinzip bei der linearen Programmierung ist das der Dualität.

Definition 2.7 *Sei ein LP-Programm gegeben durch*

$$\begin{aligned}&\min c^T x \\ &Ax \le b \\ &x \in \mathbb{R}^n.\end{aligned}$$

Es wird auch als ***primales Programm*** *bezeichnet. Das zugehörige* ***duale Programm*** *ist dann gegeben durch*

$$\begin{aligned}&\max y^T b \\ &y^T A = c^T \\ &y \le 0\end{aligned}$$

Wenn wir andere Typen von Constraints (also der Form $a^T x = b$ und $a^T x \ge b$) zulassen und außerdem gestatten, daß die Variablen als Wertebereiche nur die nicht-negativen bzw. nicht-positiven Werte annehmen dürfen, so kann man zeigen, daß das

zugehörige duale Programm folgende Gestalt hat:

primal	**dual**
$\min c^T x$	$\max y^T b$
$Ax \begin{pmatrix} \leq \\ = \\ \geq \end{pmatrix} b$	$y \begin{pmatrix} \leq 0 \\ \in \mathbb{R}^m \\ \geq 0 \end{pmatrix}$
$x \begin{pmatrix} \geq 0 \\ \in \mathbb{R}^n \\ \leq 0 \end{pmatrix}$	$y^T A \begin{pmatrix} \leq \\ = \\ \geq \end{pmatrix} c^T$

(2.1)

Jeder Variablen im primalen Programm entspricht ein Constraint im dualen Programm und umgekehrt. Wenn der i-te Constraint im primalen Programm vom Typ $a_i^T x = b_i$ ist, dann besitzt y_i im dualen Programm den Wertebereich $y_i \in \mathbb{R}$. Wenn im primalen Programm $x_j \geq 0$ ist, dann ist der j-te Constraint im dualen Programm vom Typ $(A^T)_j \cdot y \leq c_j$.

Da wir später mit Constraints umgehen müssen, die die Form von Gleichungen haben, wollen wir das duale Programm für Gleichungen mit unbeschränkten Variablen herleiten:

Lemma 2.2 *Das duale Programm zu* $\min\{c^T x \,|\, Ax = b,\ x \in \mathbb{R}^n\}$ *ist*

$$\max\{y^T b \,|\, y^T A = c,\ y \in \mathbb{R}^m\}.$$

Beweis:

Sei ein LP-Problem wie oben gegeben Dann können wir das auch schreiben als

$$\min\Big\{c^T x \,|\, \begin{pmatrix} A \\ -A \end{pmatrix} \cdot x \leq \begin{pmatrix} b \\ -b \end{pmatrix},\ x \in \mathbb{R}^n\Big\}$$

Das zugehörige duale Programm wie in Definition 2.7 lautet:

$$\max\Big\{y^T \begin{pmatrix} b \\ -b \end{pmatrix} \,\Big|\, y^T \begin{pmatrix} A \\ -A \end{pmatrix} = c^T,\ y \leq 0\Big\}$$

Wir spalten y in y_1 und y_2 auf. Dann läßt sich das duale Programm schreiben als

$$\max\{(y_1 - y_2)^T b \,|\, (y_1 - y_2)^T A = c^T,\ y_1, y_2 \leq 0\}$$

Die Differenz $y_1 - y_2$ können wir durch neue Variablen z ersetzen, die nicht mehr auf Werte kleiner oder gleich Null beschränkt sind, sondern auch positive Werte annehmen dürfen, also:

$$\max\{z^T b \,|\, z^T A = c^T,\ z \in \mathbb{R}^m\}.$$

□

Bildet man zum dualen Programm dessen duales Programm, erhält man ein LP-Programm, das äquivalent zum ursprünglichen Programm ist [17]

Der folgende grundlegende Satz gibt das Verhältnis zwischen primalem und dualem Programm wieder:

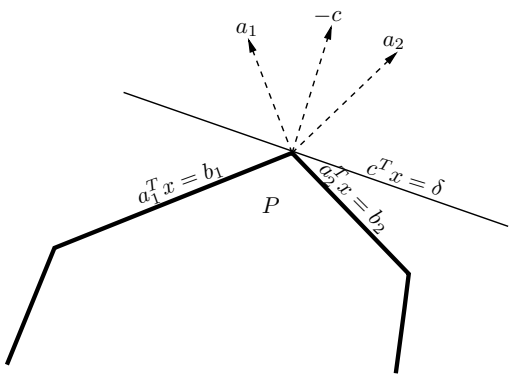


Abbildung 2.3.: Geometrische Veranschaulichung des Dualitätssatzes

Satz 2.3 (Dualitätssatz) *Wenn sowohl das primale als auch das duale Programm lösbar und beschränkt sind, dann besitzen ihre optimalen Lösungen denselben Wert, d. h.*

$$\min\{c^T x \,|\, Ax \leq b\} = \max\{y^T b \,|\, y \leq 0,\ y^T A = c^T\}.$$

Beweis:
Siehe [20, Cor. 7.1g]

Wir wollen hier an dieser Stelle wenigstens den Dualitätssatz plausibel machen, indem wir eine geometrische Veranschaulichung angeben. Betrachte dazu Abbildung 2.3.

Sei $\delta := \min\{c^T x \,|\, Ax \leq b\}$. Dieses Minimum werde im Punkt x^* angenommen. O. B. d. A. können wir annehmen, daß die ersten k Constraints von A mit Gleichheit erfüllt werden, d. h. es gilt für $i = 1, \ldots, k$: $a_i^T x^* = b_i$. Dann ist $c^T x = \delta$ eine nichtpositive Linearkombination der Gleichungen $a_1^T x = b_1, \ldots, a_k^T x = b_k$. Es existieren also $\lambda_1, \ldots, \lambda_k \leq 0$ mit

$$\lambda_1 a_1 + \lambda_2 a_2 + \cdots + \lambda_k a_k = c$$
$$\lambda_1 b_1 + \lambda_2 b_2 + \cdots + \lambda_k b_k = \delta$$

Setze $y^* = (\lambda_1, \ldots, \lambda_k, 0, \ldots, 0)^T$. Dann ist y^* eine gültige Lösung für das duale Problem $\max\{y^T b \,|\, y \leq 0, y^T A = c^T\}$. Deshalb gilt:

$$\min\{c^T x \,|\, Ax \leq b\} = \delta = \lambda_1 b_1 + \cdots + \lambda_k b_k \leq \max\{y^T b \,|\, y \leq 0, y^T A = c^T\}.$$

Da wegen $y \leq 0$ außerdem gilt:

$$c^T x = y^T A x \geq y^T b$$

folgt

$$\min\{c^T x \,|\, Ax \leq b\} \geq \max\{y^T b \,|\, y \leq 0, y^T A = c^T\}.$$

und daraus die Gleichheit. □

2.5. Totale Unimodularität

In diesem Abschnitt werden wir untersuchen, wann LP- und ILP-Polytop übereinstimmen. Dieser Fall liegt genau dann vor, wenn alle Ecken des LP-Polytops ganzzahlig sind.

Definition 2.8 *Ein Polytop P heißt **ganzzahlig**, wenn alle Ecken von P aus $\mathbb{Z}^n$ sind.*

Wenn wir wissen, daß das Polytop ganzzahlig ist, dann erhalten wir durch Lösung der zugehörigen LP-Relaxierung die optimale Lösung des ILP-Problems. Ein wichtiges Kriterium dafür ist die totale Unimodularität der Matrix A. Dies wird an einigen Stellen unseres Verfahrens von Bedeutung sein.

Die Ausführungen in diesem Abschnitt orientieren sich größtenteils an [9, Kap. 4.5], [21] und [17, Kap. III.1].

Definition 2.9 *Eine **Subdeterminante** einer Matrix $A \in \mathbb{R}^{m\times n}$ ist die Determinante einer quadratischen Teilmatrix von A, die sich aus beliebigen Zeilen und Spalten von A zusammensetzt.*

Definition 2.10 *Eine Matrix A heißt **total unimodular**, wenn jede Subdeterminante von A den Wert 0, 1 oder -1 hat.*

Daraus folgt unmittelbar, daß A als Einträge nur 0, 1 und -1 haben kann. Der folgende Satz zeigt die Bedeutung der total unimodularen Matrizen für die ganzzahlige lineare Optimierung auf:

Satz 2.4 (Hoffmann und Kruskal, 1956) *Eine ganzzahlige Matrix A ist genau dann total unimodular, wenn für jeden ganzzahligen Vektor b das Polyeder*

$$\{x \in \mathbb{R}^n \mid Ax \leq b, x \geq 0\}$$

ganzzahlig ist.

Beweis:

Sei $A \in \mathbb{R}^{m\times n}$ und $P = \{x \in \mathbb{R}^n \mid Ax \leq b, x \geq 0\}$. Die minimalen Faces von P sind gerade die Ecken von P.

Um die eine Richtung zu zeigen, nehmen wir an, daß A total unimodular ist. Sei b ein beliebiger ganzzahliger Vektor und x eine Ecke von P. x ist dann die Lösung von $A'x = b'$ für ein Teilsystem $A'x \leq b'$ von $\begin{pmatrix} A \\ -I \end{pmatrix} x \leq \begin{pmatrix} b \\ 0 \end{pmatrix}$, wobei A' eine reguläre $n \times n$-Matrix ist. Da A total unimodular ist, gilt $|\det A'| = 1$. Nach der Cramer'schen Regel erhalten wir, daß $x = (A')^{-1}b$ ganzzahlig ist.

Für die andere Richtung nehmen wir an, daß für jeden ganzzahligen Vektor b die Ecken

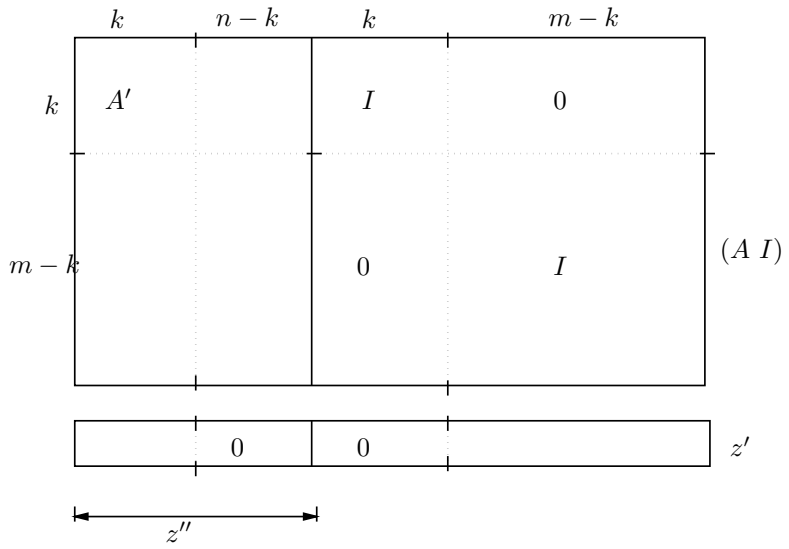


Abbildung 2.4.: Zum Beweis von Theorem 2.4

von P ganzzahlig sind. Sei A' eine reguläre $k \times k$-Teilmatrix von A. Wir müssen zeigen, daß $|\det A'| = 1$ ist. O.B.d.A. enthalte A' Elemente der ersten k Zeilen und Spalten von A (dies kann man immer durch Vertauschen von Variablen und Constraints erreichen, ohne daß sich der Betrag der Determinante ändert). Betrachte die ganzzahlige $m \times m$-Matrix B, die aus den ersten k und den letzten $m - k$ Spalten von $(A\,I)$ besteht (siehe Abbildung 2.4). Offensichtlich gilt $|\det A'| = |\det B|$. Um zu zeigen, daß $|\det B| = 1$ ist, zeigen wir, daß $\det B^{-1}$ ganzzahlig ist. Wegen $\det B \cdot \det B^{-1} = 1$ folgt daraus dann, daß $|\det B| = 1$ sein muß und wir sind fertig.
Sei e_i für $i = 1, \ldots, m$ der i-te Einheitsvektor in $\mathbb{R}^m$; wir zeigen, daß $B^{-1}e_i$ ganzzahlig ist. Dazu wähle einen ganzzahligen Vektor y mit $z := y + B^{-1}e_i \geq 0$. Dann ist $b := Bz = By + e_i$ ganzzahlig. Wir erweitern $z \in \mathbb{R}^m$ um Null-Einträge, um $z' \in \mathbb{R}^{n+m}$ zu erhalten, für das gilt:

$$(A\,I)z' = Bz = b$$

Somit gehört $z'' \in \mathbb{R}^n$, das aus den ersten n Einträgen von z' besteht, zu P. Außerdem sind n linear unabhängige Constraints mit Gleichheit erfüllt, nämlich die ersten k und die letzten $n - k$ Ungleichungen von

$$\begin{pmatrix} A \\ -I \end{pmatrix} z'' \leq 0.$$

Folglich ist z'' Ecke von P und nach Voraussetzung ganzzahlig. Aber dann muß z' ebenfalls ganzzahlig sein: seine ersten n Komponenten sind dieselben wie bei z'' und die letzten m Komponenten sind die Slack-Variablen $b - Az''$ (A und b sind ganzzahlig). Damit ist z ganzzahlig und folglich auch $B^{-1}e_i = z - y$. □

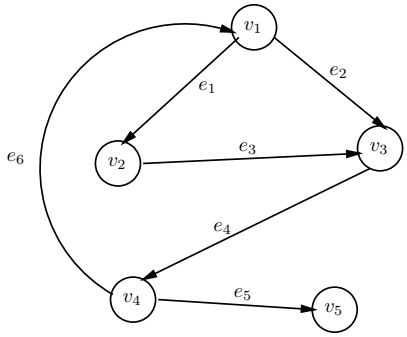


Abbildung 2.5.: Graph aus Beispiel 2.2

Wir sind später an einer speziellen Klasse von total unimodularen Matrizen interessiert, den Inzidenzmatrizen von gerichteten Graphen:

Definition 2.11 *Die* ***Inzidenzmatrix*** $M \in \{-1, 0, 1\}^{|V| \times |E|}$ *eines gerichteten Graphen* $\mathcal{G} = (V, E)$ *ist definiert durch*

$$(M)_{v,e} = \begin{cases} +1 & \textit{falls } v \textit{ Anfangspunkt von } e \\ -1 & \textit{falls } v \textit{ Endpunkt von } e \\ 0 & \textit{sonst.} \end{cases}$$

Bevor wir zeigen, daß diese Matrizen total unimodular sind, wollen wir die Definition anhand eines kleinen Beispiels verdeutlichen:

Beispiel 2.2 *Betrachte den Graphen aus Abbildung 2.5. Die zugehörige Inzidenzmatrix hat folgende Gestalt (die Knoten und Kanten sind ihren Indizes entsprechend geordnet):*

$$M = \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & -1 \\ -1 & 0 & 1 & 0 & 0 & 0 \\ 0 & -1 & -1 & 1 & 0 & 0 \\ 0 & 0 & 0 & -1 & 1 & 1 \\ 0 & 0 & 0 & 0 & -1 & 0 \end{pmatrix}$$

Satz 2.5 *Für alle gerichteten Graphen ist die Inzidenzmatrix* M *total unimodular.*

Beweis:

Wir zeigen durch Induktion über k, daß die Determinante jeder $k \times k$-Teilmatrix B von M den Wert 0, 1 oder -1 hat.

Da alle Einträge von M aus $\{-1, 0, 1\}$ sind, ist die Behauptung für $k = 1$ klar.

Sei also $k > 1$. Sei B eine $k \times k$-Teilmatrix von M. Wir unterscheiden drei Fälle:

1. B enthält eine Null-Spalte. Dann ist $\det B = 0$.
2. B enthält eine Spalte, in der genau ein Eintrag von 0 verschieden ist. Da das Vertauschen von Zeilen und Spalten höchstens das Vorzeichen der Determinante ändert, können wir annehmen, daß B folgende Gestalt hat:

$$B = \begin{pmatrix} \pm 1 & b^T \\ \mathbf{0} & B' \end{pmatrix}$$

   Gemäß der Induktionsvoraussetzung gilt $\det B' \in \{-1, 0, 1\}$ und somit auch $\det B \in \{-1, 0, 1\}$.
3. B enthält nur Spalten mit zwei von Null verschiedenen Einträgen. Dann muß jede Spalte genau eine 1 und eine -1 enthalten. Damit addieren sich die Zeilen von B aber zum Nullvektor auf und sind folglich linear abhängig. Deshalb ist $\det B = 0$.

□

2.6. Lagrange-Relaxierung

Angenommen, unser Optimierungsproblem würde sich bedeutend vereinfachen, wenn wir einige Constraints weglassen dürften. Die Idee bei der Lagrange-Relaxierung ist, die „schwierigen“ Constraints mit einem Vorfaktor zur Zielfunktion hinzuzunehmen, so daß sich der optimale Wert der Zielfunktion verschlechtert, wenn diese Constraints nicht erfüllt sind. Dadurch können wir eine gute untere Schranke für die tatsächliche Lösung berechnen.

In diesem Abschnitt gehen wir immer davon aus, daß die Constraints in zwei Gruppen aufgeteilt sind, d. h. das zu lösende Problem sieht folgendermaßen aus:

Definition 2.12 (Problemstellung) *Sei $c \in \mathbb{R}^n$, $A \in \mathbb{R}^{n \times m_1}$, $A' \in \mathbb{R}^{n \times m_2}$, $b \in \mathbb{R}_1^m$ und $b' \in \mathbb{R}^{m_2}$. Wir suchen ein $\bar{x} \in \{0,1\}^n$, in dem folgendes Minimum angenommen wird:*

$$\min_x \{c^T x \mid x \in \{0,1\}^n,\ Ax \leq b,\ A'x \leq b'\}. \tag{2.2}$$

Wir setzen

$$Q := \{x \in \{0,1\}^n \mid Ax \leq b\} \tag{2.3}$$

und nehmen an, daß wir effizient beliebige lineare Funktionen über Q minimieren können. Die Constraints, die wir in Q weggelassen haben, nehmen wir zur Zielfunktion hinzu und definieren

$$LR(\lambda) := \min\{c^T x + \lambda^T (A'x - b') \mid x \in Q\} \tag{2.4}$$

Die Faktoren $\lambda \in \mathbb{R}_{\geq 0}^{m_2}$ heißen **Lagrange-Multiplikatoren**.

Lemma 2.3 *Für alle $\lambda \geq 0$ ist $LR(\lambda) \leq c^T \bar{x}$.*

Beweis:
Es genügt zu zeigen: Es gibt ein $x^* \in Q$ mit

$$c^T x^* + \lambda^T (A' x^* - b') \leq c^T \bar{x}.$$

Dazu betrachten wir die optimale Lösung $\bar{x}$ aus Definition 2.12. Dafür gilt: $\bar{x} \in Q$ und $A'\bar{x} \leq b'$, also $A'\bar{x} - b' \leq 0$. Wegen $\lambda \geq 0$ gilt auch $\lambda^T(A'\bar{x} - b') \leq 0$. Folglich ist

$$c^T \bar{x} + \lambda^T (A' \bar{x} - b') \leq c^T \bar{x}.$$

□

Lemma 2.4 *LR ist konkav, d. h.*

$$LR(a\lambda + (1-a)\kappa) \geq a \cdot LR(\lambda) + (1-a) LR(\kappa)$$

für alle $a \in [0,1]$.

Beweis:
Sei $a \in [0,1]$. Dann gilt:

$$\begin{aligned}
LR(a\lambda + (1-a)\kappa) &= \min\{c^T x + (a\lambda^T + (1-a)\kappa^T)(A'x - b') \,|\, x \in Q\} \\
&= \min\{ac^T x + (1-a)c^T x + a\lambda^T(A'x - b') \\
&\qquad + (1-a)\kappa^T(A'x - b') \,|\, x \in Q\} \\
&= \min\{a(c^T x + \lambda^T(A'x - b')) \\
&\qquad + (1-a)(c^T x + \kappa^T(A'x - b') \,|\, x \in Q\} \\
&\geq a \cdot \min\{c^T x + \lambda^T(A'x - b') \,|\, x \in Q\} \\
&\qquad + (1-a) \cdot \min\{c^T x + \kappa(A'x - b') \,|\, x \in Q\} \\
&= a \cdot LR(\lambda) + (1-a) \cdot LR(\kappa)
\end{aligned}$$

□

Da wir in möglichst guten unteren Schranken interessiert sind, lösen wir folgendes Problem:

$$c_{LR} := \max\{LR(\lambda) \,|\, \lambda \geq 0\}$$

Dabei ist zu beachten, daß die Funktion $LR(\lambda)$ konkav und nach oben beschränkt ist (weil $LR(\lambda)$ eine untere Schranke für das ursprüngliche Problem ist, ist $c^T\bar{x}$ eine obere Schranke für $LR(\lambda)$).
Die Frage, die sich jetzt stellt, ist, wie gut diese untere Schranke ist. Der folgende Satz hilft, die Güte der Schranke abzuschätzen:

Satz 2.6 *Sei* $Q = \{x \in \mathbb{Z}^n \mid Ax \leq b\} \neq \emptyset$, $A' \in \mathbb{R}^{n \times m_2}$ *und* $b' \in \mathbb{R}^{m_2}$. *Nehmen wir an, daß die Menge* $\{x \in Q \mid A'x \leq b'\}$ *nicht leer ist. Dann ist*

$$\max\{LR(\lambda) \mid \lambda \geq 0\} = \min\{c^T x \mid A'x \leq b', x \in \text{conv}(Q)\}.$$

Beweis:
Siehe [17, Kap. II.3]. □

Das bedeutet insbesondere: Wenn wir ein ganzzahliges lineares Programm

$$\min\{c^T x \mid x \in \{0,1\}^n, Ax \leq b, A'x \leq b'\}$$

haben, für das das Polytop $\{x \in \mathbb{R}^n \mid Ax \leq b\}$ ganzzahlig ist, dann erhalten wir durch die Lagrange-Relaxierung denselben Wert für die obere Schranke wie durch eine einfache LP-Relaxierung. Wenn $\{x \in \mathbb{R}^n \mid Ax \leq b\}$ nicht ganzzahlig ist, dann ist der Wert der Lagrange-Relaxierung im Allgemeinen besser.

Wir wollen hier noch ein Verfahren zur Berechnung von c_{LR} angeben, das auf dem sogenannten Subgradienten-Verfahren beruht [9, Kap. 5.6]:

Algorithmus 2.1

- *Starte mit einem beliebigen* $\lambda^{(1)} \geq 0$.
- *Im* i*-ten Durchgang, wenn* $\lambda^{(i)}$ *bereits berechnet ist, finde einen Vektor* $x^{(i)}$, *der* $c^T x + \lambda^{(i)\,T}(A'x - b')$ *minimiert (d. h. berechne* $LR(\lambda^{(i)})$*). Setze dann*

  $$\lambda^{(i+1)} := \max\{0, \lambda^{(i)} - t_i(A'x^{(i)} - b')\}$$

  für ein geeignetes $t_i \geq 0$.

Polyak [18] konnte 1967 zeigen, daß das Verfahren gegen das λ konvergiert, das LR maximiert, falls $\lim\limits_{i \to \infty} t_i = 0$ und $\sum\limits_{i=1}^{\infty} t_i = \infty$ ist. Es bietet sich deshalb an $t_i = \frac{1}{i}$ für $i \geq 1$ zu wählen.

Damit haben wir die theoretischen Grundlagen von 0/1-ILP zusammen, die wir für unseren Algorithmus benötigen. Somit können wir uns im nächsten Kapitel den binären Entscheidungsdiagrammen (BDDs) widmen, die die zweite Grundlage für unser Verfahren darstellen.

3. Binäre Entscheidungsdiagramme

Binäre Entscheidungsdiagramme sind, wenn man einige Bedingungen an ihre Struktur stellt, eine kanonische Darstellung Boolescher Funktionen. Da sie für viele in der Praxis wichtige Funktionen mit relativ wenig Speicherplatz auskommen und die Synthese-Operationen effizient unterstützen, werden BDDs heutzutage in den meisten Tools zur Verfikation von digitalen Schaltungen eingesetzt. Wir werden sie zur kompakten Darstellung des Lösungsraums von ILP-Problemen einsetzen.

Vorher müssen wir uns die BDDs genauer anschauen (vgl. [2]):

Definition 3.1 (Syntax von BDDs) *Sei $X = \{x_1, \ldots, x_n\}$ eine Menge Boolescher Variablen. Ein binärer Entscheidungsgraph (BDD[1]) über X ist ein gerichteter azyklischer Graph $G = (V, E)$ mit folgenden Eigenschaften:*

- *Jeder Knoten ist entweder ein Blatt oder ein innerer Knoten.*
- *Jedes Blatt ist entweder mit* 0 *oder mit* 1 *beschriftet und hat keine ausgehenden Kanten.*
- *Es gibt genau einen Knoten mit Eingangsgrad* 0*. Er wird als* ***Wurzel*** *bezeichnet.*
- *Jeder innere Knoten v ist mit einer Variablen $x_i = \text{label}(v) \in X$ beschriftet und besitzt genau zwei ausgehende Kanten, deren Endknoten als* $\text{low}(v)$ *bzw.* $\text{high}(v)$ *bezeichnet werden.*

Wir definieren außerdem eine Funktion $\text{par} : E \to \{0, 1\}$, die zu einer Kante im BDD angibt, ob sie zum low-Sohn oder zum high-Sohn eines Knotens führt:

$$\text{par}(e) = \begin{cases} 1 & \text{falls } e = (u, v) \text{ und } v = \text{high}(u) \\ 0 & \text{falls } e = (u, v) \text{ und } v = \text{low}(u) \end{cases}$$

$\text{head}(e)$ bezeichne den Anfangsknoten der Kante e, $\text{tail}(e)$ den Endknoten.

Definition 3.2 (Semantik von BDDs) *Sei G ein BDD über der Variablenmenge X. Wir ordnen jedem Knoten v von G eine Boolesche Funktion f_v zu:*

[1] BDD: Binary decision diagram

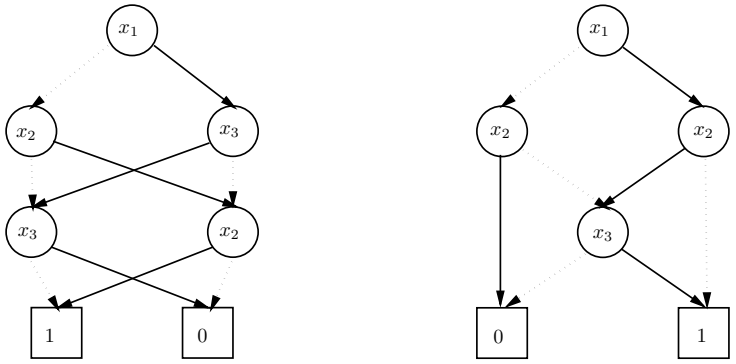


Abbildung 3.1.: Die BDDs für Beispiel 3.1

- *Ist v ein Blatt mit der Beschriftung 0, so ist $f_v(X) = 0$.*
- *Ist v ein Blatt mit der Beschriftung 1, so ist $f_v(X) = 1$.*
- *Ist v ein innerer Knoten mit Beschriftung x_i, so gilt*

$$f_v(X) = x_i f_{\text{high}(v)}(X) + \overline{x_i} f_{\text{low}(v)}(X).$$

Die durch ein BDD dargestellte Funktion ist diejenige Funkion f, die der Wurzel des BDDs zugeordnet wird.

Um eine kanonische Darstellung Boolescher Funktionen durch BDDs zu bekommen, brauchen wir weitere Einschränkungen an die Struktur:

Definition 3.3 *Ein BDD G heißt **frei**, wenn auf jedem Pfad von der Wurzel zu einem Blatt jede Variable $x_i \in X$ höchstens einmal als Beschriftung eines Knotens vorkommt.*
*Ein BDD G heißt **geordnet**, wenn es frei ist und die Variablen auf allen Pfaden von der Wurzel zu einem Blatt in derselben Reihenfolge auftreten.*
*Ein BDD G heißt **reduziert**, wenn für alle Knoten u, v gilt:*

$$u \neq v \quad \Rightarrow \quad f_u \neq f_v.$$

*Geordnete und reduzierte BDDs werden als **OBDDs**[2] bezeichnet.*

Beispiel 3.1 *In Abbildung 3.1 sind zwei BDDs für die Funktion*

$$f(x_1, x_2, x_3) = x_1\overline{x}_2 + x_1x_2 + \overline{x}_1\overline{x}_2x_3$$

[2]OBDD: Ordered binary decision diagram

dargestellt. Das linke BDD ist weder frei noch geordnet, das rechte ist frei, geordnet und reduziert. Seine Variablenordnung ist $x_1 < x_2 < x_3$.

Ein BDD ist genau dann nicht reduziert, wenn einer von folgenden drei Fällen eintritt [2]:

- Zwei Blätter tragen dieselbe Beschriftung.
- Es gibt einen Knoten v mit $\text{low}(v) = \text{high}(v)$.
- Es gibt zwei Knoten u, v mit $\text{label}(u) = \text{label}(v)$, $\text{low}(u) = \text{low}(v)$ und $\text{high}(u) = \text{high}(v)$.

Zu jedem geordneten BDD B kann ein reduziertes OBDD für dieselbe Boolesche Funktion in linearer Zeit in Anzahl der Knoten von B berechnet werden. Die verfügbaren Implementierungen von BDDs gehen jedoch einen anderen Weg: Sie sorgen bereits bei der Konstruktion der BDDs dafür, daß sie reduziert sind.

Die große Bedeutung für den Einsatz von OBDDs in der Verifikation erklärt der folgende Satz:

Satz 3.1 (Bryant, 1986) *Reduzierte und geordnete BDDs sind für eine feste Variablenordnung eine kanonische Darstellung Boolescher Funktionen.*

Beweis:
Siehe [2] oder [23]. □

Insbesondere lassen sich der Test, ob eine Funktion erfüllbar ist und ob die Funktion eine Tautologie ist, in konstanter Zeit realisieren: Eine Funktion ist genau dann erfüllbar, wenn ihr OBDD verschieden vom 0-Blatt ist. Sie ist eine Tautologie genau dann, wenn ihr OBDD nur aus dem 1-Blatt besteht.

Definition 3.4 *Ein BDD heißt* ***vollständig****, wenn auf jedem Pfad von der Wurzel zu einem Blatt jede Variable genau einmal vorkommt.*

Ein BDD sei ***1-vollständig****, wenn auf jedem Pfad von der Wurzel zum 1-Blatt jede Variable genau einmal vorkommt.*

Hierbei ist zu beachten, daß vollständige bzw. 1-vollständige BDDs im Allgemeinen nicht reduziert und somit auch nicht kanonisch sind. Insbesondere gibt es für dieselbe Funktion in der Regel mehrere vollständige bzw. 1-vollständige BDDs.

3.1. Algorithmen auf OBDDs

In diesem Abschnitt werden kurz die wichtigsten Synthese-Operationen für OBDDs vorgestellt, die wir in späteren Kapiteln verwenden werden. Dazu gehören die Booleschen Verknüpfungen und die Berechnung eines OBDDs für die charakteristische Funktion eines Constraints, wie er bei 0/1-ILP-Problemen vorkommt.

Für die Algorithmen brauchen wir den Begriff des Kofaktors nach einer Variablen x:

Definition 3.5 *Sei $f : \{0,1\}^n \to \{0,1\}$ oder $f : \{0,1\}^n \to \mathbb{Z}$ eine Funktion, die von x_i abhängt. Dann ist der **positive Kofaktor** von f nach x_i definiert als*

$$f_{x_i}(x_1, \ldots, x_n) = f(x_1, \ldots, x_{i-1}, 1, x_{i+1}, \ldots, x_n).$$

*Analog ist der **negative Kofaktor** nach x_i definiert als*

$$f_{\overline{x}_i}(x_1, \ldots, x_n) = f(x_1, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_n).$$

Um eine exponentielle Laufzeit zu vermeiden, werden bei den hier vorgestellten OBDD-Algorithmen die Zwischenergebnisse in Hashtabellen (sog. ComputedTables) abgespeichert. Um nur reduzierte OBDDs aufzubauen, wird überprüft, ob bereits ein OBDD für die gesuchte Funktion berechnet ist. Dazu dient die UniqueTable.

3.1.1. Boolesche Operationen auf OBDDs

Sämtliche binäre Boolesche Operationen lassen sich auf einen ternären Operator, der als ITE-Operator[3] bezeichnet wird, zurückführen. Er ist definiert durch:

$$\mathrm{ITE}(f, g, h) = f \cdot g + \overline{f} \cdot h$$

Beispielsweise ist $f \cdot g = \mathrm{ITE}(f, g, 0)$ oder $f + g = \mathrm{ITE}(f, 1, g)$ oder $\overline{f} = \mathrm{ITE}(f, 0, 1)$.

Wegen

$$\mathrm{ITE}(f, g, h) = x \cdot \mathrm{ITE}(f_x, g_x, h_x) + \overline{x} \cdot \mathrm{ITE}(f_{\overline{x}}, g_{\overline{x}}, h_{\overline{x}})$$

läßt sich der ITE-Operator einfach rekursiv berechnen (siehe auch [2]). Wir bezeichnen der Einfachheit halber das OBDD für eine Funktion f mit F.

Algorithmus 3.1 (ITE)

```
1  ITE(F, G, H) {
2     if (terminal case) return result;
3     if ( (F,G,H) ∈ ComputedTable ) return ComputedTable(F, G, H);
4     x = topVar(F, G, H);
5     R_x = ITE(F_x, G_x, H_x);
6     R_x̄ = ITE(F_x̄, G_x̄, H_x̄);
7     if (R_x == R_x̄) return R_x;
8     R = find_or_add_UniqueTable(x, R_x, R_x̄);
9     ComputedTable(F, G, H) = R;
10    return R;
11 }
```

Der Algorithmus hat eine Laufzeit in $O(|F| \cdot |G| \cdot |H|)$.

[3] ITE: if-then-else

3.1.2. Berechnung von charakteristischen Funktionen

Sei im Folgenden eine Funktion $f : \{0,1\}^n \to \mathbb{Z}$ gegeben. Wir sind interessiert an der charakteristischen Funktion χ_f, die wie folgt definiert ist:

$$\chi_f(x) = \begin{cases} 1 & \text{falls } f(x) \leq 0 \\ 0 & \text{sonst} \end{cases}.$$

Da die charakteristische Funktion eine Boolesche Funktion ist, läßt sie sich als OBDD darstellen. Der folgende Algorithmus berechnet zu einer Funktion f das OBDD für χ_f:

Algorithmus 3.2 (leqZero)

```
1  LeqZero(f) {
2      if ( max(f) ≤ 0 ) return bddOne;
3      if ( min(f) > 0) return bddZero;
4      if (f ∈ ComputedTable) return ComputedTable(f);

5      Sei x ∈ support(f);
6      R_x = LeqZero(f_x);
7      R_x̄ = LeqZero(f_x̄);
8      if (R_x == R_x̄) return R_x;
9      R = ITE(x, R_x, R_x̄);
10     ComputedTable(f) = R;
11     return R;
12 }
```

Die Zahl der rekursiven Aufrufe des Algorithmus ist beschränkt durch die Zahl der verschiedenen Kofaktoren von f. Diese kann allerdings exponentiell sein. Das bedeutet, es wird Funktionen geben, für die der Algorithmus exponentielle Laufzeit hat[4].

3.2. Variablenordnungen

Ein bekanntes Problem von OBDDs betrifft die Variablenordnung: So gibt es Funktionen, bei denen die Zahl der Knoten stark von der Variablenordnung abhängt: Verwendet man die Variablenordnung

$$x_1 < x_2 < \cdots < x_{2n-1} < x_{2n},$$

[4]Betrachte beispielsweise die Hidden Weighted Bit Funktion (HWB) und setze $f_n(X) = 1 - \mathrm{HWB}_n(X)$ mit $X = (x_1, \ldots, x_n)$. Dann ist $\chi_{f_n}(X) = \mathrm{HWB}_n(X)$. Bekanntermaßen benötigt HWB_n unabhängig von der Variablenordnung exponentiell viele Knoten.

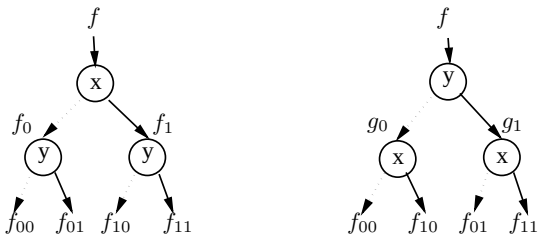


Abbildung 3.2.: Vertauschen zweier Level

so läßt sich die Funktion $f(x_1, \ldots, x_{2n}) = \sum_{i=1}^{n} x_{2i-i} x_{2i}$ mit $2n$ inneren Knoten darstellen. Wählt man stattdessen die Ordnung

$$x_1 < x_3 < \cdots < x_{2n-1} < x_2 < x_4 < \cdots < x_{2n},$$

dann braucht man mehr als 2^n innere Knoten [23, Theo. 5.3.3]. Verschlimmert wird die Situation dadurch noch, daß das Problem, eine bessere Variablenordnung für ein gegebenes OBDD zu finden, NP-hart ist.

Da die Laufzeit für die meisten OBDD-Algorithmen stark von der Knotenzahl abhängt, wurden verschiedene Verfahren zur heuristischen Verbesserung der Variablenordnung entwickelt. Die statischen Verfahren versuchen vor dem Aufbauen eines BDDs ausgehend von der Problemstruktur eine gute Ordnung zu finden. Solche statische Verfahren, die speziell auf unsere 0/1-ILP-Probleme zugeschnitten sind, werden wir in Abschnitt 6.1.1 kennenlernen.

Die dynamischen Verfahren versuchen, während das BDD aufgebaut wird, die Variablenordnung zu verbessern. Eines der wichtigsten dynamischen Verfahren ist Sifting [19], das auch wir einsetzen werden.

Sifting beruht auf der Vertauschung zweier benachbarter Variablenlevel (siehe Abbildung 3.2). Dies ist für OBDDs eine lokale Operation und kann bei geeigneter Implementierung effizient durchgeführt werden.

Sifting geht nun so vor, daß nacheinander jede Variable durch Vertauschen mit dem Vorgänger bis zur Wurzel und danach von der Wurzel bis zum untersten Level bewegt wird. Anschließend wird die Variable zu der Position zurückbewegt, die die OBDD-Größe minimiert. Für Details siehe [2, 19, 23].

Es gibt jedoch auch Funktionen, die unabhängig von der Variablenordnung eine exponentielle Zahl von OBDD-Knoten benötigen. Dazu gehören leider so wichtige Funktionen wie die Binärmultiplikation, wie bereits Bryant in [4] gezeigt hat.

4. Ein reines OBDD-Verfahren für 0/1-ILP

In diesem Abschnitt wird ein reines OBDD-Verfahren zur Lösung von 0/1-ILP-Problemen vorgestellt. Es geht zurück auf ein Verfahren, das von Lai, Pedram und Vrudhula [10] entwickelt wurde. Dieses wurde von mir als Studienarbeit implementiert, um das Potential von OBDDs bei der Optimierung abzuschätzen. Außerdem haben wir darauf aufbauend ein Verfahren gefunden, das bei einigen Problemen zu signifikanten Laufzeitverbesserungen gegenüber dem ursprünglichen Verfahren führt.

Wir gehen in diesem Kapitel immer davon aus, daß ein 0/1-ILP-Problem wie in Definition 2.1 gegeben ist.

4.1. Darstellung des Lösungsraums

Beide Verfahren beruhen darauf, daß der komplette Lösungsraum des gegebenen 0/1-ILP-Problems als OBDD dargestellt wird. Seien für $i = 1, \ldots, m$ die Constraints gegeben als $a_i^T x \leq b_i$. Die charakteristische Funktion des i-ten Constraints ist dann definiert als

$$\chi_i(x) = \begin{cases} 1 & \text{falls } a_i^T x \leq b_i \\ 0 & \text{sonst.} \end{cases}$$

Die charakteristische Funktion χ des Lösungraums ist

$$\chi(x) = \bigwedge_{i=1}^{m} \chi_i(x) = \begin{cases} 1 & \text{falls } Ax \leq b \\ 0 & \text{sonst.} \end{cases}$$

Unserer Vorgehensweise ist folgende: Wir erzeugen zuerst ein OBDD für die charakteristische Funktion jedes einzelnen Constraints. Dies geschieht mit Hilfe von Algorithmus 3.2 durch folgenden Aufruf:

$$B_i = \text{leqZero}(a_i^T x - b)$$

Als Ergebnis erhalten wir eine Liste von OBDDs, die wir mit AND verknüpfen, um so ein OBDD für χ zu erhalten.

Dabei merkt man schnell, daß die Größe der Zwischenergebnisse und damit auch die Laufzeit stark von der Reihenfolge und der Variablenordnung abhängen. Um die

Variablenordnung zu optimieren, führen wir, solange die Zahl der OBDD-Knoten ein vorgegebenes Limit nicht übersteigt, immer dann Sifting aus, wenn die Zahl der Knoten um einen festen Faktor angewachsen ist.

Um die Reihenfolge der Konjunktionen festzulegen, haben wir einige Heuristiken entwickelt:

- Verknüpfe immer die beiden OBDDs mit der geringsten Knotenzahl.
- Verknüpfe immer die beiden OBDDs, die am wenigsten erfüllende Belegungen besitzen.
- Verknüpfe immer solche OBDDs, die von ungefähr denselben Variablen abhängen.
- Verwende die Ordnung, wie sie in der Problemspezifikation vorgegeben ist.

Es hat sich gezeigt, daß hierbei keine Heuristik den anderen überlegen ist, daß es aber deutliche Unterschiede zwischen den einzelnen Heuristiken geben kann.

4.2. Direktes Minimieren

In diesem Abschnitt wollen wir das Verfahren vorstellen, mit dem wir den optimalen Wert der Zielfunktion bestimmen. Es handelt sich dabei um ein typisches Branch & Bound-Verfahren: Wir führen eine obere und eine untere Schranke für die Lösung mit. Wenn wir feststellen, daß der optimale Wert eines Teilproblems sich nicht innerhalb der Schranken befindet, können wir die Suche in diesem Teilproblem abbrechen.

Der Wert für die untere Schranke ist der optimale Wert der LP-Relaxierung. Für die obere Schranke wird zu Beginn $o = \max(c^T x)+1$ gewählt. Wir suchen nur gültige Belegungen, die besser als die obere Schranke sind. Dadurch, daß wir zu Beginn die obere Schranke auf $\max(c^T x)+1$ setzen, schließen wir anfangs keine gültige Belegung aus.

Immer wenn wir eine Belegung finden, die einen besseren Wert als die obere Schranke liefert, passen wir die obere Schranke daran an.

Der Algorithmus bekommt als Eingabe die zu minimierende Zielfunktion g, das OBDD B für die gültigen Belegungen sowie die obere und untere Schranke o und u. Er liefert `true` zurück, wenn der Wert der oberen Schranke verbessert wurde, sonst `false`.

Algorithmus 4.1 (minimize)

```
1 minimize(g, B, o, u) {
2     // Basisfälle überprüfen.
3     if (B == bddZero) return false;
```

```
4   if (min g ≥ o) return false;
5   if (isConstant(g)) {
6       o = g(0);
7       return true; }
8   if (B == bddOne) {
9       o = min g;
10      return true; }

11  // ComputedTable überprüfen.
12  localbound = o − g(0);
13  f(x) = g(x) − g(0);
14  entry = computedTable(f, G);
15  if (entry ≠ NULL) {
16      if (entry.value < entry.bound) {
17          if (entry.bound < localbound) {
18              o = entry.value + g(0);
19              return true;
20          } else return false;
21      } else if (localbound ≤ entry.bound) return false;
22  }

23  // Rekursive Aufrufe
24  x = topVar(B);
25  entry.bound = localbound;
26  if (min f_x < min f_x̄) {
27      hret = minimize(f_x, B_x, localbound, u − g(0));
28      lret = minimize(f_x̄, B_x̄, localbound, u − g(0));
29  } else {
30      lret = minimize(f_x̄, B_x̄, localbound, u − g(0));
31      hret = minimize(f_x, B_x, localbound, u − g(0));
32  }

33  // Lösung überprüfen
34  if (lret || hret) {
35      o = localbound + g(0);
36      entry.value = localbound;
37      computedTable(f, B) = entry;
38      return true;
39  } else {
40      entry.value = entry.bound;
41      computedTable(f, B) = entry;
```

```
42          return false;
43      }
44  }
```

Zu diesem Algorithmus sind einige Erläuterungen notwendig: Der Algorithmus besteht aus vier Teilen: Im ersten Teil (Zeilen 1 – 10) werden die Basisfälle überprüft, in denen die Rekursion abgebrochen werden kann. Dies ist beispielsweise der Fall, wenn es keine erfüllbaren Belegungen mehr gibt.

Der zweite Teil (Zeilen 11 – 22) dient zur Überprüfung der ComputedTable. Um die Trefferrate zu erhöhen, wird die Zielfunktion normalisiert durch $f(x) = g(x) - g(0)$ und entsprechend die obere Schranke durch localbound $= o - g(0)$. Wenn ein Eintrag in der ComputedTable gefunden wurde, muß überprüft werden, ob er verwendet werden kann. Dies hängt von der oberen Schranke zu dem Zeitpunkt, als der Eintrag generiert wurde, ab. Diese ist in entry.bound gespeichert. In entry.value steht, falls der Aufruf, der den Eintrag generierte, erfolgreich war, der Wert der Lösung. Ansonsten ist der Wert von entry.value gleich entry.bound.
Wenn also entry.value < entry.bound ist, dann war der letzte Aufruf erfolgreich. Falls jetzt zusätzlich noch entry.value < localbound ist, dann verbessert dieser Eintrag unsere aktuelle Lösung. Wir setzen $o =$ entry.value $+ g(0)$ und geben true zurück. Im anderen Falle geben wir false zurück und aktualisieren o nicht.
Es kann jedoch auch sein, daß localbound zu dem Zeitpunkt, als der Eintrag generiert wurde, kleiner war als jetzt. Wenn dann der Aufruf erfolglos war, können wir daraus keinen verwertbaren Schluß ziehen und müssen weitersuchen.

Der dritte Teil von minimize (Zeilen 23 – 32) führt die rekursiven Aufrufe für die Kofaktoren aus. Es hat sich gezeigt, daß die Laufzeit besser ist, wenn zuerst der Aufruf für den Kofaktor ausgeführt wird, der das kleinere Minimum besitzt.

Der letzte Teil (Zeilen 33 – 44) wertet die Ergebnisse der rekursiven Aufrufe aus und setzt daraus das zurückzugebende Ergebnis und den Eintrag für die ComputedTable zusammen.

4.2.1. Probleme und Lösungen

Das größte Problem, auf das man bei diesem Verfahren stößt, ist, daß das OBDD für die charakteristische Funktion so groß wird, daß es nicht mehr in den Hauptspeicher paßt. Dieses Problem tritt bei der Verknüpfung der OBDDs für die einzelnen Constraints auf. Selbst wenn die OBDDs für die einzelnen χ_i klein sind, kann das OBDD für χ zu groß werden. Deshalb führen wir einen Parameter limit ein. Bei der AND-Verknüpfung der OBDDs werden nur solche verknüpft, deren Knotenzahl kleiner als limit ist. Bleibt am Schluß mehr als ein OBDD übrig, dann werden von allen OBDDs und der Zielfunktion die beiden Kofaktoren nach der obersten in den OBDDs vorkommenden Variablen berechnet. Das Verfahren wird dann rekursiv auf

die beiden Teilprobleme angewendet und am Schluß die bessere Lösung genommen.

Algorithmus 4.2 (solve)

```
1  solve(g, constraints, o, u) {
2      if (max(g) < u) return false;
3      if (min(g) ≥ o) return false;
4      if (bddZero ∈ constraints) return false;
5      new_constraints = conjunction(constraints);
6      if (|new_constraints| == 1) return minimize(g, new_constraints.first, o, u);
7      else {
8          x = topVar(new_constraints);
9          lret = solve(g_x, new_constraints_x, o, u);
10         hret = solve(g_x̄, new_constraints_x̄, o, u);
11         return (lret || hret);
12     }
13 }
```

4.3. Binäre Suche

Experimentelle Ergebnisse haben beim bisher vorgestellten Verfahren gezeigt, daß der Algorithmus oft schnell eine gute oder sogar optimale Lösung findet, er aber danach trotzdem noch das gesamte OBDD absuchen muß, um die Optimalität zu zeigen. Darüber hinaus ist es bei einem gegebenen OBDD möglich, in linearer Zeit in Anzahl der Variablen (nicht der Knoten!) eine beliebige Lösung zu finden.

Um diese Beobachtungen auszunutzen gehen wir folgendermaßen vor: Seien o und u eine obere bzw. untere Schranke für die Lösung. Dann setzen wir

$$m := \left\lfloor \frac{o+u}{2} \right\rfloor$$

und nehmen als zusätzlichen Constraint

$$g(x) \leq m$$

hinzu. Wenn eine Lösung mit Wert l existiert, dann wissen wir, daß die optimale Lösung im Bereich zwischen u und l liegen muß. Wir setzen dann $o := l$ und wiederholen das Verfahren.

Wenn keine Lösung existiert, dann muß – wenn das Problem überhaupt lösbar ist – die Lösung im Bereich zwischen $m+1$ und o liegen. Setze also $u := m+1$ und wiederhole.

Sobald $u \geq o$ ist, wird das Verfahren abgebrochen. Die letzte gefundene Lösung ist optimal.

Das Verfahren läßt sich folgendermaßen in Pseudocode formulieren:

Algorithmus 4.3 (binsolve)

```
1  binsolve(c, goal, u, o) {
2      c = conjunctConstraints(c);
3      solution = NULL;
4      while (u < o) {
5          m = (u + o) div 2;
6          c' = addConstraint(c, "goal – m ≤ 0");
7          sol = getAnySolution(c');
8          if (sol == NULL) u = m + 1;
9          else {
10             o = goal(sol);
11             solution = sol;
12         }
13     }
14     return solution;
15 }
```

Der Parameter c ist eine Liste mit den BDDs für die charakteristischen Funktionen der Constraints, goal die Zielfunktion, u und o sind eine untere bzw. obere Schranke für die Lösung. Die Funktion getAnySolution liefert einen Pfad im BDD c' von der Wurzel zum 1-Blatt zurück, wenn ein solcher existiert, und NULL sonst.

4.3.1. Probleme und Lösungen

Doch auch dieses Verfahren ist nicht frei von Problemen: Es hat sich gezeigt, daß das OBDD für den zusätzlichen Constraint $g(x) \leq m$ oft sehr groß wird. Dies liegt daran, daß ein normaler Constraint meist nur von wenigen Variablen abhängt. Die Zielfunktion jedoch hängt in der Regel von allen Variablen ab. Wenn die Zielfunktion symmetrisch ist, dann ist das OBDD für den zusätzlichen Constraint klein. In diesen Fällen hat sich die binäre Suche dem ersten Verfahren deutlich überlegen gezeigt.

Um auch Probleme, deren Zielfunktion nicht symmetrisch ist, in den Griff zu bekommen, machen wir von der folgenden Beobachtung Gebrauch: wenn es für einen Kofaktor der charakteristischen Funktion der „normalen“ Constraints keine erfüllenden Belegung mehr gibt, so spielt der zusätzliche Constraint keine Rolle mehr und wir können diesen Kofaktor der charakteristischen Funktion des zusätzlichen Constraints auf Null setzen.

Wir haben den Algorithmus LeqZero (Algorithmus 3.2) entsprechend angepaßt: Er besitzt jetzt einen zusätzlichen Parameter B für das OBDD der charakteristischen Funktion der „normalen“ Constraints.

Algorithmus 4.4 (leqZero)

```
1  LeqZero(f, B) {
```

```
2    if ( max(f) ≤ 0 ) return bddOne;
3    if ( min(f) > 0) return bddZero;
4    if ( B == bddZero) return bddZero;
5    if ((f, B) ∈ ComputedTable) return result;

6    Sei x ∈ support(f);
7    R_x = LeqZero(B_x, f_x);
8    R_x̄ = LeqZero(B_x̄, f_x̄);
9    if (R_x == R_x̄) return R_x;
10   R = ITE(x, R_x, R_x̄);
11   ComputedTable(f, B) = R;
12   return R;
13 }
```

In Zeile 4 ist der neue Basisfall hinzugekommen. Da das Ergebnis vom übergebenen OBDD B abhängt, mußte in den Zeilen 5 und 11 die ComputedTable erweitert werden, so daß das übergebene OBDD mit abgespeichert wird.

Wenn die Zielfunktion symmetrisch ist, verwenden wir die alte Variante von leqZero, da diese dort schneller ist. Ist die Zielfunktion nicht symmetrisch, verwenden wir die neue Variante.

Bemerkung 4.1 *Alle verwendeten Algorithmen (leqZero, minimize, binäre Suche, solve) machen keinerlei Annahmen über die Struktur der Constraints und der Zielfunktion. Das bedeutet, die Algorithmen können nicht nur verwendet werden, um 0/1-ILP-Probleme zu lösen, sondern bei beliebigen 0/1-Optimierungsproblemen. Als Operationen auf den beteiligten Funktionen sind nur die Kofaktorbildung und die Berechnung von Minimum und Maximum notwendig.*

4.4. Experimentelle Ergebnisse

Sowohl die binäre Suche als auch das direkte Minimieren wurden mit Hilfe des OBDD-Packages CUDD [22] implementiert. Als Benchmark-Daten dienten einige 0/1-ILP-Probleme aus der Kollektion MIPLIB [3].[1]

In der folgenden Tabelle sind die Laufzeiten für einige Probleme zusammengestellt. Als Vergleich wurden die Laufzeiten des schnellsten freien ILP-Solvers lp_solve 4.0 angegeben.

[1] Wir konnten hier nicht dieselben Benchmarks wie in Abschnitt 6.7.1 verwenden, da bei den hier vorgestellten Ansatz die BDDs für die hfo-Benchmarks zu groß wurden.

Problem	direktes Minimieren	binäre Suche	lp_solve 4.0
stein9	0.01	0.001	0.01
stein15	0.02	0.01	0.03
stein27	0.84	0.32	3.96
stein45	328.15	98.848	259.11
p0033	0.10	0.60	0.34
p0040	0.04	18.19	0.01
bm23	11.24	53.36	0.07

Die stein*-Probleme sind eine Reihe von Problemen mit einer symmetrischen Zielfunktion, bei denen die herkömmlichen ILP-Solver bekanntermaßen recht lange Laufzeiten haben. Unser Verfahren (insbesondere die binäre Suche) scheint damit gut zurechtzukommen.

Die Zielfunktion der p*-Reihe ist nicht symmetrisch. Dort hat die binäre Suche Schwierigkeiten, das OBDD für den zusätzlichen Constraint aufzubauen. Entsprechend groß sind dort die Laufzeiten.

Das Problem bm23 liegt zwischen den beiden Extremen.

5. Die „klassischen“ ILP-Verfahren

In diesem Kapitel soll kurz das heutzutage in den meisten ILP-Solver verwendete Verfahren besprochen werden. Es ist eine Kombination aus einem Branch & Bound-Verfahren und einem Verfahren, das auf der Erzeugung von Schnittebenen beruht.

Wir werden die beiden Verfahren zuerst getrennt betrachten und dann zeigen, wie sie kombiniert werden können.

Dieses Kapitel folgt zu großen Teilen den Übersichtsartikeln von Mitchell [14, 15, 16].

5.1. Branch & Bound

Ein typisches Branch & Bound-Verfahren geht folgendermaßen vor: Durch die Aufteilung des Problems („Branch“) in mehrere einfachere Teilprobleme wird das Problem vereinfacht. Die optimale Lösung des gesamten Problems ergibt sich als das Optimum der Lösungen der Teilprobleme.

Ist bei einem Teilproblem vor der rekursiven Anwendung des Verfahrens bereits klar, daß es die optimale Lösung nicht enthalten kann, so braucht es nicht weiter betrachtet zu werden („Pruning“).

Wir beschränken uns hier auf 0/1-ILP. Das vorgestellte Branch & Bound-Verfahren kann aber leicht auf allgemeine ILP-Probleme verallgemeinert werden.

Wir werden in diesem Abschnitt sehen, wie der Lösungsraum aufgeteilt werden kann und wie man sehen kann, daß ein Teilproblem nicht zur optimalen Lösung führt.

Das ursprüngliche Problem $\min\{c^T x \mid Ax \leq b, x \in \{0,1\}^n\}$ bezeichnen wir mit $ILP^{(0)}$. Wir speichern alle ungelösten Teilprobleme in einer Liste L. Zu Beginn gilt also $L = \{ILP^{(0)}\}$.

Um die Lösung von Teilproblemen abbrechen zu können, müssen wir eine obere und eine untere Schranke für die Lösung speichern. Sie werden mit o bzw. u bezeichnet. Die untere Schranke stammt aus einer Relaxierung des Problems, die obere Schranke ist der Wert der bisher besten Lösung.

Insgesamt kann dann das Branch & Bound-Verfahren durch folgenden Algorithmus beschrieben werden:

Algorithmus 5.1

1. *Setze $o = +\infty$ und $u = -\infty$. Sei $L = \{ILP^{(0)}\}$.*

2. *Falls $L = \emptyset$ ist, dann ist derjenige Punkt x^*, an dem der Wert von o angenommen wurde, optimal. Wenn $o = \infty$ und kein solcher Punkt existiert, dann ist das Problem unlösbar.*

3. *Wähle und lösche ein Teilproblem $ILP^{(k)}$ aus L. Löse eine Relaxierung von $ILP^{(k)}$ (also z. B. eine LP-Relaxierung oder eine Lagrange-Relaxierung). Sei δ_k der optimale Wert der Relaxierung und sei $x^{(k)}$ die optimale Lösung der Relaxierung, falls eine solche existiert (d. h. $\delta_k = c^T x^{(k)}$ oder $\delta^{(k)} = -\infty$).*

4. *Falls $\delta^{(k)} \geq o$ gehe zu 2.*
   Falls $\delta^{(k)} < o$ und $x^{(k)} \in \{0,1\}^n$, dann setze $u := \delta^{(k)}$ und lösche aus L all diejenigen Teilprobleme, deren untere Schranke größer als o ist. Gehe anschließend zu 2.

5. *Teile das Problem in zwei Teile auf, z. B. indem eine Variable im einen Teilproblem auf 0 und im anderen auf 1 gesetzt wird. Füge die beiden Teilprobleme in L ein und gehe zu 2.*

Um das Verfahren auf allgemeine ILP-Probleme anwenden zu können, muß lediglich überall $\{0,1\}$ durch $\mathbb{Z}$ ersetzt und Schritt 5 angepaßt werden. Wir teilen dort das Problem in zwei Teilprobleme auf, indem wir einmal für ein geeignetes $m \in \mathbb{Z}$ den Constraint $x_i \leq m$ und einmal $x_i \geq m + 1$ hinzufügen.

Als offene Fragen bleiben noch, welches Teilproblem in Schritt 3 und welche Variable für die Partitionierung in Schritt 5 ausgewählt werden sollen. Dafür gibt es inzwischen zahlreiche Heuristiken, von denen jeweils einige exemplarisch vorgestellt werden sollen.

Auswahl des Teilproblems

- Die zu lösenden Teilprobleme bilden einen binären Baum mit dem ursprünglichen Problem an der Wurzel. Da sind die beiden Durchlaufreihenfolgen Tiefensuche und Breitensuche naheliegend. Die Tiefensuche hat den Vorteil, daß man meist schnell gültige Lösungen findet und somit die obere Schranke anpassen kann.

- Eine weitere Möglichkeit ist, immer den Knoten in Schritt 3 auszuwählen, der die größte untere Schranke besitzt. Man hofft, daß man davon viele Teilprobleme abschneiden kann, wenn deren untere Schranke größer als der Wert der bisher besten Lösung ist.

- Wähle dasjenige Teilproblem aus, dessen optimale LP-Lösung $x^{(k)}$ am nächsten an einem $\{0,1\}$-Vektor liegt oder am weitesten entfernt ist, d. h.

$$s^{(k)} = \sum_{i=1}^{n} \min \left\{ x_i^{(k)} - \left\lfloor x_i^{(k)} \right\rfloor,\ \left\lceil x_i^{(k)} \right\rceil - x_i^{(k)} \right\}$$

  minimiert bzw. maximiert.

Bisher gibt es keine Heuristik, die in allen Fällen den anderen überlegen ist.

Auswahl der Branching-Variablen Wenn eine Lösung x^* der LP-Relaxierung des Teilproblems vorliegt, dann ist es die Standard-Strategie, diejenige Variable x_j für das Branching auszuwählen, deren Wert x_j^* am nächsten bei $\frac{1}{2}$ liegt.

Es gibt aber Heuristiken, die in vielen Fällen eine bessere Laufzeit liefern. Dazu gehört z. B. die Schätzung der Pseudo-Kosten der einzelnen Variablen (siehe [12]). Die Idee hinter dieser Heuristik ist, die Informationen über den Erfolg der Branching-Variablen auszunutzen, die man bei bereits gelösten Teilproblemen verwendet hat.

Sei P die Menge der Teilprobleme (außer dem Wurzelknoten), die bereits gelöst worden sind. Anfangs ist diese Menge leer. P^+ bezeichne die Menge aller rechten Söhne, bei denen die Branching-Variable auf 1 gesetzt worden ist, P^- die Menge der linken Söhne, d. h. $P = P^+ \dot\cup P^-$. Für ein Problem $p \in P$ sei

$f(p)$ der Vater von p

$v(p)$ der Index der Branching-Variablen, die beim Übergang von $f(p)$ zu p auf einen festen Wert gesetzt wurde.

$x(p)$ die optimale Lösung der LP-Relaxierung im Knoten p

$z(p)$ der Wert der optimalen LP-Lösung $x(p)$.

Die oberen Pseudo-Kosten einer Variablen x_j sind dann

$$\Phi^+(x_j) = \frac{1}{|P_{x_j}^+|} \sum_{p \in P_{x_j}^+} \frac{z(p) - z(f(p))}{\left\lceil x_{v(p)}(f(p)) \right\rceil - x_{v(p)}(f(p))}.$$

Entsprechend sind die unteren Pseudo-Kosten definiert als

$$\Phi^-(x_j) = \frac{1}{|P_{x_j}^-|} \sum_{p \in P_{x_j}^-} \frac{z(p) - z(f(p))}{x_{v(p)}(f(p)) - \left\lfloor x_{v(p)}(f(p)) \right\rfloor}.$$

Die Mengen $P^+_{x_j}$ und $P^-_{x_j}$ sind definiert durch

$$P^+_{x_j} = \{p \in P^+ \mid v(p) = x_j\} \qquad \text{und} \qquad P^-_{x_j} = \{p \in P^- \mid v(p) = x_j\},$$

falls x_j bereits einmal als Splitvariable verwendet worden ist. Sonst ist $P^+_{x_j} = P^+$ und $P^-_{x_j} = P^-$.

Da diese Heuristik nicht gerade leicht zu verstehen ist, will ich ein paar Erläuterungen dazu angeben: Betrachten wir zuerst die Zähler der Brüche: $z(p) - z(f(p))$. Dies ist der Wert, um den sich die Zielfunktion beim Übergang vom Vater $f(p)$ zum Teilproblem p verändert hat. Die Nenner $\lceil x_{v(p)}(f(p)) \rceil - x_{v(p)}(f(p))$ bzw. $x_{v(p)}(f(p)) - \lfloor x_{v(p)}(f(p)) \rfloor$ der Brüche geben die Änderung des Wertes der Splitvariablen an, die beim Übergang von $f(p)$ zu p verwendet wurde. Die Änderung der Zielfunktion wird also durch die Brüche mit dem Unterschied im Wert der Splitvariblen normiert.

Der Wert $\Phi^-(x_j)$ bzw. $\Phi^+(x_j)$ ergibt sich dann als arithmetisches Mittel der normierten Änderungen von allen Teilproblemen $p \in P^-_{x_j}$ bzw. $p \in P^+_{x_j}$.

Jetzt ist nur noch die Frage zu klären, wie stark die oberen und unteren Pseudo-Kosten gewichtet werden sollen. Dazu sei $\alpha \in [0, 1]$ ein vorher festgelegter Wert (z. B. $\alpha = \frac{1}{2}$. Wähle als Split-Variable diejenige Variable x_j, die

$$\Phi(x_i) = \alpha\Phi^+(x_i) + (1 - \alpha)\Phi^-(x_i)$$

maximiert.

Eine Schwäche dieses Verfahrens ist der Anfang, wenn nur wenige gelöste Teilprobleme zur Verfügung stehen. Dann ist der Wert von Φ für alle Variablen nahezu identisch und damit wenig aussagekräftig.

Es gibt zahlreiche noch weitaus ausgefeiltere Heuristiken sowohl zur Auswahl der Split-Variablen als auch zur Auswahl des nächsten Teilproblems. Für eine kurze Beschreibung des Branch & Bound-Ansatzes sollten die obigen Ausführungen jedoch genügen.

5.2. Schnittebenenverfahren

Es gibt eine zweite Klasse von erfolgreichen Verfahren zur Lösung von ILP-Problemen, die Schnittebenenverfahren. Die Idee, die hinter diesen Verfahren steckt, beruht auf der Existenz von Schnittebenen (siehe dazu Satz 2.2). Solange die Lösung x^* der LP-Relaxierung nicht ganzzahlig ist, finde eine Schnittebene, die gültig für das ganze ILP-Polytop, aber nicht für x^* ist. Füge die Schnittebene zum Problem hinzu. Dieses Verfahren wird iteriert, bis die Lösung der Relaxierung ganzzahlig ist. Es ergibt sich der folgende Algorithmus:

Algorithmus 5.2

1. *Löse die LP-Relaxierung des Problems. Sei x^* dessen optimale Lösung.*
2. *Falls $x^* \in \mathbb{Z}^n$ ist, dann ist x^* die optimale Lösung des ganzzahligen Problems. Fertig.*
3. *Berechne eine Schnittebene, die gültig für das ILP-Polytop, aber nicht für x^* ist, und füge sie zum Problem hinzu.*
4. *Gehe zu 1.*

Im Gegensatz zum Branch & Bound-Verfahren aus dem letzten Abschnitt ist hier die Terminierung nicht unbedingt gesichert. Werden die Schnittebenen nach bestimmten Regeln erzeugt, wie z. B. die Gomory-Cuts [7], oder wenn die Schnittebenen Facetten sind, dann kann man zeigen, daß das Verfahren nach endlich vielen Schritten terminiert.

Bevor wir zeigen, wie man die beiden vorgestellten Verfahren kombinieren kann, wollen wir das Schnittebenenverfahren an einem Beispiel veranschaulichen:

Beispiel 5.1 *Betrachte das ganzzahlige lineare Programm*

$$\begin{array}{lrcrcrl} \min & -2x_1 & - & x_2 & & & \\ & x_1 & + & 2x_2 & \leq & 7 & \\ & 2x_1 & - & x_2 & \leq & 3 & \\ & x_1 & & & \geq & 0 & \textit{ganzzahlig} \\ & & & x_2 & \geq & 0 & \textit{ganzzahlig} \end{array}$$

In Abbildung 5.1 ist das Problem graphisch dargestellt. Wir wenden das Schnittebenenverfahren an, um die optimale ganzzahlige Lösung zu bestimmen:

- *1. Schritt: Lösung der LP-Relaxierung liefert $x^* = (2.6, 2.2)^T \notin \mathbb{Z}^2$. Wir berechnen also eine Schnittebene, die x^* abschneidet. Die Ungleichung $x_1 + x_2 \leq 4$ leistet beispielsweise das Gewünschte. Sie wird zum Problem hinzugenommen.*
- *2. Schritt: Erneute Lösung der LP-Relaxierung liefert: $x^* = \left(2\frac{1}{3}, 1\frac{2}{3}\right)^T \notin \mathbb{Z}^2$. Wir brauchen eine weitere Schnittebene, beispielsweise $x_1 \leq 2$.*
- *3. Schritt: Als neue Lösung der LP-Relaxierung erhalten wir: $x^* = (2, 2)^T \in \mathbb{Z}^2$. Da x^* jetzt ganzzahlig ist, haben wir die optimale Lösung gefunden.*

Verfahren, wie solche Schnittebenen für 0/1-ILP automatisch generiert werden können, werden wir im nächsten Kapitel kennenlernen.

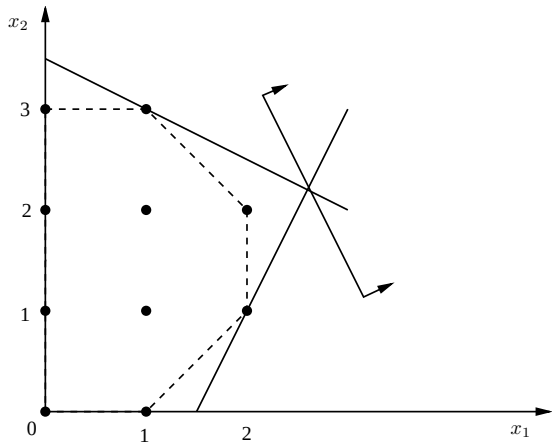


Abbildung 5.1.: Beispiel für das Schnittebenenverfahren

5.3. Branch & Cut

Es ist naheliegend, die beiden bisher vorgestellten Verfahren zu kombinieren, da es beim Branch & Bound-Verfahren notwendig ist, eine gute untere Schranke zu besitzen, um möglichst viele Teilprobleme abschneiden zu können, und mit dem Schnittebenenverfahren die Schranken verbessert werden können.

Man kann zwei Varianten unterscheiden: Die sogenannten Cut & Branch-Verfahren fügen Schnittebenen nur im Wurzelknoten hinzu, die anderen auch in anderen Knoten des Branch & Bound-Baums. Die Schnittebenen nur im Wurzelknoten hinzuzufügen hat den Vorteil, daß sie für alle anderen Knoten auch gültig sind. Außerdem wird der Speicherplatzbedarf reduziert, da sich das zu lösende Problem von Knoten zu Knoten nur geringfügig ändert. Andererseits hat sich gezeigt, daß einige schwierige Probleme mit einem Cut & Branch-Verfahren nicht oder nur sehr schwer zu lösen sind. In diesen Fällen sind die Verfahren, die Schnittebenen auch in den anderen Knoten erzeugen, oft erfolgreicher.

Wir wollen uns das Prinzip an einem einfachen Beispiel verdeutlichen (siehe auch [14]):

Beispiel 5.2 *Betrachte das ganzzahlige lineare Programm*

$$\begin{array}{lrcrcll} \min & -5x_1 & - & 6x_2 & & & \\ & x_1 & + & 2x_2 & \leq & 7 & \\ & 2x_1 & - & x_2 & \leq & 3 & \\ & x_1 & & & \geq & 0 & \textit{ganzzahlig} \\ & & & x_2 & \geq & 0 & \textit{ganzzahlig} \end{array}$$

In Abbildung 5.2 ist das Problem graphisch dargestellt. Wir lösen zuerst die LP-Re-

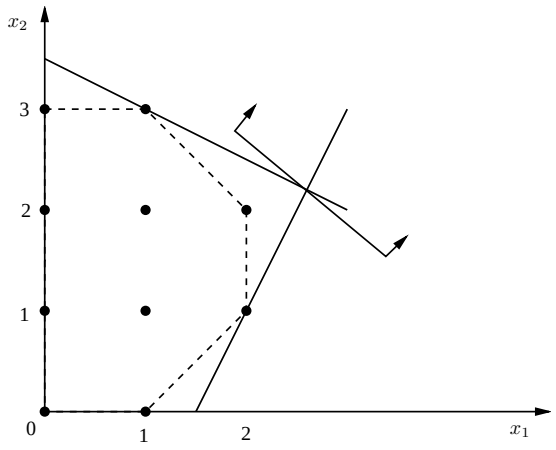


Abbildung 5.2.: Beispiel für Branch & Cut

laxierung des Problems und erhalten $x^* = (2.6, 2.2)^T$ *mit Wert* -26.2. *Wir haben nun zwei Möglichkeiten: Endweder teilen wir das Problem in zwei Teilprobleme auf und lösen diese rekursiv oder wir berechnen eine Schnittebene, um die untere Schranke zu verbessern. Nehmen wir an, wir wählen die erste Möglichkeit und splitten bezüglich* x_2. *Dadurch erhalten wir folgende Teilprobleme:*

1. Teilprobleme:

$$\begin{array}{lrcrcrcl} \min & -5x_1 & - & 6x_2 & & & & \\ & x_1 & + & 2x_2 & \leq & 7 & & \\ & 2x_1 & - & x_2 & \leq & 3 & & \\ & & & \mathbf{x_2} & \geq & \mathbf{3} & & \\ & x_1 & & & \geq & 0 & \textit{ganzzahlig} & \\ & & & x_2 & \geq & 0 & \textit{ganzzahlig} & \end{array}$$

2. Teilproblem:

$$\begin{array}{lrcrcrcl} \min & -5x_1 & - & 6x_2 & & & & \\ & x_1 & + & 2x_2 & \leq & 7 & & \\ & 2x_1 & - & x_2 & \leq & 3 & & \\ & & & \mathbf{x_2} & \leq & \mathbf{2} & & \\ & x_1 & & & \geq & 0 & \textit{ganzzahlig} & \\ & & & x_2 & \geq & 0 & \textit{ganzzahlig} & \end{array}$$

Die optimale Lösung des Gesamtproblems ist die bessere der beiden Lösungen der Teilprobleme.

Als Lösung der LP-Relaxierung des 1. Teilproblems erhalten wir $x_1^* = (1, 3)^T$ *mit Wert* -23. *Da* x_1^* *ganzzahlig ist, haben wir die optimale Teillösung gefunden.*

Die Lösung der LP-Relaxierung des 2. Teilproblems ist $x_2^* = (2.5, 2)^T$ *mit Wert* -24.5. *Wir haben wieder die beiden Möglichkeiten, entweder das Problem aufzuteilen oder eine Schnittebene zu erzeugen. Dieses Mal gehen wir den zweiten Weg. Beispielsweise ist* $x_1 + 2x_2 \leq 6$ *eine Ungleichung, die für das zweite Teilproblem gültig ist. Außerdem verletzt* x_2^* *die Ungleichung. Sie ist somit eine gültige Schnittebene. Fügen wir sie zum zweiten Teilproblem hinzu und lösen wieder die LP-Relaxierung, erhalten wir* $x_3^* = (2.4, 1.8)^T$ *mit Wert* -22.6. *Da dieser Wert schlechter ist als die Lösung des 1. Teilproblems, brauchen wir das zweite nicht länger zu betrachten und*

sind fertig.

Die Lösung des Gesamtproblems ist $x = (1, 3)^T$ *mit Wert* -23.

Anhand dieses Beispiels sollte die prinzipielle Funktionsweise der Branch & Cut-Verfahren klar geworden sein. Trotzdem sind einige Punkte zu klären: Vor allem, wann soll eine Schnittebene generiert werden und wann ist es besser, das Problem aufzuteilen?

In der Regel lohnt es sich nicht, in jedem Knoten Schnittebenen zu erzeugen, da die Teilprobleme sonst stark anwachsen. Außerdem kostet das Erzeugen von Schnittebenen Zeit. Deshalb werden Schnittebenen beispielsweise nur in jedem achten Knoten oder nur in Knoten, deren Tiefe ein Vielfaches von Acht ist, erzeugt. Oder man erzeugt Schnittebenen immer dann, wenn die LP-Relaxierung des Teilproblems nur wenig unterhalb der bisher besten Lösung liegt, weil man hofft, die Schranke so weit zu verbessern, daß das Teilproblem abgeschnitten werden kann.

Im nächsten Kapitel werden wir eine Technik namens Lifting kennenlernen, mit deren Hilfe man Schnittebenen, die man an einem Knoten erzeugt hat, auch für andere Knoten gültig machen kann.

6. Schnittebenengenerierung für 0/1-ILP mit OBDDs

In diesem Kapitel wollen wir zeigen, wie man Schnittebenen, wie sie im Branch & Cut-Verfahren aus dem vorigen Kapitel benötigt werden, effizient unter Verwendung von OBDDs erzeugen kann.

6.1. Aufbau des OBDDs

Wir werden wie in Abschnitt 4.1 ein OBDD für die charakteristische Funktion der Constraints aufbauen. Um das Problem, daß das OBDD zu groß wird, zu vermeiden, gehen wir hier allerdings einen anderen Weg: Wir wählen eine geschickte Teilmenge der Constraints aus, und erzeugen das OBDD nur für deren charakteristische Funktion.

Außerdem beschreiben wir, wie wir die übrigen Constraints, die nicht im BDD berücksichtigt worden sind, verwenden können, um einerseits die Überapproximation zu verbessern und andererseits das BDD zu verkleinern. Dieselbe Methode können wir auch anwenden, um das BDD zu verkleinern, wenn wir eine gültige (aber nicht notwendigerweise optimale) Lösung des ILP-Problems kennen.

6.1.1. Initiale Variablenordnung

Bevor wir anfangen, ein BDD aufzubauen, müssen wir eine initiale Variablenordnung festlegen. Daß eine gute Variablenordnung von enomer Wichtigkeit ist, sieht man, wenn man das MIPLIB-Problem [3] `p0040` betrachtet: Wählt man eine gute Variablenordnung, so besteht das BDD für den Lösungsraum aus weniger als 800 Knoten. Bei einer schlechten Variablenordnung hingegen braucht man für das BDD mehr als 20 000 Knoten!

Leider ist das Problem, eine optimale Variablenordnung für eine Boolesche Funktion zu finden, NP-hart [23, Kap. 5.4]. Wir sind folglich auf Heuristiken angewiesen.

Wir haben verschiedene Heuristiken untersucht:

NaturalOrder Es wird die natürliche Variablenordnung

$$x_1 < x_2 < \cdots < x_n$$

verwendet.

SumOrder Betrachten wir folgenden Constraint:

$$5x_1 - x_2 - 2x_3 \leq 1.$$

Man sieht leicht ein, daß bei allen erfüllenden Belegungen $x_1 = 0$ gelten muß, weil der Koeffizient von x_1 deutlich größer als die anderen Koeffizienten ist. Dies führt uns zu folgender Idee: Variablen, deren Koeffizienten betragsmäßig groß sind, sind wichtig und gehören an den Anfang der Variablenordnung. Wir definieren deshalb für jede Variable x_j einen Wert h_j wie folgt:

$$h_j = \sum_{i=0}^{m} |a_{ij}|$$

und sortieren die Variablen absteigend nach ihrem h_j-Wert.

NormSumOrder Diese Heuristik funktioniert ganz ähnlich wie die vorige. Der einzige Unterschied ist, daß die Constraints normalisiert werden, d. h. man ersetzt $a^T x \leq b$ durch $\frac{1}{\|a\|} a^T x \leq \frac{b}{\|a\|}$.

Experimente, deren Ergebnisse in Tabelle 6.1 dargestellt sind, haben schnell gezeigt, daß solche Heuristiken, die lediglich die Größe der Koeffizienten, nicht aber die Struktur der Constraint, d. h. die Verteilung der von Null verschiedenen Koeffizienten in der Constraint-Matrix, berücksichtigen, relativ schlechte Ordnungen liefern.[1]

Deshalb haben wir folgende Idee verfolgt: Partitioniere die Constraints geschickt in paarweise disjunkte Teilmengen. Für jede Teilmenge wird unabhängig von den anderen eine Variablenordnung mit Hilfe einer der oben genannten Heuristiken bestimmt. Mit Hilfe eines Verfahrens, das sich Interleaving [6] nennt, werden die Variablenordnungen der einzelnen Teilmengen zu einer einzigen Variablenordnung zusammengesetzt.

Doch was ist eine „geschickte“ Partitionierung? Sie sollte folgende Eigenschaften erfüllen:

- Die Teilmengen sind paarweise disjunkt, nicht leer und jeder Constraint ist in einer Teilmenge enthalten.
- Die Constraints in einer Teilmenge sollten einen ähnlichen Support haben, d. h. von ungefähr denselben Variablen abhängen.

[1] NaturalOrder ist eine Ausnahme, weil die Benchmarks meistens so spezifiziert sind, daß die Problemstruktur berücksichtig wird.

- Die Zahl der Variablen, die in einer Teilmenge von Constraints vorkommen, sollte nicht zu groß werden, damit beim Interleaving möglichst viele Teilmengen berücksichtigt werden können.

Zwei Verfahren, die von Marc Herbstritt und Thomas Kmieciak in [8] für die Partitionierung von Schaltkreisausgängen entwickelt wurde, erfüllen diese Anforderungen.

Partitionierung der Constraints Grundlegend für die Partitionierung ist die Information, wieviele Variablen zwei Constraints gemeinsam haben. Sie wird in der Output-Correspondence-Matrix gespeichert.

Definition 6.1 *Seien die Constraints $c_1, \ldots, c_m$ gegeben. Sei ferner* $\text{support}(c_i)$ *die Menge der Variablen, von denen der Constraint c_i abhängt.*
Dann ist die ***Output-Correspondence-Matrix*** $OCM \in \mathbb{R}^{m \times m}$ *definiert durch*

$$OCM_{ij} = |\,\text{support}(c_i) \cap \text{support}(c_j)|.$$

Wir gehen bei der Partitionierung folgendermaßen vor: Wir erzeugen eine neue, noch leere Teilmenge M_j und wählen aus der Menge C der Constraints denjenigen aus, der von den meisten Variablen abhängt, löschen ihn aus C und fügen ihn in M_j ein. Wir nennen diesen Constraint den Leader von M_j. Dann entfernen wir nacheinander alle Constraints, die das Partitionierungskriterium erfüllen aus C und fügen sie in M_j ein, bis kein Constraint mehr das Kriterium erfüllt. Dann erzeugen wir eine neue Teilmenge M_{j+1} und beginnen am Anfang.
Folgende zwei Kriterien werden verwendet:

Definition 6.2 (WOG-Kriterium) *Ein Constraint c_i wird in die Teilmenge M_j aufgenommen, wenn sein Support* $\text{support}(c_i)$ *eine Teilmenge des Supports des Leaders von M_j ist.*

Definition 6.3 (BOG-Kriterium) *Ein Constraint c_i wird in die Teilmenge M_j aufgenommen, wenn sei Support* $\text{support}(c_i)$ *eine Teilmenge des Supports jedes anderen Elements von M_j ist.*

Für eine detaillierte Untersuchung dieser Verfahren, und wie sie mit Hilfe der Ouput-Correspondence-Matrix effizient implementiert werden können, verweisen wir auf den Technischen Bericht [8].

Zur Veranschaulichung führen wir das Verfahren an einem konkreten Beispiel durch:

Beispiel 6.1 *Betrachten wir die folgenden vier Constraints:*

$$\begin{aligned} c_1 &: & x_1 + x_2 + x_3 + x_4 &\leq 2 \\ c_2 &: & 2x_1 + 3x_2 &\leq 2 \\ c_3 &: & x_3 - x_4 &\leq 0 \\ c_4 &: & x_3 + 2x_4 - x_5 &\leq 2 \end{aligned}$$

Die Support-Mengen der Constraints sind:

$$\begin{aligned} \text{support}(c_1) &= \{x_1, x_2, x_3, x_4\} \\ \text{support}(c_2) &= \{x_1, x_2\} \\ \text{support}(c_3) &= \{x_3, x_4\} \\ \text{support}(c_4) &= \{x_3, x_4, x_5\} \end{aligned}$$

Folglich hat die Output-Correspondence-Matrix OCM folgende Form:

$$OCM = \begin{pmatrix} 4 & 2 & 2 & 2 \\ 2 & 2 & 0 & 0 \\ 2 & 0 & 2 & 2 \\ 2 & 0 & 2 & 3 \end{pmatrix}$$

WOG: *Wegen* $\text{support}(c_2) \subseteq \text{support}(c_1)$ *und* $\text{support}(c_3) \subseteq \text{support}(c_1)$ *erhalten wir als erste Teilmenge* $\{c_1, c_2, c_3\}$*. Es bleibt nur noch* c_4 *übrig, da* $\text{support}(c_4) \not\subseteq \text{support}(c_1)$*. Also erzeugt das WOG-Kriterium die Partitionierung* $\{\{c_1, c_2, c_3\}, \{c_4\}\}$*.*

BOG: *Die erste Teilmenge ist* $\{f_1, f_2\}$*, weil* $\text{support}(f_3) \not\subseteq \text{support}(f_2)$*. Der Leader der nächsten Teilmenge ist* f_4*, weil* $|\text{support}(f_4)| > |\text{support}(f_3)|$*. Weil* $\text{support}(f_3) \subseteq \text{support}(f_4)$ *ist die zweite Teilmenge* $\{f_4, f_3\}$*. Also erzeugt das BOG-Kriterium die Partitionierung* $\{\{f_1, f_2\}, \{f_4, f_3\}\}$*.*

Unsere Situation ist nun folgende: Wir haben eine Partitionierung der Constraints berechnet. Nun bestimmen wir für jede Teilmenge mit Hilfe der bereits beschriebenen Heuristiken eine partielle Variablenordnung. Dabei lassen wir die Position all der Variablen unspezifiziert, die nicht in einem Constraint aus der gerade betrachteten Teilmenge vorkommen.

Die so entstehenden partiellen Ordnungen sortieren wir aufsteigend nach der Anzahl der vorkommenden Variablen, weil das Interleaving mehr Teilordnungen berücksichtigen kann, wenn wir die kurzen Ordnungen an den Anfang stellen.

Num erzeugen wir eine totale Ordnung auf den Variablen, indem wir das Verfahren Interleaving anwenden.

Interleaving Das Verfahren Interleaving wurde von Fujii et. al. [6] entwickelt, um eine gute Variablenordnung für Boolesche Funktionen $f : \{0,1\}^n \to \{0,1\}^m$ mit mehreren Ausgängen zu finden.

In Pseudocode läßt sich der Algorithmus folgendermaßen formulieren:

Algorithmus 6.1 (Interleaving)

```
1  interleaving(variable orders pvo) {
2      output = ∅;
3      for each partial order o ∈ pvo {
4          for each variable v in o from top {
5              if (v ∉ output) {
6                  if (v is top of o) {
7                      insert v as top of output;
8                  } else {
9                      let w be the variable just before v in output;
10                     insert v just after w in output.
11                 }
12             }
13         }
14     }
15     return output;
16 }
```

Wir haben nun eine Reihe von Variablenordnungen bestimmt. Die Tabelle 6.1 enthält für einige Benchmark-Probleme und einige Variablenordnungen die Größen der zugehörigen BDDs. Die Größenverteilung ist typisch auch für andere Benchmarks. Deshalb verwenden wir in Zukunft immer OCMOrder als Heuristik zur Bestimmung einer guten Variablenordnung.

6.1.2. Bau des OBDDs

Nachdem wir eine gute Variablenordnung bestimmt haben, können wir beginnen, BDDs aufzubauen.

Sei x^* die Lösung der LP-Relaxierung. Dann gibt es eine Teilmenge der Constraints, die in x^* mit Gleichheit erfüllt sind, da x^* eine Ecke des LP-Polytops ist. Davon wählen wir eine maximale Teilmenge aus, so daß die Normalenvektoren a_i linear unabhängig sind. Sie sei o. B. d. A. gegeben durch $T = \{a_1^T x \leq b_1, \ldots, a_k^T x \leq b_k\}$. Der Punkt x^* ist weiterhin optimale LP-Lösung in dem Polytop, das durch die Constraints aus T beschrieben ist.

Solange das OBDD für die charakteristische Funktion der Constraints aus T hinreichend klein ist, fügen wir nacheinander weitere Constraints hinzu, um die Approximation des Lösungsraums zu verbessern.

Heuristik	**p0033**	**p0040**	**stein15**	**stein27**
NaturalOrder	377	832	761	25204
SumOrder	1604	22286	761	25172
NormSumOrder	795	22286	761	25172
RandomOrder[a]	1505	41308	702	48211
OCMOrder<NaturalOrder>(BOG)	265	708	649	44560
OCMOrder<SumOrder>(BOG)	529	708	649	44560
OCMOrder<NormSumOrder>(BOG)	535	708	649	44560
OCMOrder<RandomOrder>(BOG)[a]	470	678	650	46612
OCMOrder<NaturalOrder>(WOG)	198	707	779	25172
OCMOrder<SumOrder>(WOG)	580	707	779	25204
OCMOrder<NormSumOrder>(WOG)	562	707	779	25204
OCMOrder<RandomOrder>(WOG)[a]	592	673	753	48645

[a] Mittel über 100 zufällige Ordnungen

Tabelle 6.1.: BDD-Größen für einige Probleme aus der MIPLIB [3]

Wir erhalten dadurch eine Überapproximation des Lösungsraums. Das bedeutet, wenn eine Schnittebene für die Teilmenge der Constraints, für die wir das OBDD aufgebaut haben, gültig ist, dann ist die auch für das ursprüngliche Polytop gültig.

6.1.3. Verkleinerung des BDDs

Auf zwei Arten läßt sich das BDD z. T. deutlich verkleinern, so daß anschließend die Schnittebenengenerierung schneller erfolgen kann:

Verkleinerung durch Fixieren von Variablen

Das OBDD läßt sich mit Hilfe der folgenden Beobachtung verkleinern: Oft sind in x^* einige Einträge auf 0 bzw. 1 gesetzt, d. h. wir können die Indexmenge $I = \{1, \ldots, n\}$ der Variablen in drei Teile partitionieren: $I = I_0 \dot\cup F \dot\cup I_1$ mit

- I_0 enthält die Indizes der Variablen, deren Wert in x^* gleich 0 ist.
- I_1 enthält die Indizes der Variablen, deren Wert in x^* gleich 1 ist.
- In F sind die Variablen mit einem Wert aus $(0, 1)$ enthalten.

Da der Punkt x^* nicht ganzzahlig ist (sonst hätten wir die optimale Lösung gefunden), gilt $F \neq \emptyset$.

Wir setzen in den Constraints alle Variablen x_i mit $i \in I_0$ auf 0 und alle mit $i \in I_1$ auf 1. Dann berechnen wir dafür die charakteristische Funktion (d. h. wir beschränken uns auf einen Kofaktor der ursprünglichen charakteristischen Funktion).

Wenn wir mit Hilfe dieses OBDDs eine Schnittebene erzeugen, ist sie natürlich nur für einen Teilraum des ursprünglichen Lösungsraums gültig. Deshalb müssen wir eine Technik namens Lifting anwenden, um die Schnittebene für das ganze Polytop (bzw. die Überapproximation) gültig zu machen. Dies werden wir im Abschnitt 6.6 beschreiben. Ein Nachteil dieses Verfahrens ist, daß die geliftete Schnittebene nicht mehr ganz so stark ist wie eine Schnittebene, die ohne Fixierung für das ganze Polytop erzeugt wurde. Allerdings kann durch Einsatz dieses Verfahrens manchmal das BDD aufgebaut werden, wenn dies ohne Variablenfixierungen nicht mehr gelingt.

Verkleinern durch Löschen unnötiger Knoten und Kanten

Angenommen, wir haben einen Constraint

$$a^T x \leq b$$

gegeben, von dem wir wissen, daß die optimale Lösung diesen Constraint erfüllen muß, dessen charakteristische Funktion aber nicht im BDD enthalten ist. Dazu gehören beispielsweise alle Constraints, die nicht am Aufbau des BDDs beteiligt sind.

Manchmal kennt man (durch Verwendung einer Heuristik o. ä.) eine gültige Lösung $\hat{x}$, die aber nicht notwendigerweise optimal ist. Dann erhalten wir einen gültigen Constraint durch

$$c^T x \leq c^T \hat{x}.$$

All diese Constriants können wir verwenden, um das BDD zu verkleinern.

Uns brauchen nur solche ganzzahligen Punkte aus dem Polytop zu interessieren, die diesen Zusatzconstraint $a^T x \leq b$ erfüllen.

Eine naheliegende Möglichkeit wäre, für die charakteristische Funktion des Zusatzconstraints das BDD aufzubauen und es mit dem BDD für die normalen Constraints zu verknüpfen. Dadurch kann sich aber die Knotenzahl des BDDs erheblich vergrößern, so daß dieses Verfahren im Allgemeinen nicht praktikabel ist.

Stattdessen gehen wir anders vor: Die Punkte des Polytops, die den Zusatzconstraint erfüllen, entsprechen Pfaden im 1-vollständigen BDD von der Wurzel zum 1-Blatt, deren Länge kleiner oder gleich b ist, wenn wir die Kostenfunktion folgendermaßen wählen:

$$c(e) = \begin{cases} 0 & \text{falls } \mathrm{par}(e) = 0 \\ a_i & \text{falls } \mathrm{par}(e) = 1 \text{ und } \mathrm{label}(\mathrm{head}(e)) = x_i \end{cases} \tag{6.1}$$

Damit wir auch mit nicht 1-vollständigen BDDs arbeiten können, verallgemeinern wir die Kostenfunktion so, daß sie auch mit Kanten $e = (u, v)$ umgehen kann, die ein oder mehrere Level überspringen. Wir wählen dazu die Kosten von e wie die Kosten des kürzesten Weges von u nach v im entsprechenden 1-vollständigen BDD.

Wenn wir das BDD 1-vollständig machen würden, dann müßten wir für jedes Level, das übersprungen wird, einen redundanten Knoten einfügen, dessen beide ausgehenden Kanten auf denselben Knoten zeigen. Auf dem kürzesten Weg wählen wir deshalb immer die billigere von beiden ausgehenden Kanten, d h. die high-Kante, falls deren Kosten negativ sind, und ansonsten die low-Kante, deren Kosten immer Null betragen. Die Variablenordnung sei durch eine Permutation π gegeben, die jeder Variablen das zugehörige Level zuordnet. Wir erhalten damit für unsere verallgemeinerte Kostenfunktion:

$$c^*(e) = c(u,v) + \sum_{i=\text{level}(u)+1}^{\text{level}(v)-1} \min\{0, a_{\pi^{-1}(i)}\}. \tag{6.2}$$

Wir berechnen nun bezüglich der Kostenfunktion c^* für jeden inneren Knoten v die Entfernung $\text{dist}(\mathit{root}, v)$ von der Wurzel zu v und $\text{dist}(v, \mathit{leaf1})$ von v zum 1-Blatt. Die Länge des kürzesten Pfades $\text{dist}(v)$ von der Wurzel zum 1-Blatt, der über einen gegebenen Knoten $v \in V$ geht, berechnet sich zu

$$\text{dist}(v) = \text{dist}(\mathit{root}, v) + \text{dist}(v, \mathit{leaf1}).$$

Entsprechend ist die Länge $\text{dist}(e)$ des kürzesten Pfads von der Wurzel zum 1-Blatt durch eine gegebene Kante $e = (u, v) \in E$

$$\text{dist}(e) = \text{dist}(\mathit{root}, u) + c(e) + \text{dist}(v, \mathit{leaf1}).$$

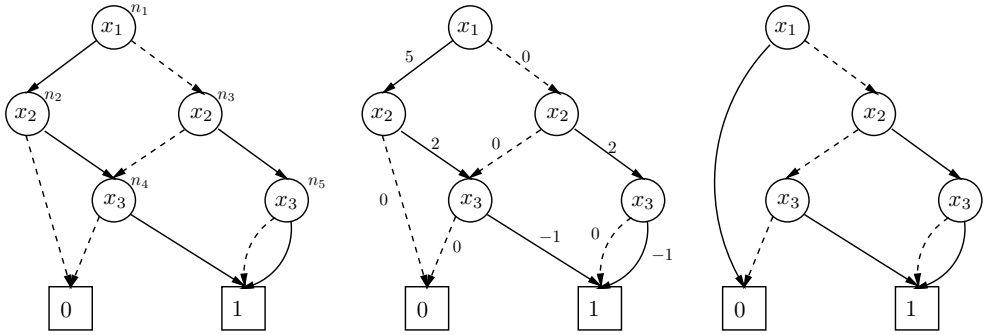
Wir können nun alle Knoten v mit $\text{dist}(v) > b$ und alle Kanten e mit $\text{dist}(e) > b$ aus dem BDD löschen. Einen Knoten v löschen wir, indem wir alle in v eingehenden Kanten auf das 0-Blatt umlenken. Zu löschende Kanten lenken wir ebenfalls auf das 0-Blatt um. Bevor wir den Algorithmus angeben, mit dem wir die unnötigen Knoten löschen und Kanten umlenken, wollen wir das Prinzip an einem Beispiel veranschaulichen:

Beispiel 6.2 *Nehmen wir an, wir haben linke das BDD aus Abbildung 6.1 und den folgenden Constraint gegeben:*

$$5x_1 + 2x_2 - x_3 \leq 5$$

Wir weisen allen 0*-Kanten das Gewicht* 0 *zu, allen* 1*-Kanten, die von* x_1*-Knoten ausgehen, das Gewicht* 5*, allen* 1*-Kanten, die in* x_2*-Knoten beginnen, das Gewicht* 2 *und den* 1*-Kanten auf dem untersten Level das Gewicht* -1 *zu. Diese Situation ist in Abbildung 6.1b dargestellt.*

Wenn wir für jeden Knoten die Länge des kürzesten Pfades, der durch ihn hindurchführt, berechnen, erhalten wir folgende Werte:



(a) vor dem Löschen (b) mit Kantengewichten (c) nach dem Löschen

Abbildung 6.1.: Beispiel zum Löschen unnötiger Knoten und Kanten

Knoten	n_1	n_2	n_3	n_4	n_5
Entfernung	-1	6	-1	-1	1

Wir sehen, daß kein Pfad, der durch den Knoten n_2 geht, den obigen Constraint erfüllt. Wir entfernen den Knoten n_2 und lenken alle eingehenden Kanten auf das 0-Blatt um. Dadurch erhalten wir das BDD in Abbildung 6.1c.

Nachdem wir das Prinzip der Löschung verdeutlicht haben, wird im Folgenden der Algorithmus angegeben, der diese Löschung durchführt:

Algorithmus 6.2

```
1  deleteNodes(BDD bdd, dist, b) {
2      if (bdd == bddZero || bdd == bddOne) return bdd;
3      if (bdd ∈ ComputedTable) return ComputedTable(bdd);
4      if (dist(bdd.root) > b) return bddZero;

5      BDD lowSon = bdd.low;
6      if (dist(bdd.root, lowSon.root) > b) lowResult = bddZero;
7      else lowResult = deleteNodes(lowSon, dist, b);

8      BDD highSon = bdd.high;
9      if (dist(bdd.root, highSon.root > b) highResult = bddZero;
10     else highResult = deleteNodes(highSon, dist, b);

11     BDD result = ITE(bdd.topVar, highResult, lowResult);
12     ComputedTable(bdd) = result;

13     return result;
14 }
```

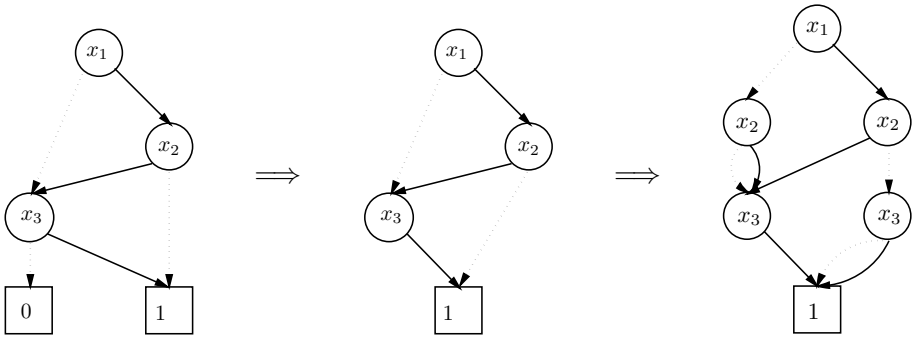


Abbildung 6.2.: Erzeugung eines 1-vollständigen BDDs

Unter der Annahme, daß wir in konstanter Zeit auf die ComputedTable zugreifen können, hat der Algorithmus lineare Laufzeit in der Zahl der BDD-Knoten.

Die Vorteile des Algorithmus liegen auf der Hand: Durch das Löschen der unnötigen Knoten und Kanten wird das BDD kleiner, so daß die nachfolgend die Schnittebenen schneller erzeugt werden können. Außerdem wird die Überapproximation des Lösungsraums verbessert. Dadurch können bessere Schnittebenen erzeugt werden. Weiterhin kann die Löschung bereits bei der Konjunktion der einzelnen BDDs für die Constraints durchgeführt werden. Dadurch kann – wenn die Löschung sinnvoll eingesetzt wird – die Zeit für den Aufbau des BDDs verringert werden.

Experimente haben gezeigt, daß sich das BDD bei vielen Problemen beträchtlich verkleinern läßt, insbesondere dann, wenn man den Zusatzconstraint wie oben beschrieben aus einer gültigen Lösung konstruiert.

6.2. BDD- und Flußpolytope

In diesem Abschnitt werden wir mit Hilfe des BDDs, wie er im vorigen Abschnitt beschrieben wurde, das Problem in ein höherdimensionales Polytop transformieren. Nach der Beschreibung des höherdimensionalen Polytops werden wir den Zusammenhang mit dem ursprünglichen Polytop aufzeigen.

Um die Darstellung einfach zu halten, gehen wir davon aus, daß wir in diesem Abschnitt immer ein 1-vollständiges BDD, dessen 0-Blatt entfernt worden ist, für eine Überapproximation des Lösungsraums gegeben haben. Dies läßt sich leicht erreichen, wie in Abbildung 6.2 zu sehen ist. Obwohl das Ergebnis der Transformation streng genommen kein OBDD mehr ist, bezeichnen wir es weiterhin als BDD.

Sei $\mathcal{P}$ die Menge der Pfade von der Wurzel von B zum 1-Blatt. Jeder Pfad $p \in \mathcal{P}$ entspricht einer Variablenbelegung der Variablen x_i durch den charakteristischen

Vektor χ_p von p. Es ist $\chi_p(x_i) = \text{par}(e)$, falls e diejenige Kante aus p ist, deren Anfangsknoten mit x_i beschriftet ist. Die konvexe Hülle der charakteristischen Vektoren bildet das BDD-Polytop:

Definition 6.4 (BDD-Polytop) *Das Polytop, das durch ein BDD beschrieben wird, ist gegeben durch*

$$P_{BDD} = \text{conv}\{\chi_p \,|\, p \in \mathcal{P}\}$$

Das BDD-Polytop stimmt also mit dem ILP-Polytop der Constraints überein, aus denen das BDD aufgebaut wurde.

Wir beschreiben jetzt ein höherdimensionales 0/1-Polytop, dessen Variablen nicht den Knotenbeschrifungen wie beim BDD-Polytop entsprechen, sondern den Kanten des BDDs. Dazu erweitern wir den Graphen $G = (V, E)$ des BDDs zu einem Flußnetzwerk. Für jede Kante e des BDDs sei f_e eine Variable, die den Fluß durch die Kante e angibt. Die Wurzel sei die einzige Quelle des Flußnetzwerkes mit ausgehenden Fluß 1. Die einzige Senke ist das 1-Blatt mit einem eingehenden Fluß von 1. In allen anderen Knoten fordern wir, das der Fluß, der in den Knoten hineingeht, gleich dem Fluß ist, der den Knoten verläßt.

Dies führt zu folgender Beschreibung:

Definition 6.5 (Flußpolytop)

$$\begin{aligned} P_{flow} = \Big\{ f \in \mathbb{R}^{|E|} \,|\, & \sum_{e \in \delta^-(v)} f_e - \sum_{e \in \delta^+(v)} f_e = 0 \quad \textit{für alle } v \in V \setminus \{\textit{root},\ \textit{leaf1}\}, \\ & - \sum_{e \in \delta^+(\textit{root})} f_e = -1 \\ & \sum_{e \in \delta^-(\textit{leaf1})} f_e = 1 \\ & 0 \le f_e \le 1 \quad \textit{für alle } e \in E \Big\} \end{aligned} \tag{6.3}$$

Es gilt jetzt das folgende Lemma:

Lemma 6.1 *Die Polytope P_{BDD} und P_{flow} sind beides 0/1-Polytope mit derselben Zahl von Ecken.*

Beweis:

Sei das Flußpolytop gegeben durch die Constraints $Af = b$ und $0 \le f \le 1$. Dann ist A die Inzidenzmatrix des BDD-Graphen. Sei I die Einheitsmatrix der Dimension $|E| \times |E|$.

Wir können das Flußpolytop folglich auch schreiben als

$$P_{flow} = \left\{ f \in \mathbb{R}^{|E|} \;\middle|\; \begin{pmatrix} A \\ -A \\ I \end{pmatrix} f \leq \begin{pmatrix} b \\ -b \\ \mathbf{1} \end{pmatrix}, f \geq 0 \right\}.$$

Die Inzidenzmatrix A ist nach Satz 2.5 total unimodular. Wenn A total unimodular ist, dann auch $\begin{pmatrix} A \\ -A \\ I \end{pmatrix}$. Nach dem Satz von Hoffmann/Kruskal (Satz 2.4) ist P_{flow} ein ganzzahliges 0/1-Polytop.

In einem ganzzahligen 0/1-Polytop ist jeder ganzzahlige Punkt eine Ecke, wie man sich leicht klarmacht. Da aber jeder ganzzahlige Fluß im BDD ein Pfad p von der Wurzel zum 1-Blatt ist, kann ihm über den charakteristischen Vektor χ_p eine Variablenbelegung zugeordnet werden, die eine Ecke des BDD-Polytops darstellt.

Umgekehrt gilt: Zu jeder Ecke des BDD-Polytops gibt es einen Pfad von der Wurzel zum 1-Blatt. Dieser stellt einen ganzzahligen Fluß im Fluß-Polytop dar und ist folglich eine Ecke von P_{flow}. □

Wenn wir einen beliebigen (also nicht notwendigerweise ganzzahligen) Fluß haben, dann möchten wir daraus eine gültige Variablenbelegung für die x_i berechnen können. Dies kann mit Hilfe der folgenden linearen Gleichungen geschehen:

$$x_i = \sum_{\substack{e \in E \\ \text{head}(e) = x_i}} \text{par}(e) \cdot f_e \tag{6.4}$$

Um den Wert von x_i zu bestimmen, summieren wir also die Flüsse der high-Kanten, die von Knoten mit Beschriftung x_i ausgehen, auf.

Wir wollen uns diesen Zusammenhang zumindest plausibel machen: Sei f ein ganzzahliger Fluß. Dann ist auf jedem Level der Fluß genau einer Kante gleich Eins; alle anderen sind Null. Wenn eine low-Kante den Wert Eins hat, denn muß die zugehörige Variable den Wert Null bekommen. Ist der Wert einer high-Kante gleich Eins, so bekommt die zugehörige Variable den Wert Eins. In der Summe 6.4 ist – wenn der Fluß ganzzahlig ist – höchstens ein f_e verschieden von Null und dann gleich Eins. Wenn es sich um eine high-Kante handelt, ist $\text{par}(e) = 1$ und somit auch $x_i = 1$. Ansonsten gilt $\text{par}(e) = 0$ und $x_i = 0$.

Wir fügen für $i = 1, \ldots, n$ die Gleichungen (6.4) zur Beschreibung von P_{flow} in 6.3 hinzu und erhalten so das Polytop $P(f, x) \subset \mathbb{R}^{|E|+n}$, wobei $|E|$ die Zahl der Kanten des BDD ist.

Definition 6.6 *Die **Projektion** von $P(f, x)$ auf $\mathbb{R}^n$ ist definiert durch*

$$\text{Proj}_x(P(f, x)) = \{x \in \mathbb{R}^n \mid \exists f \in \mathbb{R}^{|E|} : (f, x) \in P(f, x)\}$$

Wir zeigen jetzt, daß die Projektion von $P(f,x)$ auf x gerade wieder das BDD-Polytop ergibt.

Satz 6.1

$$P_{BDD} = \mathrm{Proj}_x(P(f,x))$$

Beweis:

„$\supseteq$“ Sei $(f^*, x^*) \in P(f,x)$. Der Fluß f^* kann geschrieben werden als Konvexkombination von Ecken f^j von P_{flow} für j aus einer geeigneten Indexmenge J, d. h.

$$f^* = \sum_{j \in J} \lambda_j f^j$$

mit $\lambda_j \geq 0$ für alle $j \in J$ und $\sum\limits_{j \in J} \lambda_j = 1$. Es gilt

$$\begin{aligned} x_i^* &= \sum_{\substack{e \in E \\ \mathrm{head}(e) = x_i}} \mathrm{par}(e) f^*(e) \\ &= \sum_{\substack{e \in E \\ \mathrm{head}(e) = x_i}} \mathrm{par}(e) \cdot \left(\sum_{j \in J} \lambda_j f_e^j \right) \\ &= \sum_{j \in J} \lambda_j \sum_{\substack{e \in E \\ \mathrm{head}(e) = x_i}} \mathrm{par}(e) f_e^j \end{aligned}$$

Da f^j ein ganzzahliger Fluß ist, bilden die Kanten e mit $f_e^j \neq 0$ (d. h. $f_e^j = 1$) einen Pfad von der Wurzel zum 1-Blatt. Wir können deshalb die innere Summe ersetzen durch $\chi_{f^j}(x_i)$ und erhalten damit

$$x_i^* = \sum_{j \in J} \lambda_i \chi_{f^j}(x_i).$$

Da für $j \in J$ χ_{f^j} eine Ecke des BDD-Polytops ist, folgt unmittelbar $x^* \in P_{BDD}$.

„$\subseteq$“ Sei nun $x^* \in P_{BDD}$. Dann kann x^* als eine Konvexkombination von Ecken von P_{BDD} geschrieben werden:

$$x^* = \sum_{j \in J} \lambda_j x^j,$$

wobei für $j \in J$ gilt: $\lambda_j \geq 0$ und $\sum\limits_{j \in J} \lambda_j = 1$. Da die x_j ganzzahlig sind, gibt es für jedes x_j einen Pfad f^j mit $x^j = \chi_{f^j}$. Wir konstruieren daraus einen Fluß $f := \sum\limits_{j \in J} \lambda_j f^j$.

Dann gilt damit $f \in P_{flow}$ und

$$\begin{aligned} x_j^* &= \sum_{j \in J} \lambda_j x_i^j = \sum_{j \in J} \lambda_j \chi_{f^j}(x_i) \\ &= \sum_{j \in J} \lambda_j \sum_{\substack{e \in E \\ \text{head}(e) = x_i}} \text{par}(e) f_e^j = \sum_{\substack{e \in E \\ \text{head}(e) = x_i}} \text{par}(e) \cdot \left(\sum_{j \in J} \lambda_j f_e^j \right) \\ &= \sum_{\substack{e \in E \\ \text{head}(e) = x_i}} \text{par}(e) f^*(e) \end{aligned}$$

□

Damit haben wir alle grundlegenden Fakten beisammen, um im nächsten Abschnitt mit Hilfe von OBDDs Schnittebenen zu erzeugen.

6.3. Erzeugung von Schnittebenen

Wir gehen in diesem Abschnitt davon aus, daß ein Punkt x^* gegeben ist und daß wir zu diesem Punkt das BDD wie in Abschnitt 6.1 aufgebaut haben. Außerdem nehmen wir wieder an, daß das BDD 1-vollständig und das 0-Blatt entfernt worden ist.

Wir werden zwei Verfahren kennenlernen, mit denen wir entweder zeigen können, daß x^* im BDD-Polytop liegt, oder eine Schnittebene finden, die gültig ist für das BDD-Polytop, aber nicht für x^*.

6.3.1. Schnittebenen durch Lösung eines LP-Problems

Hier wird ein Verfahren beschrieben, das uns gestattet, durch Lösen eines einzigen LP-Problems eine Schnittebene zu berechnen oder zu zeigen, daß keine existiert.

Betrachte dazu das Polytop $P(f, x)$ aus dem vorigen Abschnitt. Es werde durch Ungleichungen $Ax + Bf \leq b$ beschrieben.

Als ersten Schritt wollen wir entscheiden, ob der Punkt x^* im BDD-Polytop enthalten ist. Dazu müssen wir prüfen, ob es ein f gibt mit

$$Af + Bx^* \leq b.$$

Die Idee ist nun, dieses Gleichungssystem so umzuformen, daß wir – falls $x^* \notin P_{BDD}$ ist – eine Schnittebene erhalten.

Wir definieren den **Projektionskegel** von $\text{Proj}_x(P(f, x))$ durch

$$W := \{v \in \mathbb{R}^k \mid v^T A = 0, v \geq 0\}$$

Nach [1, Theo. 1] läßt sich die Projektion dann auch schreiben als

$$\text{Proj}_x(P(f,x)) = \{x \in \mathbb{R}^n \mid (v^T B)x \leq v^T b, v \in W\}, \tag{6.5}$$

Mit Hilfe der Gleichung (6.5) können wir das Separierungsproblem folgendermaßen lösen und ggf. eine Schnittebene erzeugen:
Es ist

$$x^* \in \text{Proj}_x(P(f,x)) \quad \Leftrightarrow \quad (v^T B)x^* \leq v^T B \quad \text{für alle } v \in W.$$

Dies läßt sich umformen zu $v^T(Bx^* - b) \leq 0$ für alle $v \geq 0$ mit $v^T A = 0$ und führt zu folgendem linearen Programm:

$$\begin{aligned} \max\, & (Bx^* - b)^T v \\ & v^T A = 0 \\ & v \geq 0 \end{aligned} \tag{6.6}$$

Wir unterscheiden zwei Fälle:

1. $x^* \in \text{Proj}_x(P(f,x)) = P_{BDD}$. Das ist genau dann der Fall, wenn das Maximum in (6.6) kleiner oder gleich Null ist. Wir können und brauchen keine Schnittebene zu erzeugen.

2. $x^* \notin \text{Proj}_x(P(f,x)) = P_{BDD}$. Das Maximum ist genau dann positiv. Allerdings kann der Fall eintreten, daß das LP nicht beschränkt ist. Deshalb normalisieren wir v durch den linearen Constraint $\sum_{i=1}^{m} v_i \leq 1$.
   Sei $\hat{v}$ die Lösung von (6.6). Wenn $(Bx^* - b)\hat{v} > 0$ ist, dann ist

   $$(\hat{v}^T B)x \leq \hat{v}^T b \tag{6.7}$$

   eine Schnittebene, die x^* von P_{BDD} separiert.

Wir wollen hier noch einen interessanten Zusammenhang angeben:
Um festzustellen, ob ein gegebenes x^* in $P(f,x)$ enthalten ist, können wir folgendes (primales) Programm lösen:

$$\begin{aligned} \min\, & 0^T f \\ & Af \leq (b - Bx^*) \\ & f \in \mathbb{R}^{|E|} \end{aligned} \tag{6.8}$$

Es genügt hier $f \in \mathbb{R}^{|E|}$ zu fordern, da die Constraints für $0 \leq f \leq 1$ bereits in der Matrix A enhalten sind. Bilden wir dazu das duale Programm, dann erhalten wir:

$$\begin{aligned} \max\, & (Bx^* - b)^T v \\ & v^T A = 0 \\ & v \geq 0 \end{aligned} \tag{6.9}$$

Dies ist genau das Programm aus (6.6). Wir werden darauf gleich noch einmal zurückkommen.

Verkleinerung des LP

Betrachtet man die Beschreibung von $P(f,x)$ genauer, stellt man fest, daß es (und damit auch das Programm in (6.6)) viele redundante Constraints enthält. Indem wir diese beseitigen, können wir die Laufzeit deutlich verbessern und den Speicherbedarf verringern.

Die Beschreibung von $P(f,x)$ hat folgende Gestalt:

$$
\begin{array}{r||l}
 & Af \quad +Bx \quad \leq b \\
\hline\hline
\forall v \in V & \sum_{e\in\delta^-(v)} f_e - \sum_{e\in\delta^+(v)} f_e \qquad = \begin{cases} -1 & \text{falls } v = \textit{root} \\ +1 & \text{falls } v = \textit{leaf1} \\ 0 & \text{sonst} \end{cases} \\
\hline
\forall e \in E & -f_e \qquad\qquad \leq 0 \\
\hline
\forall e \in E & f_e \qquad\qquad \leq 1 \\
\hline
\forall i = 1,\ldots,n & \sum_{\substack{e\in E \\ \mathrm{head}(e)=x_i}} \mathrm{par}(e) f_e \quad -x_i \quad = 0
\end{array}
\tag{6.10}
$$

Da der ausgehende Fluß der Wurzel gleich Eins ist, wir nur nicht-negative Flüsse zulassen und in jedem von der Wurzel und dem 1-Blatt verschiedenen Knoten die Erhaltung des Flusses sicherstellen, können wir die Constraints $\forall e \in E : f_e \leq 1$ weglassen.

Die verbleibende Matrix A besteht aus drei Teilen:

- Der erste Teil ist die Inzidenzmatrix des BDDs. Wir nennen sie N.
- Der zweite Teil ist eine negative Einheitsmatrix der Dimension $|E| \times |E|$, die wir mit $-I_{|E|}$ bezeichnen.
- Der dritte Teil besteht aus den Levelgleichungen, die den Zusammenhang zwischen den x_i und den f_e herstellen. Diesen Teil bezeichnen wir mit L.

Die Matrix B besteht aus einer $(|V| + |E|) \times n$-dimensionalen 0-Matrix und einer negativen Einheitsmatrix $-I_n$ der Dimension $n \times n$. Der Vektor b hat nur zwei von Null verschiedene Einträge, die sich unter den ersten $|V|$ Einträgen befinden. Deshalb spalten wir b in drei Teile auf: $b^T = (b^T_{|V|}, 0^T_{|E|}, 0^T_n)$. Dasselbe machen wir mit dem Vektor v und erhalten $v^T = (v^T_{|V|}, v^T_{|E|}, v^T_n)$. Dies führt zu folgender äquivalenten

Formulierung von (6.6):

$$\begin{aligned} \max\ & b_{|V|}^T v_{|V|} - (x^*)^T v_n \\ & (N^T,\ -I_{|E|},\ L^T) \begin{pmatrix} v_{|V|} \\ v_{|E|} \\ v_n \end{pmatrix} = 0 \\ & v_{|V|} \in \mathbb{R}^{|V|} \\ & v_{|E|} \geq 0 \\ & v_n \in \mathbb{R}^n \end{aligned} \tag{6.11}$$

Die Variablen $v_{|V|}$ und v_n sind jetzt nicht mehr auf nicht-positive Werte beschränkt, da sie aus der Dualisierung von Gleichungen stammen (siehe dazu Abschnitt 2.4).

Wir können $-b_{|V|}^T v_{|V|}$ auch explizit als $v_{root} - v_{leaf1}$ schreiben, wobei v_{root} und v_{leaf1} diejenigen Komponenten von $v_{|V|}$ sind, die zur Wurzel bzw. zum 1-Blatt des BDDs gehören. Die anderen Komponenten von $b_{|V|}$ sind Null.

Die Variablen $v_{|E|}$ erscheinen nicht in der Zielfunktion. Sie dienen nur dazu, daß bei den Constraints in (6.11) die Gleichheit erfüllt ist. Solche Variablen nennt man **Slack-Variablen**. Streicht man die Variablen $v_{|E|}$ aus den Constraints, so muß man das Gleichheitszeichen durch ein Größer-Gleich-Zeichen ersetzen.

Damit erhalten wir die vereinfachte Version des Separierungs-LPs:

$$\begin{aligned} \max\ & v_{root} - v_{leaf1} - (x^*)^T v_n \\ & (N^T,\ L^T) \begin{pmatrix} v_{|V|} \\ v_n \end{pmatrix} \geq 0 \\ & v_{|V|} \in \mathbb{R}^{|V|} \\ & v_n \in \mathbb{R}^n \end{aligned} \tag{6.12}$$

Falls $x^* \notin P_{BDD}$ ist, dann kann auch dieses LP-Problem unbeschränkt sein. Wir garantieren in diesem Fall die Existenz des Maximums durch die Beschränkung von v durch $\|v\|_\infty \leq 1$.

Sei $\hat{v}$ wieder die Lösung des LPs. Wenn der Wert des Maximums positiv ist, dann ist

$$\hat{v}_n^T x \geq \hat{v}_{root} - \hat{v}_{leaf1}$$

eine Schnittebene.

Trotz beträchtlicher Verkleinerung ist das zu lösende LP-Problem in (6.12) noch relativ groß (d. h. sowohl die Zahl der Variablen als auch die Zahl der Constraints sind von der Größenordnung des BDDs). Das bewirkt eine verhältnismäßig hohe Laufzeit für die Schnittebenengenerierung. Deshalb haben wir ein Verfahren gesucht, das ohne das Lösen eines LP auskommt.

6.3.2. Schnittebenen durch Lagrange-Relaxierung

Um Schnittebenen ohne Lösen eines LP zu berechnen, greifen wir die Idee der Lagrange-Relaxierung aus Abschnitt 2.6 auf.

Betrachten wir dazu das Polytop $P(f, x)$ und die Art, wie wir es aus dem Fluß-Polytop P_{flow} durch Hinzufügen der Gleichungen (6.4) erhalten haben. Falls $x^* \in P_{BDD}$ ist, dann existiert eine Lösung des folgenden primalen Problems (siehe auch (6.8)):

$$\begin{aligned} &\min\ 0^T f \\ &\quad f \in P_{flow} \\ &\quad \forall i: \sum_{\substack{e \in E \\ \text{head}(E)=x_i}} \text{par}(e) \cdot f_e = x_i^* \end{aligned} \tag{6.13}$$

Wenn wir dieses Problem betrachten, stellen wir fest, daß es die Gleichungen aus (6.4) sind, die das Problem schwierig machen. Ohne sie ist die Constraint-Matrix als Inzidenzmatrix des BDD-Graphen total unimodular. Das bedeutet, daß für jede beliebige Zielfunktion die Lösung ganzzahlig und damit ein Pfad von der Wurzel zum 1-Blatt ist. Wir müssen dann denjenigen Pfad auswählen, der bezüglich der Zielfunktion die geringsten Kosten besitzt. Wählen wir geeignete Kantengewichte, so brauchen wir nur einen kürzesten Weg von der Wurzel zum 1-Blatt berechnen, um das Optimierungsproblem zu lösen. Dies ist in linearer Zeit in der Größe des BDD-Graphen möglich.

Davor müssen wir die schwierigen Constraints aus (6.4) loswerden. Dies gelingt uns mit der Lagrange-Relaxierung aus Abschnitt 2.6: Wir ziehen sie mit Vorfaktoren λ_i zur Zielfunktion hinzu und erhalten dann:

$$LR(\lambda) = \lambda^T x^* + \min_{f \in P_{flow}} \sum_{i=1}^{n} -\lambda_i \Big(\sum_{\substack{e \in E \\ \text{head}(e)=x_i}} \text{par}(e) f_e \Big) \tag{6.14}$$

Nach Lemma 2.3 gilt für alle $\lambda \geq 0$: $LR(\lambda) \leq 0$. Das Minimum in (6.14) berechnen wir – wie oben schon gesagt – mit Hilfe eines azyklischen kürzeste Wege-Algorithmus, indem wir die Kantengewichte folgendermaßen setzen:

$$\forall e \in E: w(e) = \begin{cases} -\lambda_i & \text{falls head}(e) = x_i \text{ und par}(e) = 1 \\ 0 & \text{sonst} \end{cases} \tag{6.15}$$

Da wir lediglich einen kürzesten Pfad von der Wurzel von der Wurzel zum 1-Blatt im 1-vollständigen BDD berechnen müssen, können wir wie in Abschnitt 6.1.3 auf

Seite 53 die Gewichtsfunktion verallgemeinern und den kürzesten Pfad berechnen, ohne daß wir das BDD vorher 1-vollständig gemacht haben müssen.

Nehmen wir an, daß $x^* \notin P_{BDD}$ ist. Um eine separierende Schnittebene zu berechnen, wenden wir das Subgradienten-Verfahren aus Algorithmus 2.1 an: Wir beginnen mit $\lambda^{(1)} = 1 \in \mathbb{R}^n$.
Sei $\lambda^{(k)}$ dasjenige λ, das nach k Iterationen gefunden wurde. Dann minimieren wir $LR(\lambda^{(k)})$. Sei $f^{(k)}$ der kürzeste Pfad dazu. Wir setzen

$$\xi_i^{(k)} := \sum_{\substack{e \in E \\ \text{head}(e) = x_i}} \text{par}(e) f_e^{(k)}$$

Für alle $i = 1, \ldots, n$ ist $\xi_i^{(k)} \in \{0, 1\}$, da $f^{(k)}$ ein Pfad ist. Außerdem stellt $\xi^{(k)} \in \{0, 1\}^n$ eine Ecke von P_{BDD} dar.

Falls $LR(\lambda^{(k)}) > 0$ ist, stoppen wir, wobei

$$\lambda^{(k)^T} x \leq \lambda^{(k)^T} \cdot \xi^{(k)} \tag{6.16}$$

eine Schnittebene ist, die x^* von P_{BDD} trennt. Andernfalls setzen wir

$$\lambda^{(k+1)} := \lambda^{(k)} + \frac{1}{k}\left(x^* - \xi^{(k)}\right) \tag{6.17}$$

Geometrisch kann dies interpretiert werden als die Rotation der Ebene $\lambda^{(k)^T} x \leq \lambda^{(k)^T} \xi^{(k)}$ entlang der Richtung des Vektors $x^* - \xi^{(k)}$.

Experimente haben gezeigt, daß die Schnittebenengenerierung mit Hilfe der Langrange-Relaxierung um Größenordnungen schneller ist als das Lösen des LP-Problems aus dem ersten Verfahren. In unserem Programm werden wir deshalb ausschließlich die Lagrange-Relaxierung verwenden.

Im nächsten Abschnitt behandeln wir zwei Spezialfälle, in denen eine Schnittebene bedeutend einfacher erzeugt werden kann.

6.4. Schnittebenen für Spezialfälle

Wir betrachten in diesem Abschnitt zwei Spezialfälle: zum einen den Fall, wenn das BDD nur aus dem 0-Blatt besteht, d. h. wenn es keine gültigen Pfade gibt. Zum anderen betrachten wir den Fall, wenn das Polytop nicht die höchstmögliche Dimension besitzt.

Wir gehen wieder davon aus, daß das BDD wie in Abschnitt 6.1 konstruiert wurde. Die Variablenmenge teilen wir wiederum in drei disjunkte Teile auf:

$$I = I_0 \,\dot\cup\, F \,\dot\cup\, I_1$$

wobei für alle $i \in I_0$ gilt $x_i^* = 0$, für alle $i \in I_1$ $x_i^* = 1$ und die übrigen Komponenten von x^* fraktionale Werte haben. Es gelte $\emptyset \neq F \neq \{1, \ldots, n\}$.

6.4.1. Exclusion Cut

Wenn das BDD leer ist, dann wissen wir, daß es keine gültige Lösung gibt, wenn wir die Variablen in I_0 auf 0 und die Variablen in I_1 auf 1 setzen. Deshalb können wir mit Hilfe einer Schnittebene all die Punkte $\hat{x}$ ausschließen, bei denen für $i \in I_0$ $\hat{x}_i = 0$ und für $i \in I_1$ $\hat{x}_i = 1$ ist. Zu denen gehört auch x^*.

Betrachte dazu die folgende lineare Funktion:

$$\sum_{i \in I_1} x_i + \sum_{i \in I_0} (1 - x_i)$$

Alle Punkte, die wir ausschließen wollen, maximieren diese Funktion, und der maximale Wert ist $|I_0| + |I_1|$. Um diese Punkte abzuschneiden, verringern wir den maximalen Wert um Eins. Dies führt zu folgender Schnittebene:

$$\sum_{i \in I_1} x_i + \sum_{i \in I_0} (1 - x_i) \leq |I_0| + |I_1| - 1$$

Dies ist äquivalent zu

$$\sum_{i \in I_1} x_i + \sum_{i \in I_0} -x_i \leq |I_1| - 1 \tag{6.18}$$

Für den Punkt x^* ist diese Schnittebene nicht gültig, jedoch nach Konstruktion für alle Punkte im BDD-Polytop.

6.4.2. Implication Cut

Wenn es Variablen gibt, deren Knoten im BDD alle entweder nur eine low-Kante oder nur eine high-Kante besitzen (d. h. die andere Kante wurde entfernt, weil sie direkt auf das 0-Blatt zeigte), dann besitzt das zugehörige Polytop nicht die volle Dimension. In einer solchen Situation impliziert die Fixierung bestimmter Variablen in I_0 und I_1 auf die entsprechenden Werte die Fixierung anderer Variablen. Dies können wir ausnutzen, um eine Schnittebene („Implication Cut“) zu erzeugen.

Wir müssen zwei Fälle unterscheiden, nämlich ob nur low- oder nur high-Kanten vorkommen:

- Es besitzen alle mit x_k beschrifteten Knoten nur eine ausgehende low-Kante. Das bedeutet: In jeder gültigen Belegung, bei der die Variablen in I_0 und I_1 entsprechend fixiert sind, muß x_k den Wert 0 erhalten. Dies läßt sich durch folgende logische Formel beschreiben:

$$\Big(\bigwedge_{i \in I_1} x_i \wedge \bigwedge_{i \in I_0} \overline{x}_i \Big) \Rightarrow \overline{x}_k$$

Elimination der Implikation liefert:

$$\Big(\bigvee_{i\in I_1} \overline{x}_i \vee \bigvee_{i\in I_0} x_i\Big) \vee \overline{x}_k$$

Diese Disjunktion läßt sich leicht in eine lineare Ungleichung umwandeln. Dadurch erhalten wir:

$$\sum_{i\in I_1}(1-x_i) + \sum_{i\in I_0} x_i + (1-x_k) \geq 1$$

Durch Vereinfachung erhalten wir den Implication Cut:

$$\sum_{i\in I_1} x_i + \sum_{i\in I_0} -x_i + x_k \leq |I_1| \tag{6.19}$$

- Alle mit x_k beschrifteten Knoten besitzen nur eine ausgehende high-Kante. In jeder gültigen Belegung, bei der die Variablen aus I_0 und I_1 entsprechend fixiert sind, muß also $x_k = 1$ gelten. Dadurch erhalten wir die logische Formel:

$$\Big(\bigwedge_{i\in I_1} x_i \wedge \bigwedge_{i\in I_0} \overline{x}_i\Big) \Rightarrow x_k$$

Dies ist äquivalent zu folgender Formel in disjunktiver Normalform:

$$\Big(\bigvee_{i\in I_1} \overline{x}_i \vee \bigvee_{i\in I_0} x_i\Big) \vee x_k$$

Diese können wir folgendermaßen als Ungleichung ausdrücken:

$$\sum_{i\in I_1}(1-x_i) + \sum_{i\in I_0} x_i + x_k \geq 1$$

Wenn wir diese vereinfachen, erhalten wir den Implication Cut für diesen Fall:

$$\sum_{i\in I_1} x_i + \sum_{i\in I_0} -x_i - x_k \leq |I_1| - 1 \tag{6.20}$$

Nach Konstruktion sind diese Implication Cuts gültig für alle Punkte in P_{BDD}, aber nicht für x^*.

6.5. Verstärkung der Schnittebenen

Nicht alle Schnittebenen, die gültig für ein Polytop sind und einen vorgegebenen Punkt außerhalb des Polytops abschneiden, sind für unsere Zwecke gleich gut geeignet. Im allgemeinen ist eine Schnittebene für uns umso besser, je mehr Punkte des LP-Polytops sie abschneidet. Somit sind die optimalen Schnittebenen diejenigen, die für alle Punkte einer Facette des ILP-Polytops mit Gleichheit erfüllt sind.

Wir werden in diesem Abschnitt zwei Verfahren kennenlernen, die uns gestatten, mit Hilfe des BDDs eine gegebene Schnittebene zu verbessern. Das erste Verfahren stellt sicher, daß mindestens eine Ecke des ILP-Polytops die Schnittebene mit Gleichheit erfüllt. Das zweite Verfahren versucht anschließend, die Schnittebene so zu verändern, daß sie eine Facette des ILP-Polytops oder zumindest ein Face hoher Dimension enthält.

Erste Verbesserung der Schnittebene Wir gehen davon aus, daß wir eine Schnittebene der Form $\pi^T x \leq \pi_0$ gegeben haben. Wir wollen sie verstärken, so daß sie in einem Punkt des ILP-Polytops mit Gleichheit erfüllt ist. Dazu vergrößern wir die Koeffizienten auf der linken Seite bzw. verkleinern die rechte Seite.

Wir können die Ungleichung umformen zu

$$\pi_i x_i \leq \pi_0 - \sum_{\substack{j=1 \\ j \neq i}}^{n} \pi_j x_j$$

Wenn wir $x_i = 1$ setzen, ist der maximale Wert für π_i gegeben durch

$$\pi_i = \pi_0 + \min_{\substack{x \in P_{BDD} \\ x_i = 1}} \sum_{\substack{j=1 \\ j \neq i}} -\pi_j x_j$$

Dieses Minimum kann mit Hilfe eines azyklische-kürzeste-Wege-Algorithmus auf dem BDD-Graphen berechnet werden, indem wir die Kantengewichte für $e \in E$ folgendermaßen setzen:

$$c(e) = \begin{cases} -\pi_j & \text{falls } \mathrm{par}(e) = 1, \mathrm{label}(\mathrm{head}(e)) = x_j \text{ und } j \neq i \\ -M & \text{falls } \mathrm{par}(e) = 1 \text{ und } \mathrm{label}(\mathrm{head}(e)) = x_i \\ 0 & \text{falls } \mathrm{par}(e) = 0. \end{cases}$$

M soll eine hinreichend große positive Zahl sein. Dadurch stellen wir sicher, daß auf dem kürzesten Pfad $x_i = 1$ gilt. Um den Wert von π_i zu erhalten, addieren wir zur Länge des kürzesten Pfades M und π_0.

Die Koeffizienten π_i der verstärkten Schnittebene hängen von der Reihenfolge ab, in der die Koeffizienten verstärkt werden. Wir können durch die Wahl verschiedener Reihenfolgen eine Menge unterschiedlicher Schnittebenen erzeugen.

Die rechte Seite der Ungleichung können wir mit einem entsprechenden Verfahren verbessern, d. h. π_0 verkleinern. Wir können π_0 folgendermaßen setzen:

$$\pi_0 = \max_{x \in P_{BDD}} \sum_{j=1}^{n} \pi_j x_j = - \min_{x \in P_{BDD}} \sum_{j=1}^{n} -\pi_j x_j$$

Wir setzen die Kantengewichte folgendermaßen:

$$c(e) = \begin{cases} -\pi_j & \text{falls par}(e) = 1 \text{ und label}(\text{head}(e)) = x_j \\ 0 & \text{falls par}(e) = 0 \end{cases}$$

Dann ergibt sich der Wert von π_0 als die Länge des kürzesten Wegs von der Wurzel des 1-vollständigen BDDs zum 1-Blatt.

Mit Hilfe dieser Methode erreichen wir, daß die Schnittebene mindestens eine Ecke des BDD-Polytops berührt. Die Laufzeit des Verfahrens ist $(n+1)\cdot|E|$, da wir $(n+1)$ Mal kürzeste Wege auf dem BDD-Graphen berechnen müssen. Da dieser azyklisch ist, ist die Laufzeit des kürzeste-Wege-Algorithmus linear in der Zahl der Kanten.

Zweite Verbesserung der Schnittebene In diesem Abschnitt werden wir ein Verfahren vorstellen, das die Schnittebene, die wir durch das erste Verfahren erhalten, weiter verstärkt, so daß sie eine Facette oder ein möglichst hochdimensionales Face des BDD-Polytops enthält.

Wir wollen uns dazu zuerst überlegen, wie wir erkennen können, ob eine gegebene Schnittebene $\pi^T x \leq \pi_0$ eine Facette enthält. Dazu nehmen wir an, daß das BDD-Polytop volldimensional ist. Außerdem gehen wir davon aus, daß π_0 – wie bereits oben beschrieben – verstärkt wurde, so daß die Schnittebene in mindestens einem ganzzahligen Punkt des Polytops mit Gleichheit erfüllt ist. Dann ist π_0 die Länge des kürzesten Pfads im 1-vollständigen BDD von der Wurzel zum 1-Blatt, wenn wir die Kantengewichte wie oben erläutert setzen. Sei

$$F = \{x \in P_{BDD} \mid \pi^T x = \pi_0\}$$

die Menge der kürzesten Pfade. Sei W der Vektorraum aller Normalenvektoren, für die alle Pfade in F dieselbe Länge habe, d. h.

$$W = \{c \in \mathbb{R}^n \mid \exists\, \beta \in \mathbb{R}\, \forall x \in F : c^T x = \beta\}.$$

Damit können wir zwei Kriterien formulieren, wann eine Schnittebene eine Facette enthält:

Lemma 6.2 *Es gilt* $\dim(W) = 1$*, d. h. es gibt ein* $w \in \mathbb{R}^n$ *mit* $W = \{\lambda \cdot w \mid \lambda \in \mathbb{R}\}$ *genau dann, wenn* $\pi^T x \leq \pi_0$ *bzw. die affine Hülle von* F *eine Facette enthält.*

Beweis:
Das Lemma folgt aus der Tatsache, daß W der Orthogonalraum von $\text{aff}(F)$ ist und daß damit $\dim(W) + \dim(\text{aff}(F)) = \dim(W) + \dim(F) = n$ gilt. □

Eine andere Möglichkeit zu testen, ob $\pi^T x \leq \pi_0$ eine Facette enthält, liefert uns das folgende Lemma:

Lemma 6.3 *F enthält genau dann n affin unabhängige Vektoren, wenn $\pi^T x \leq \pi_0$ bzw. die affine Hülle von F eine Facette enthält.*

Beweis:
Folgt aus [17, Kap. I.5, Prop. 6.6] mit $k = n$. □

Um die Menge F zu berechnen, müssen wir diejenigen Ecken des BDD-Polytops bestimmen, die die gegebene Schnittebene mit Gleichheit erfüllen. Dies kann durch einen durch einen Algorithmus geschehen, der die kürzesten Wege in einem azyklischen Graphen berechnet. Ein Problem ist, daß die Zahl der kürzesten Wege exponentiell sein kann, so daß es nicht immer praktikabel ist, die Wege aufzuzählen.

Ein weiteres Problem ist die Laufzeit der beiden Tests, ob die Schnittebene eine Facette enthält. Bei beiden Tests müssen wir große lineare Gleichungssysteme lösen. Dies kann nicht effizient genug geschehen.

Aus diesen Gründen verzichten wir darauf, in jedem Fall eine Schnittebene zu erzeugen, die eine Facette enthält, und verwenden stattdessen ein anderes Kriterium, um die Schnittebene zu verbessern. Wir werden die Mächtigkeit der Menge F vergrößern, was gleichzeitig die Dimension von W verringert. Mit etwas Glück verstärken wir dadurch die Schnittebene zu einer Facette. Dies kann allerdings für die beiden Methoden, die wir vorstellen, nicht garantiert werden.

Verbesserung der Koeffizienten in randomisierter Reihenfolge Sei eine für das ILP-Polytop gültige Schnittebene $\pi^T x \leq \pi_0$ und ein $k \in \{1, \ldots, n\}$ gegeben. Unser Ziel ist es, den Koeffizienten π_k so zu verstärken, daß sich die Mächtigkeit von F erhöht. Die Idee, die hinter unserer Verbesserung steckt, ist folgende: Oft laufen auf dem Level k alle kürzesten Wege von der Wurzel zum 1-Blatt über die 0-Kanten oder alle über die 1-Kanten. Wir passen dann die Kosten der 1-Kanten so an, daß sowohl 1- als auch 0-Kanten von den kürzesten Wegen verwenden werden können.

O. B. d. A. nehmen wir an, daß wir die Variablenordnung $x_1 < x_2 < \cdots < x_n$ verwenden, d. h. daß die Knoten, die mit der Variablen x_i beschriftet sind, auf dem i-ten Level liegen. Außerdem zählen wir die Kanten, die von Knoten auf dem i-ten Level ausgehen, zum i-ten Level hinzu.

Die Kantenkosten setzen wir wie in Formel (6.1), d. h. alle 0-Kanten bekommen Gewicht 0 und die 1-Kanten auf Level k das Gewicht π_k.

Für jeden Knoten v im 1-vollständigen BDD-Graphen auf dem Level k berechnen wir die Länge und Anzahl der kürzesten Wege von der Wurzel zu v. Entsprechend berechnen wir für jeden Knoten w auf dem $(k+1)$-ten Level die Länge und Anzahl der kürzesten Wege von w zum 1-Blatt.

Unter all den Kanten e auf Level k mit $\text{par}(e) = 0$ suchen wir diejenigen, für die $\text{dist}(\text{head}(e)) + \text{dist}(\text{tail}(e))$ minimal ist. Sei α der minimale Wert und p_α die Zahl der Pfade, die diese Kanten verwenden.

Dasselbe machen wir mit den Kanten e mit $\text{par}(e) = 1$. Auch dort suchen wir diejenigen Kanten e, bei denen $\text{dist}(\text{head}(e)) + \text{dist}(\text{tail}(e))$ minimal ist. Das Minimum bezeichnen wir mit β und die Zahl der Pfade, die diese Kanten verwenden, mit p_β.

Wir unterscheiden nun 3 Fälle:

- Wenn $\alpha = \beta + \pi_k$ ist, dann sind wir fertig. Wir können π_k nicht verstärken.
- Wenn $\alpha < \beta + \pi_k$ ist, dann gibt es p_α kürzeste Pfade im BDD. Sie benutzen lediglich die 0-Kanten auf dem Level k. Um die Zahl der kürzesten Wegen insgesamt zu vergrößern, müssen wir die Pfade, die über die 1-Kanten laufen, billiger machen.
- Wenn $\alpha > \beta + \pi_k$ ist, dann gibt es p_β kürzeste Pfade im BDD, welche die nur 1-Kanten auf dem Level k benutzen. Wir müssen die 1-Kanten teurer machen (die 0-Kanten haben immer Kosten 0), um die Zahl der kürzesten Wege zu vergrößern.

In beiden Fällen, in denen wir π_k verstärken können, setzen wir

$$\pi_k := \alpha - \beta.$$

Damit werden beide Arten von Kanten für kürzeste Wege von der Wurzel zum 1-Blatt verwendet. Wir passen die rechte Seite noch entsprechend an, indem wir $\pi_0 := \alpha$ setzen. Dadurch erhöht sich die Zahl der Ecken, welche die gegebene Schnittebene $\pi^T x \leq \pi_0$ mit Gleichheit erfüllen, auf $p_\alpha + p_\beta$.

Ein Problem, auf das man stößt, ist folgendes: Dadurch, daß wir den Koeffizenten π_k verkleinern, wenn $\alpha < \beta + \pi_k$ ist, kann es sein, daß die Ebene den vorgegebenen Punkt nicht mehr abschneidet. Wir prüfen deshalb nach jeder Verstärkung, ob der Punkt x^* noch abgeschnitten wird. Ist dies nicht der Fall, dann lassen wir einfach π_k unverändert.

Das Ergebnis der Verstärkung hängt von der Reihenfolge ab, in der die Level betrachtet werden. Je nach Wahl der Reihenfolge können wir verschiedene Schnittebenen generieren.

Wie groß ist die Laufzeit dieses Verfahrens? Um die kürzesten Wege zu berechnen, müssen wir, weil der BDD-Graph azyklisch ist, jede Kante einmal betrachten. Da wir in der Regel alle Koeffizienten verstärken wollen, müssen wir bei n Koeffizienten n mal die kürzesten Wege bestimmen. Somit ergibt sich eine Laufzeit von $n \cdot |E|$.

Levelweise Verbesserung der Koeffizienten Wenn wir nicht verschiedene Permutationen der Variablen betrachten, sondern sie immer in der Reihenfolge betrachten, die die Variablenordnung des BDD vorgibt, dann können wir die Laufzeit des Verfahrens verbessern.

Zuerst berechnen wir die Kosten und die Zahl der kürzesten Wege der Knoten auf den Leveln unterhalb der Wurzel zum 1-Blatt. Danach verbessern wir π_1 wie oben beschrieben. Nach jeder Berechnung eines π_k kann das Update der Kosten und der Zahl der kürzesten Wege von der Wurzel zu den Knoten auf Level $(k+1)$ aus den Informationen der Knoten auf Level k berechnet werden.

Insgesamt wird jede Kante dreimal betrachtet, was eine Laufzeit von $3 \cdot |E|$ zur Folge hat.

6.6. Lifting der Schnittebenen

Die Exclusion und Implication Cuts sind gültig für alle Punkte in P_{BDD}. Das bedeutet, sie können unmittelbar verwendet werden. Die übrigen Schnittebenen sind durch die Fixierung der Variablen in I_0 und I_1 nur gültig für ein Face von P_{BDD}. Vor ihrer Verwendung müssen sie für das gesamte BDD-Polytop gültig gemacht werden. Dieser Vorgang heißt **Lifting**.

Wir gehen wieder davon aus, daß die Variablen wie in Abschnitt 6.1 partitioniert sind in $I = I_0 \dot\cup I_1 \dot\cup F$ mit $\emptyset \neq F \neq \{1, \ldots, n\}$. Das Face, für das wir das BDD gebaut haben, ist dann gegeben durch

$$\mathcal{F} = P_{BDD} \cap \bigcap_{\delta \in \{0,1\}} \{x \in [0,1]^n \,|\, x_i = \delta \ \forall i \in I_\delta\}$$

Für $i \in I_0$ sei $\mathcal{F}_{i \in I_1}$ das Face, das mit $I_0 \setminus \{i\}$ und $I_1 \cup \{i\}$ konstruiert wurde, und $\mathcal{F}_{i \in F}$ das durch $I_0 \setminus \{i\}$ und I_1 definierte Face. Analog definieren wir für $i \in I_1$ die Faces $\mathcal{F}_{i \in I_0}$ und $\mathcal{F}_{i \in F}$.

Sei eine für $\mathcal{F}$ gültige Schnittebene gegeben durch

$$\sum_{j \in F} \pi_j x_j \leq \pi_0. \tag{6.21}$$

Lemma 6.4 *Für $i \in I_0$ ist die Schnittebene*

$$\alpha_i x_i + \sum_{j \in F} \pi_j x_j \leq \pi_0 \tag{6.22}$$

gültig für das Face $\mathcal{F}_{i \in F}$ mit $\alpha_i = \pi_0 - \zeta$, wobei ζ die Lösung des folgenden LP-Problems ist:

$$\begin{aligned} \max \ & \sum_{j \in F} \pi_j x_j \\ & x \in \mathcal{F}_{i \in I_1} \end{aligned}$$

Beweis:
Falls $\bar{x} \in \mathcal{F}$ ist, dann ist

$$\alpha_i \bar{x}_i + \sum_{j \in F} \pi_j \bar{x}_j = \sum_{j \in F} \pi_j \bar{x}_j \leq \pi_0,$$

weil (6.21) für das Face $\mathcal{F}$ gültig ist.
Falls $\bar{x} \in \mathcal{F}_{i \in I_1}$ ist, dann gilt

$$\alpha_i \bar{x}_i + \sum_{j \in F} \pi_j \bar{x}_j = \alpha_i + \sum_{j \in F} \pi_j \bar{x}_j \leq \alpha_i + \zeta = \pi_0$$

nach Konstruktion von α_i. □

Entsprechendes gilt für $i \in I_1$:

Lemma 6.5 *Für $i \in I_1$ ist die Schnittebene*

$$\beta_i x_i + \sum_{j \in F} \pi_j x_j \leq \pi_0 + \beta_i \tag{6.23}$$

gültig für das Face $\mathcal{F}_{i \in F}$ mit $\beta_i = \zeta - \pi_0$, wobei ζ die Lösung des folgenden LP-Problems ist:

$$\begin{aligned} \max \ & \sum_{j \in F} \pi_j x_j \\ & x \in \mathcal{F}_{i \in I_0} \end{aligned}$$

Beweis:
Analog zum Beweis von Lemma 6.4. □

Um eine Schnittebene zu erhalten, die für das ganze BDD-Polytop gültig ist, wenden wir die oben beschriebene Lifting-Methode nacheinander für alle Variablen aus $I_0 \cup I_1$ an. Die Koeffizienten α_i und β_i hängen von der Reihenfolge ab, in

der die Variablen geliftet werden. Dadurch können wir eine Reihe verschiedener Schnittebenen erzeugen, indem wir verschiedene Reihenfolgen verwenden.

Falls die gegebene Schnittebene in (6.21) ganzzahlige Koeffizienten besitzt, d. h. $\pi_0, \pi_j \in \mathbb{Z}$ für $j \in F$ ist, dann können wir die Lösungen der LPs ganzzahlig machen, d. h. ζ durch $\lfloor \zeta \rfloor$ ersetzen.

Die Größe der beim Lifting zu lösenden LP-Probleme ist viel kleiner als die Größe des LP-Problems, das in Abschnitt 6.3.1 zur Erzeugung einer Schnittebene gelöst werden mußte. Jenes besitzt $O(n)$ Variablen, dieses jedoch $O(|E|) = O(|V|)$. Zwischen n und $|V|$ kann ein exponentieller Unterschied bestehen.

Wir möchten möglichst starke Schnittebenen haben, d. h. am liebsten solche, die eine Facette von P_{BDD} enthalten. Selbst wenn die Schnittebene in (6.21) eine Facette von $\mathcal{F}$ definiert, braucht die geliftete Schnittebene diese Eigenschaft für P_{BDD} nicht länger besitzen. Um dies zu erreichen, müßten wir die LP-Probleme, die beim Liften auftreten, durch die entsprechenden ILP-Probleme ersetzen, indem wir deren Variablen auf die Werte $\{0, 1\}$ einschränken. Aufgrund der viel zu hohen Laufzeit ist dies jedoch nicht praktikabel.

6.7. Implementierung und experimentelle Ergebnisse

Die Implementierung der Schnittebenengenerierung wurde gemeinsam mit Markus Behle vom Max-Planck-Institut für Informatik in Saarbrücken vorgenommen. Meine Aufgabe war es, eine C++-Bibliothek zu erstellen, mit deren Hilfe die benötigten BDDs aufgebaut werden können. Folgende Punkte waren dabei zu beachten:

- Es soll eine effiziente Schnittstelle zum Datenaustausch mit dem Programm von Markus Behle geben. Durch den Datenaustausch sollen keine unnötigen Kopien der Daten entstehen.

- Implementierung der in Abschnitt 6.1.1 vorgestellten initialen Variablenordnungen.

- Constraints sollen nach Belieben aktiviert oder deaktiviert werden können. Ebenso sollen Variablen jederzeit auf feste Werte fixiert oder die Fixierung aufgehoben werden können.

- Für die aktivierten Constraints sollen BDDs aufgebaut werden. Dabei war darauf zu achten, daß diese BDDs nur einmal berechnet werden, auch wenn mehrere Schnittebenen erzeugt werden sollten oder andere Constraints aktiviert werden.

- Die BDDs sollen durch Löschen unnötiger Knoten mit Hilfe von Constraints, die der Benutzer vorgibt, verkleinert werden können. Dies erfolgt gemäß dem Verfahren, das in Abschnitt 6.1.3 beschrieben wurde.

- Die BDDs der aktivierten Constraints müssen effizient verknüpft werden. *Während* der Verknüpfung sollen die BDDs durch Reordering (das heißt z. B. Sifting) oder durch Löschen von Knoten verkleinert werden können. Die dazu verwendeten Constraints werden automatisch ausgewählt.

- Für die Verknüpfungsreihenfolge sollen verschiedene Heuristiken verwendet werden können (siehe dazu Abschnitt 4.1).

- Für all diese Operationen soll der BDD reduziert gehalten werden. Das bedeutet insbesondere, daß er nicht 1-vollständig gemacht werden darf, um die an diversen Stellen benötigten kürzesten Wege zu bestimmen.

Als Bibliothek für die OBDDs wurde CUDD von Fabio Somenzi [22] verwendet. Als Branch & Cut-Framework diente CPLEX der Firma ILOG. CPLEX gilt heutzutage als der mit Abstand schnellste ILP-Solver auf dem Markt.

Wir ersetzten die von CPLEX konstruierten Schnittebenen durch unsere eigenen, ließen jedoch das Branch & Cut-Framework ansonsten unverändert. Dadurch erreichten wir, daß wir die Schnittebenen von CPLEX bezüglich ihrer Stärke und der Zeit, die CPLEX für ihre Generierung benötigt, gut mit unseren Schnittebenen vergleichen können.

Die Quelltexte zur Bibliothek sind auf der beiliegenden CD zu finden. Als Übersicht kann das Klassendiagramm in Abbildung 6.3 dienen.

6.7.1. Experimentelle Ergebnisse

In diesem Abschnitt wollen wir einen Vergleich zwischen CPLEX uns unserem Solver anstellen. Wir verwenden dazu für unser Programm folgende Konfiguration:

- Wir erzeugen nur im Wurzelknoten des Branch & Bound-Baums maximal 50 Schnittebenen aus maximal 4 verschiedenen BDDs. Die genaue Zahl hängt davon ab, wie viele Schnittebenen CPLEX anfordert.

- Wir fixieren keine Variablen, wenn wir auch ohne Fixierungen das BDD aufgebaut bekommen (was bei den vorliegenden Benchmarks durchgehend der Fall war).

- Als initiale Variablenordnung verwenden wir `OCMVarOrder<NaturalOrder>` (entweder die BOG-Variante oder die WOG-Variante).

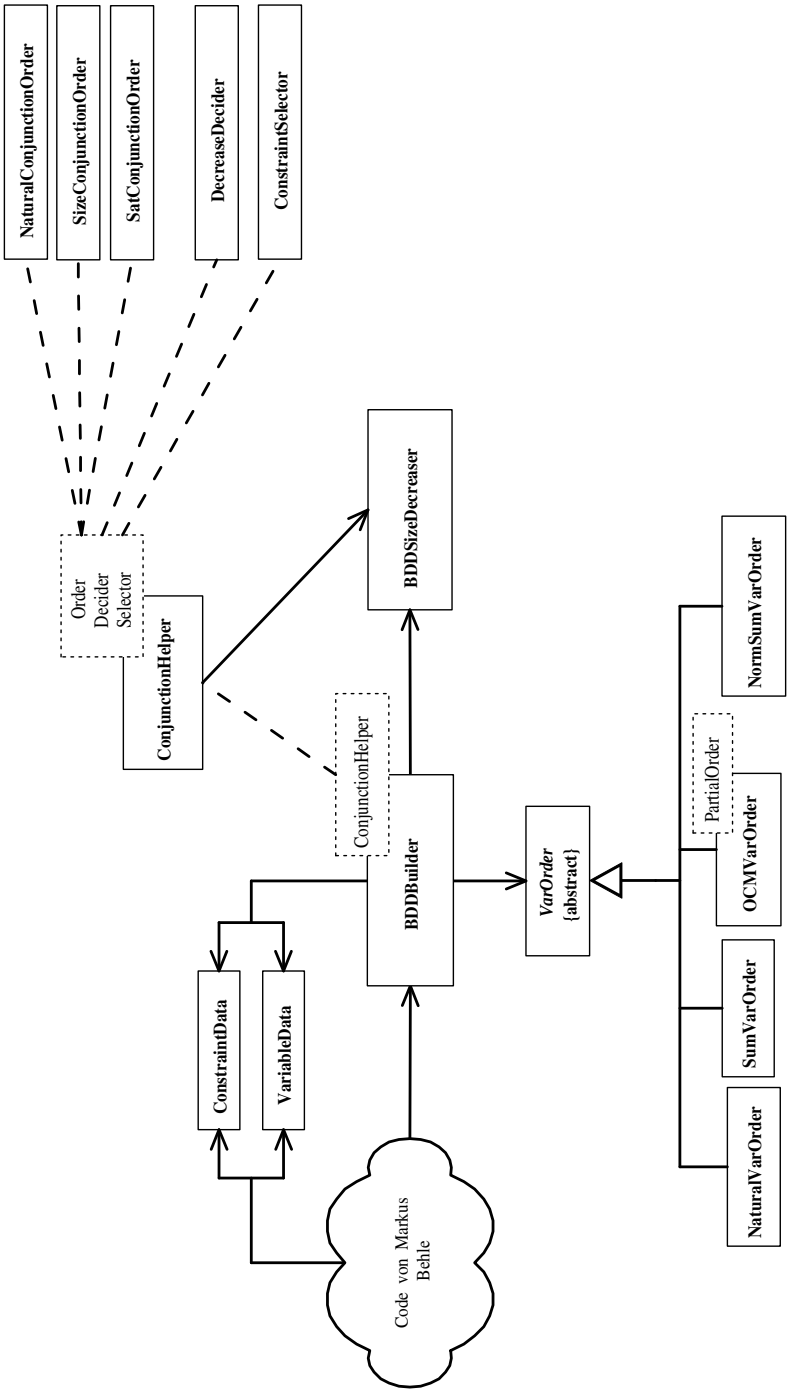


Abbildung 6.3.: Klassendiagramm der BDD-Bibliothek

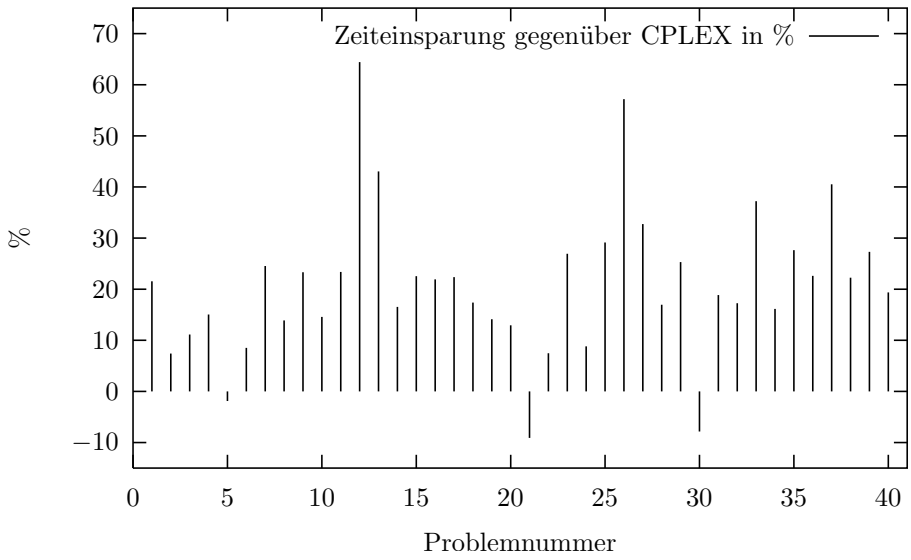


Abbildung 6.4.: Zeiteinsparung gegenüber CPLEX bei den hfo6-Benchmarks

CPLEX lassen wir für die Vergleichswerte mit den Standardeinstellungen laufen.

Als Benchmarks nehmen wir eine Menge von SAT-Problemen [5]. Um nicht nur Entscheidungsprobleme, sondern „echte“ Optimierungsprobleme zu erhalten, ergänzen wir als Zielfunktion

$$g(x_1, \ldots, x_n) = \sum_{i=1}^{n} x_i.$$

In den Abbildungen 6.4 bis 6.6 ist der prozentuale Anteil der Laufzeit von CPLEX, den wir mit unserem BDD-Ansatz einsparen, dargestellt, also der Wert

$$\frac{\text{Laufzeit von CPLEX} - \text{Laufzeit mit BDDs}}{\text{Laufzeit von CPLEX}} \cdot 100\%.$$

Weitere Informationen dazu sind in Tabelle A.1 im Anhang A zusammengestellt.

Insgesamt ergibt sich eine mittlere Beschleunigung von 16.06% gegenüber CPLEX. Die wenigen Probleme, bei denen wir langsamer als CPLEX sind, sind normalerweise relativ einfache Probleme, die CPLEX schnell gelöst bekommt. Dort ist der Overhead, der durch das Bauen der BDDs entsteht, zu groß, um mit CPLEX konkurrieren zu können.

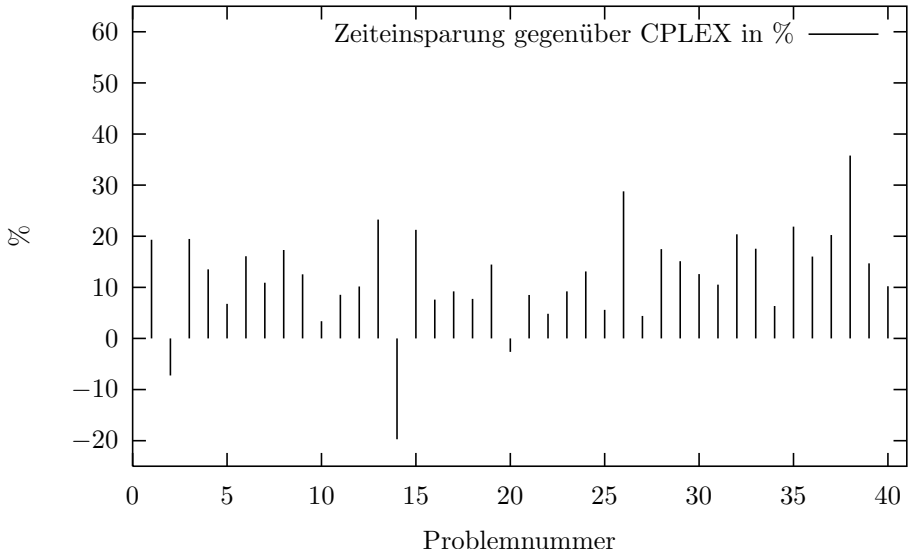


Abbildung 6.5.: Zeiteinsparung gegenüber CPLEX bei den hfo7-Benchmarks

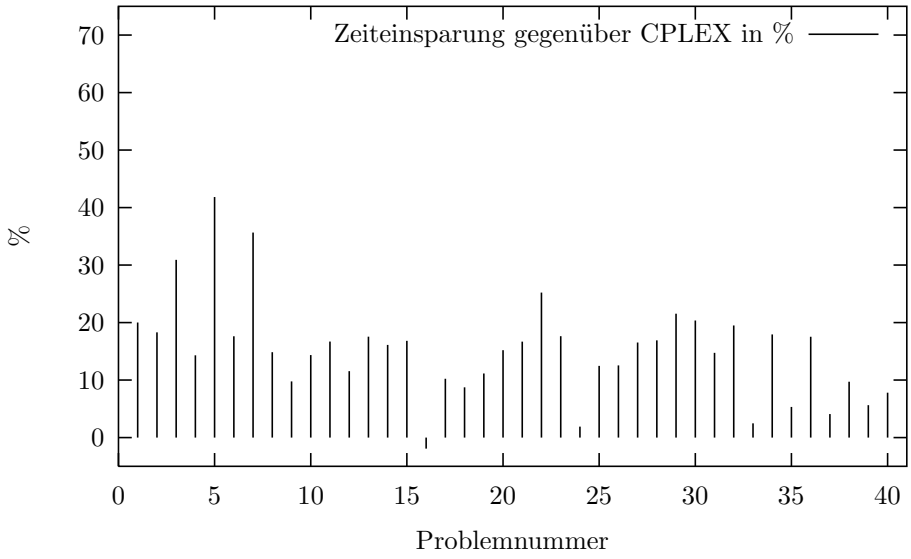


Abbildung 6.6.: Zeiteinsparung gegenüber CPLEX bei den hfo8-Benchmarks

7. Zusammenfassung und Ausblick

In diesem letzten Kapitel wird eine kurze Zusammenfassung von dem gegeben, was in den vorigen Kapiteln präsentiert wurde. Außerdem wird erörtert, was es über diese Arbeit hinaus noch zu tun gibt.

7.1. Zusammenfassung

Das Ziel dieser Arbeit war es zu zeigen, wie binäre Entscheidungsdiagramme in der ganzzahligen linearen Programmierung eingesetzt werden können, um die klassischen Lösungsverfahren effizienter zu machen.

Der erste Teil dieser Arbeit stellte die Grundlagen vor: zum einen ist das die notwendige Theorie der ganzzahligen linearen Optimierung und zum anderen die BDD-Technologie. Außerdem wurden kurz die klassischen ILP-Verfahren Branch & Bound sowie die Cutting-Plane-Verfahren vorgestellt.

Der erste Schritt unseres Verfahrens zur Schnittebenenerzeugung mit Hilfe von BDDs war der Aufbau des BDDs. Dabei mußten wir dafür sorgen, daß das BDD möglichst klein bleibt. Dies erreichten wir durch eine gute initiale Variablenordnung und ein Verfahren, das im BDD nicht verwendete Constraints dazu benutzt, Knoten zu löschen, die für uns nicht notwendig sind. Außerdem konnten wir bestimmte Variablen fixieren. Allerdings mußte dann die Schnittebene am Ende nochmals geliftet werden, bevor sie eingesetzt werden konnte.

Nach dem BDD-Aufbau stellten wir zwei Verfahren vor, wie aus dem BDD eine Schnittebene berechnet werden kann. Anschließend wurden zwei Techniken zur Verstärkung der erzeugten Schnittebene vorgestellt.

Experimente haben gezeigt, daß unser Ansatz zwar bei großen Problemen nicht mit den herkömmlichen ILP-Solvern mithalten kann, weil dort die BDDs zu groß werden. Aber bei kleinen, schwierigen Problemen, mit denen die heutigen Solver Probleme haben (wie z. B. bei den von uns verwendeten Benchmarks) kann unser Ansatz selbst mit den schnellsten verfügbaren Solvern ohne weiteres mithalten.

7.2. Ausblick

Was bleibt noch zu klären? Da ist einerseits die Frage, wie man das bisherige Verfahren verbessern kann. Das heißt beispielsweise, ob es ein rein kombinatorisches

Verfahren gibt, mit dessen Hilfe man effizient aus dem BDD eine Schnittebene berechnen kann.

Unser bisheriges Verfahren, mit Hilfe der Lagrange-Relaxierung eine Schnittebene zu bestimmen ist nicht kombinatorisch. Kombinatorische Verfahren zeichnen sich dadurch aus, daß die Lösung direkt „ausgerechnet“ wird. Beim Subgradienten-Verfahren zur Berechnung der Lagrange-Relaxierung müssen wir solange iterieren, bis wir einen positiven Wert erreicht haben oder wissen, daß der Grenzwert kleiner oder gleich Null ist.

Kombinatorische Verfahren sind beispielsweise Algorithmen zur Berechnung kürzester Wege, maximaler Flüsse, minimaler Spannbäume ...

Ein anderer Punkt ist, ob man das Verfahren auf allgemeinere Probleme wie z. B. die gemischt-ganzzahligen linearen Programme anwenden kann, bei denen nur ein Teil der Variablen ganzzahlig ist, die übrigen aber reell sind:

Definition 7.1 *Ein **gemischt-ganzzahliges Programm** (MILP[1]) ist gegeben durch Matrizen $A \in \mathbb{Z}^{m \times n}$, $B \in \mathbb{Z}^{m \times p}$ und Vektoren $b \in \mathbb{Z}^m$ und $c \in \mathbb{Z}^{n+p}$. Gesucht ist sind $x \in \mathbb{Z}^n$ und $y \in \mathbb{R}^p$, so daß der Wert von $g(x, y) = c^T \cdot \begin{pmatrix} x \\ y \end{pmatrix}$ minimal ist unter der Bedingung, daß*

$$Ax + By \leq b$$

gilt.

Dabei ist nicht klar, wie man die reellen Variablen behandeln soll, da naturgemäß in BDDs nur ganzzahlige Variablen vorkommen können. Vorstellbar ist, daß man zuerst nur die ganzzahligen Variablen berücksichtigt und durch eine Technik wie das Lifting (siehe Abschnitt 6.6) die erzeugte Schnittebene für die reellen Variablen gültig macht. An dieser Stelle besteht auf jeden Fall noch Forschungsbedarf.

Da unser größtes Problem ist, daß die verwendeten BDDs sehr groß werden, könnte es sich lohnen, andere Datenstrukturen zur Darstellung von Booleschen Funktionen zu untersuchen, die zwar nicht mehr kanonisch, aber dafür vielleicht kompakter sind. Ein Beispiel dafür sind die And-Inverter-Graphen. Da die Synthese-Operationen, die wir zum Aufbau des BDDs verwendet haben, auch für die And-Inverter-Graphen existieren, sollte sich das Verfahren aus Kapitel 4 relativ einfach auf And-Inverter-Graphen übertragen lassen.

[1] MILP: Mixed integer linear program

A. Laufzeitmessungen für die hfo-Benchmarks

Die Tabelle enthält folgende Informationen:

1. *Problem*: Name der Instanz
2. *#vars*: Anzahl der Variablen, von denen das Problem abhängt.
3. *sat*: Ist das Problem erfüllbar (d. h. ist das ILP-Polytop verschieden von $\emptyset$)?
4. *CPLEX*: Laufzeit von CPLEX in Sekunden, wenn die Standard-Einstellungen verwendet werden.
5. *BDD-Ansatz*: Laufzeit, wenn wir – wie oben beschrieben – die CPLEX-Schnittebenen durch unsere eigenen ersetzen.
6. %: Prozentuale Laufzeiteinsparung, wenn unser Ansatz verwendet wird.

Problem	#vars	sat	CPLEX	BDD-Ansatz	%
hfo6-001	40	nein	948.57	744.24	21.54
hfo6-002	40	nein	952.16	881.63	7.41
hfo6-003	40	nein	1012.95	900.26	11.12
hfo6-004	40	nein	884.01	751.00	15.05
hfo6-005	40	ja	661.70	674.15	−1.88
hfo6-006	40	ja	546.30	499.80	8.51
hfo6-007	40	nein	1028.08	775.87	24.53
hfo6-008	40	ja	657.88	566.55	13.88
hfo6-009	40	ja	227.68	174.60	23.31
hfo6-010	40	nein	932.56	796.64	14.57
hfo6-011	40	ja	461.10	353.28	23.38
hfo6-012	40	ja	136.19	48.44	64.43
hfo6-013	40	ja	490.63	279.47	43.04
hfo6-014	40	nein	799.14	667.04	16.53
hfo6-015	40	ja	959.86	743.51	22.54
hfo6-016	40	nein	961.23	750.58	21.91
hfo6-017	40	nein	937.19	727.63	22.36

Problem	#vars	sat	CPLEX	BDD-Ansatz	%
hfo6-018	40	ja	793.64	655.70	17.38
hfo6-019	40	nein	922.08	791.86	14.12
hfo6-020	40	ja	437.75	381.14	12.93
hfo6-021	40	ja	224.65	245.10	−9.10
hfo6-022	40	ja	788.46	729.53	7.47
hfo6-023	40	nein	972.10	710.34	26.93
hfo6-024	40	ja	230.01	209.74	8.81
hfo6-025	40	nein	1082.80	767.36	29.13
hfo6-026	40	ja	470.63	201.51	57.18
hfo6-027	40	ja	264.51	177.92	32.74
hfo6-028	40	ja	775.67	644.12	16.96
hfo6-029	40	nein	943.68	704.88	25.31
hfo6-030	40	ja	229.77	247.78	−7.84
hfo6-031	40	nein	1005.79	816.13	18.86
hfo6-032	40	nein	880.84	728.91	17.25
hfo6-033	40	ja	768.38	482.38	37.22
hfo6-034	40	nein	929.11	779.09	16.15
hfo6-035	40	nein	1047.60	758.02	27.64
hfo6-036	40	ja	935.37	723.91	22.61
hfo6-037	40	ja	528.59	314.36	40.53
hfo6-038	40	nein	940.00	730.90	22.24
hfo6-039	40	nein	1158.33	842.08	27.30
hfo6-040	40	nein	998.65	805.19	19.37
hfo7-001	32	ja	378.14	305.13	19.31
hfo7-002	32	ja	56.95	61.08	−7.25
hfo7-003	32	nein	653.65	526.46	19.46
hfo7-004	32	nein	698.22	603.84	13.52
hfo7-005	32	ja	379.38	353.69	6.77
hfo7-006	32	ja	282.30	236.87	16.09
hfo7-007	32	ja	650.28	579.42	10.90
hfo7-008	32	ja	197.52	163.34	17.30
hfo7-009	32	ja	365.07	319.29	12.54
hfo7-010	32	nein	583.75	564.09	3.37
hfo7-011	32	nein	609.43	557.39	8.54
hfo7-012	32	nein	600.36	539.38	10.16
hfo7-013	32	nein	666.47	511.47	23.26
hfo7-014	32	ja	27.27	32.65	−19.73
hfo7-015	32	nein	695.91	548.05	21.25
hfo7-016	32	ja	324.20	299.56	7.60

Problem	#vars	sat	CPLEX	BDD-Ansatz	%
hfo7-017	32	nein	618.49	561.55	9.21
hfo7-018	32	nein	618.18	570.33	7.74
hfo7-019	32	nein	711.85	608.96	14.45
hfo7-020	32	ja	93.69	96.15	−2.63
hfo7-021	32	ja	414.35	379.09	8.51
hfo7-022	32	ja	408.06	388.36	4.83
hfo7-023	32	ja	255.92	232.34	9.21
hfo7-024	32	ja	166.70	144.84	13.11
hfo7-025	32	ja	233.67	220.58	5.60
hfo7-026	32	ja	752.07	535.57	28.79
hfo7-027	32	ja	346.35	331.07	4.41
hfo7-028	32	nein	666.28	549.79	17.48
hfo7-029	32	ja	441.78	374.98	15.12
hfo7-030	32	nein	656.08	573.39	12.60
hfo7-031	32	ja	597.74	534.84	10.52
hfo7-032	32	nein	671.32	534.58	20.37
hfo7-033	32	ja	554.89	457.44	17.56
hfo7-034	32	nein	599.33	561.36	6.34
hfo7-035	32	nein	688.98	538.24	21.88
hfo7-036	32	nein	654.13	549.33	16.02
hfo7-037	32	nein	677.95	540.76	20.24
hfo7-038	32	nein	869.92	558.55	35.79
hfo7-039	32	nein	675.13	575.96	14.69
hfo7-040	32	nein	637.50	572.33	10.22
hfo8-001	27	nein	730.21	584.12	20.01
hfo8-002	27	nein	721.40	589.40	18.30
hfo8-003	27	ja	183.53	126.80	30.91
hfo8-004	27	nein	719.29	616.45	14.30
hfo8-005	27	ja	48.59	28.26	41.84
hfo8-006	27	ja	414.48	341.50	17.61
hfo8-007	27	ja	21.45	13.80	35.66
hfo8-008	27	ja	658.62	560.85	14.84
hfo8-009	27	ja	102.24	92.26	9.76
hfo8-010	27	nein	683.51	585.46	14.35
hfo8-011	27	ja	275.38	229.39	16.70
hfo8-012	27	nein	638.72	564.97	11.55
hfo8-013	27	ja	340.98	281.13	17.55
hfo8-014	27	ja	361.95	303.60	16.12
hfo8-015	27	ja	164.87	137.14	16.82

Problem	#vars	sat	CPLEX	BDD-Ansatz	%
hfo8-016	27	ja	124.69	127.08	−1.92
hfo8-017	27	ja	404.63	363.27	10.22
hfo8-018	27	ja	466.75	425.95	8.74
hfo8-019	27	nein	643.27	571.63	11.14
hfo8-020	27	nein	680.90	577.39	15.20
hfo8-021	27	nein	658.74	548.86	16.68
hfo8-022	27	ja	715.93	535.39	25.22
hfo8-023	27	ja	614.98	506.53	17.63
hfo8-024	27	ja	356.01	349.21	1.91
hfo8-025	27	nein	699.04	611.86	12.47
hfo8-026	27	nein	678.12	593.00	12.55
hfo8-027	27	nein	727.62	607.33	16.53
hfo8-028	27	ja	606.09	503.63	16.91
hfo8-029	27	ja	622.00	488.05	21.54
hfo8-030	27	nein	751.83	598.78	20.36
hfo8-031	27	nein	697.75	594.99	14.73
hfo8-032	27	nein	729.35	587.14	19.50
hfo8-033	27	ja	299.80	292.37	2.48
hfo8-034	27	nein	715.67	587.31	17.94
hfo8-035	27	ja	263.15	249.12	5.33
hfo8-036	27	nein	722.24	595.65	17.53
hfo8-037	27	nein	661.19	634.15	4.09
hfo8-038	27	nein	679.57	613.58	9.71
hfo8-039	27	nein	620.06	585.17	5.63
hfo8-040	27	nein	649.23	598.58	7.80

Tabelle A.1.: Laufzeiten für diverse Benchmarks

Literaturverzeichnis

[1] Egon Balas. Projection and lifting in combinatorial optimization. In Michael Jünger and Denis Naddef, editors, *Computational combinatorial optimization*, volume 2241 of *Lecture Notes in Computer Science*, pages 26–56, Berlin, 2001. Springer-Verlag. Optimal or provably near-optimal solutions, Papers from the Spring School held in Schloß Dagstuhl, May 15–19, 2000.

[2] Bernd Becker and Rolf Drechsler. *Graphenbasierte Funktionsdarstellung.* B. G. Teubner, Stuttgart, 1998.

[3] R. E. Bixby, S. Ceria, C. M. McZeal, and M. W. P. Svalsbergh. *An updated mixed integer programming library: MIPLIB 3.0.* Submitted to SIAM News, März 1996.

[4] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. In *IEEE Trans. on Comp. 35(8):677–691*, 1986.

[5] M. Buro and H. Kleine Büning. Report on a SAT competition. *Bulletin of the European Association for Theoretical Computer Science*, 49:143–151, 1993. Technical Contributions.

[6] Hiroshige Fujii, Goichi Ootomo, and Chikahiro Hori. Interleaving based variable ordering methods for ordered binary decision diagrams. In *Int'l Conf. on CAD*, pages 38 – 41, 1993.

[7] R. E. Gomory. Solving linear programming problems in integers. In R. Bellman and M. Hall, editors, *Combinatorial Analysis, Proceedings of Symposia in Applied Mathematics X*, pages 211 – 215. American Mathematical Society, 1960.

[8] Marc Herbstritt, Thomas Kmieciak, and Bernd Becker. Circuit partitioning for SAT-based combinatorial circuit verification – a case study. Technical Report 206, July 2004.

[9] Bernhard Korte and Jens Vygen. *Combinatorial Optimization.* Springer Verlag, 2nd edition, 2002.

[10] Y.-T. Lai, M. Pedram, and S.B.K. Vrudhula. EVBDD-based algorithms for integer linear programming, spectral transformation, and functional decomposition. In *IEEE Trans. on CAD 13(8):959–975*, 1994.

[11] Claude Lemaréchal. Lagrangian relaxation. In M. Jünger and D. Naddef, editors, *Computational Combinatorial Optimization*, Lecture Notes in Computer Science LNCS 2241. Springer Verlag, 2001.

[12] Alexander Martin. General mixed integer programming: Computational issues for branch-and-cut algorithms. In *Computational Combinatorial Optimization*, Lecture Notes in Computer Science LNCS 2241. Springer Verlag, 2001.

[13] Kurt Mehlhorn. *Data Structures and Algorithms 2 (Graph Algorithms and NP-Completeness)*. Springer Verlag, 1984.

[14] John E. Mitchell. Branch-and-cut algorithms for integer programming. In *Encyclopedia of Optimization*. Kluwer Academic Press, 2001.

[15] John E. Mitchell. Cutting plane algorithms for integer programming, 2001.

[16] John E. Mitchell and Eva K. Lee. Branch & bound methods for integer programming. In *Encyclopedia of Optimization*. Kluwer Academic Press, 2001.

[17] George L. Nemhauser and Laurence A. Wolsey. *Integer and combinatorial optimization*. Wiley-Interscience Series in Discrete Mathematics and Optimization. John Wiley & Sons Inc., New York, 1988. A Wiley-Interscience Publication.

[18] P. T. Polyak. A general method for solving extremal problems. In *Soviet Mathematics Doklady*, volume 8, pages 593–597, 1967.

[19] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Int'l Conf. on CAD*, pages 42–47, 1993.

[20] Alexander Schrijver. *Theory of linear and integer programming*. Wiley-Interscience Series in Discrete Mathematics. John Wiley & Sons Ltd., Chichester, 1986. A Wiley-Interscience Publication.

[21] Alexander Schrijver. *A Course in Combinatorial Optimization*, 2003.

[22] Fabio Somenzi. *CUDD: CU Decision Diagram Package. Release 2.4.0*. University of Colorado at Boulder, 2004.

[23] Ingo Wegener. *Branching programs and binary decision diagrams*. SIAM Monographs on Discrete Mathematics and Applications. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 2000. Theory and applications.