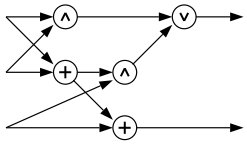
Diplomarbeit

Transformationen zwischen verschiedenen BDD-Modellen

Thomas Jansen



Diplomarbeit
am Fachbereich Informatik
der Universität Dortmund

März 1996

Betreuer:

Prof. Dr. Ingo Wegener

Inhaltsverzeichnis

1 Einleitung **1**

2 BDD-Modelle: Definitionen, Eigenschaften, Algorithmen **3**

2.1 Free Binary Decision Diagrams (FBDDs) und Ordered Binary Decision Diagrams (OBDDs) 5

2.2 Graphgesteuerte BDDs 15

2.3 Zero-suppressed Binary Decision Diagrams (ZBDDs) 22

2.4 Ordered Functional Decision Diagrams (OFDDs) 27

2.5 Zusammenfassung 34

3 Transformationsalgorithmen **35**

3.1 Die Transformationsalgorithmen von Savický und Wegener 35

3.1.1 Transformation eines FBDD in ein πOBDD 38

3.1.2 Transformation eines π'OBDD in ein πOBDD 39

3.2 Transformationen mit ZBDDs 41

3.2.1 Transformation eines πOBDD in ein πZBDD 45

3.2.2 Transformation eines πZBDD in ein πOBDD 55

3.2.3 Transformation eines π'ZBDD in ein πZBDD 56

3.3 Transformationen mit graphgesteuerten BDDs 63

3.3.1 Transformation eines FBDD in ein graphgesteuertes BDD 63

3.3.2 Transformationen zwischen LBDDs und WBDDs 64

3.3.3 Transformation eines graphgesteuerten BDD in ein πOBDD 65

3.3.4 Transformation eines πOBDD in ein WBDD 68

3.3.5 Transformation eines πOBDD in ein LBDD 71

3.4 Transformationen mit OFDDs 72

3.4.1 Transformation eines πOBDD in ein πOFDD 74

3.4.2 Transformation eines πOFDD in ein πOBDD 77

3.4.3 Empirische Aussagen über die Transformationen zwischen πOBDDs und πOFDDs 78

3.4.4 Weiterführende Gedanken zur Transformation eines πOBDD in ein πOFDD 94

3.4.5 Transformation eines π'OFDD in ein πOFDD 97

4 Abschließende Bemerkungen **100**
4.1 Transformationen mit OFDDs 100
4.2 Transformationen mit ZBDDs 100
4.3 Transformationen zwischen graphgesteuerten BDDs 102
4.4 Tabellarische Gesamtübersicht 102

A Integration von OFDDs in das BDD-Paket von Long **103**

B Bemerkungen zur Implementierung von Transformationsalgorithmen **108**

Literatur **111**

Tabellenverzeichnis **113**

Abbildungsverzeichnis **113**

1 Einleitung

Für den Herbst 1992 kündigt die Firma Intel die Verfügbarkeit eines neuen Prozessors für Personal Computer an, der den Namen Pentium tragen soll[1]. Im Frühjahr 1993 kommt es dann tatsächlich zur Auslieferung. Erst im Juni 1994 fällt einem Intel-Mitarbeiter bei einer Kontrolle ein Fehler in der Fließkommarecheneinheit des Prozessors auf: Divisionsbefehle mit bestimmten Operatoren sind betroffen, sie liefern Ergebnisse mit zu großer Ungenauigkeit[2]. Intel beginnt daraufhin, die Produktion des Prozessors umzustellen, um künftig bereinigte Prozessoren verkaufen zu können; die Öffentlichkeit wird aber nicht informiert. Erst im Oktober 1994 wird der Rechenfehler von einem Benutzer entdeckt, der die Nachricht im Internet verbreitet; im November 1994 berichten schließlich die Massenmedien. Daraufhin entschließt sich die Firma Intel am 20. Dezember 1994, allen Kunden einen Umtausch des fehlerhaften Prozessors anzubieten. Bis dahin sind beinahe sechs Millionen Pentium-Prozessoren verkauft worden. Man schätzt, daß sich die Kosten der Umtauschaktion für Intel auf etwa 300 Millionen Dollar belaufen.

Offensichtlich ist es wünschenswert, im Besitz von Werkzeugen zu sein, die eine frühe Erkennung solcher Fehler möglich machen. Es gibt auch tatsächlich eine ganze Reihe von Datenstrukturen und Algorithmen, die eine Hardwareverifikation zum Ziel haben. Weil das Problem der Verifikation von Schaltkreisen sich leicht auf das NP-vollständige Erfüllbarkeitsproblem reduzieren läßt, kann man allerdings von keinem Werkzeug zuverlässige, schnelle Verifikation für alle Schaltkreise erwarten. Die im Rahmen dieser Arbeit angesprochenen Datenstrukturen und Methoden wären durchaus geeignet gewesen, die frühzeitige Entdeckung des Pentium-Fehlers zu ermöglichen.

Boolesche Funktionen $f : \{0,1\}^n \to \{0,1\}^m$ stellen in einer Vielzahl von praktischen Anwendungen den Dreh- und Angelpunkt dar; das gilt nicht nur für die Hardwareverifikation sondern beispielsweise auch für CAD, Testmustergenerierung und die Analyse endlicher Automaten. Datenstrukturen und Algorithmen auf diesen Datenstrukturen, welche den Umgang mit Booleschen Funktionen praktikabel machen, sind offenbar von großer praktischer Bedeutung. Die vielleicht naheliegenste Darstellungsweise sind Schaltkreise; allerdings sind eine ganze Reihe wichtiger Operationen auf Schaltkreisen nicht effizient durchführbar. Die NP-Vollständigkeit der zugehörigen Entscheidungsprobleme macht es überaus wahrscheinlich, daß dieser Zustand von Dauer ist. Eine verbreitete Datenstruktur für Boolesche Funktionen sind OBDDs, die von Bryant (1986) eingeführt wurden. Alle praktisch relevanten Operationen lassen sich auf OBDDs überaus effizient ausführen; bedauerlicherweise haben viele Funktionen aber nur OBDDs exponentieller Größe. Das muß auch so sein: Es gibt offenbar genau 2^{2^n} Boolesche Funktionen $f : \{0,1\}^n \to \{0,1\}$; eine Darstellungsform Boolescher Funktionen, die mit polynomieller Länge $p(n)$ für ein Polynom p auskommt, kann aber nur $2^{p(n)}$ verschiedene Boolesche Funktionen darstellen. Also

[1] Die Chronologie der Ereignisse geht auf Hüskes (1995) und Schnurer (1993) zurück.

[2] Bei Divisionen mit langen Fließkommazahlen soll gemäß der Spezifikation der relative Fehler in einer Größeordnung von 10^{-16} bis 10^{-9} liegen; kommen in den Faktoren aber bestimmte Bitmuster vor, so liegt der relative Fehler tatsächlich in einer Großenordnung von 10^{-5}. Eine ausführliche Betrachtung zum Fehlerausmaß und zur Fehlerwahrscheinlichkeit findet man bei Pratt (1995).

gibt es immer viele Funktionen, die nur exponentiell große Darstellungen haben, wenn alle Booleschen Funktionen in der gewählten Darstellung überhaupt repräsentiert werden können. Die einzig vernünftige Hoffnung kann höchstens sein, möglichst viele in der Praxis wichtige Funktionen auf polynomiellem Platz darstellen zu können. Es gibt aber leider auch eine Reihe praktisch höchst relevanter Funktionen (etwa bei der Multiplikation von n-Bitzahlen), die nur exponentiell große OBDD-Darstellungen haben. Um diesem Problem zu begegnen, wurden eine Reihe alternativer Datenstrukturen vorgeschlagen, die alle eine gewisse Familienähnlichkeit zu OBDDs aufweisen und verschiedene Vorzüge haben. Keine bekannte Datenstruktur kann jedoch alle in der Praxis vorkommenden Probleme und Aufgabenstellungen befriedigend bewältigen, so daß es nahe liegt, in praktischen Anwendungen mehrere verschiedene Datenstrukturen zu verwenden. Durch den Umgang mit verschiedenen BDD-artigen Datenstrukturen entsteht auf natürliche Weise der Wunsch, mit Hilfe geeigneter Algorithmen möglichst effizient die Darstellung einer Booleschen Funktion in einem Modell in eine Darstellung derselben Funktion in einem anderen Modell zu transformieren. Solche Transformationsalgorithmen sind Gegenstand dieser Diplomarbeit. Dabei wird hier nicht nur der Wechsel von einem BDD-Modell zu einem anderen sondern auch der Wechsel der Variablenordnung innerhalb eines BDD-Modells als Transformation aufgefaßt. Somit können Transformationsalgorithmen auch als Teile von Programmen verwendet werden, die versuchen, zu einer gegebenen BDD-Instanz eine bessere Ordnung zu berechnen.

Alle BDD-artigen Datenstrukturen sind graphbasiert und können angemessen in zwei Schritten beschrieben werden. Der erste Teil einer BDD-Definition besteht aus der Beschreibung der *Syntax*, das sind die formalen Regeln, denen der Graph und alle anderen Teile des beschriebenen BDD-Modells gehorchen müssen. Den zweiten Definitionsteil bildet die *Semantik*, die festlegt, wie eine bestimmte Instanz eines BDD-Modells zu interpretieren ist. Die Semantik eines BDD-Modells kann auf verschiedene Weisen festgelegt werden: man kann etwa definieren, wie zu einer BDD-Instanz die repräsentierte Funktion f berechnet werden kann, oder man kann definieren, wie zu einer BDD-Instanz und einer Variablenbelegung $a \in \{0,1\}^n$ der Funktionswert $f(a) \in \{0,1\}$ berechnet wird. Im Rahmen dieser Arbeit wird von beiden Definitionsmöglichkeiten für die Semantik Gebrauch gemacht.

Im nächsten Kapitel werden alle betrachteten BDD-Modelle kurz definiert und ihre für diese Arbeit wichtigen Eigenschaften besprochen[3]. Das Kapitel 3 ist den Transformationsalgorithmen gewidmet. Dabei werden sowohl effiziente Algorithmen vorgestellt, als auch Algorithmen, die im worst case exponentielle Laufzeit und exponentiellen Platzbedarf haben. Um die letztgenannten Algorithmen besser einschätzen zu können, werden wir über diese Algorithmen empirisch gewonnene Aussagen machen. Das vierte Kapitel faßt die Ergebnisse zusammen, zeigt Perspektiven auf und bespricht kurz die Kombination hier vorgestellter Algorithmen. In zwei Anhängen werden schließlich noch einige Aspekte konkreter Implementierungen besprochen.

[3]Eine Übersicht über verschiedene Modelle findet man bei Wegener (1994) und Bryant (1995). Erste Grundlage für die Darstellung hier ist häufig Wegener (1994/95).

2 BDD-Modelle: Definitionen, Eigenschaften, Algorithmen

Boolesche Funktionen $f : \{0,1\}^n \to \{0,1\}^m$ haben n Eingangsvariablen und berechnen m Funktionswerte. Man kann sich bei der Konstruktion von Transformationsalgorithmen auf Funktionen $f_p : \{0,1\}^n \to \{0,1\}$ beschränken, auf Funktionen also, die nur einen binären Funktionswert berechnen. Jede allgemeine Boolesche Funktion f mit einem m-dimensionalem 0-1-Wertevektor kann offenbar durch m geeignete Funktionen $f_1, \ldots, f_m$ ersetzt werden, die alle einen binären Funktionswert haben. Die Eingangsvariablen werden o. B. d. A. im folgenden stets mit $x_1, \ldots, x_n$ benannt.

Als gemeinsame Anforderung an Datenstrukturen für Boolesche Funktionen läßt sich außer dem Wunsch nach moderater Größe der Datenstruktur eine Reihe von Operationen formulieren, die möglichst alle effizient durchführbar sein sollen. Eine Liste wünschenswerter Operationen enthält vermutlich meist auch die folgenden Operationen, wenn die Datenstruktur eine Boolesche Funktion $f : \{0,1\}^n \to \{0,1\}$ repräsentiert.

Auswertung: Berechne zu einer Variablenbelegung $a \in \{0,1\}^n$ den Funktionswert $f(a)$.

Erfüllbarkeit: Teste, ob es eine Variablenbelegung a gibt, für die $f(a) = 1$ gilt.

Minimierung: Berechne aus der gegebenen Darstellung eine kleinste Darstellung von f; ist diese Darstellung eindeutig, spricht man von Reduktion.

Äquivalenztest: Entscheide, ob zwei verschiedene Instanzen eines BDD-Modells die gleiche Funktion darstellen. Für den Äquivalenztest ist es offenbar wünschenswert, daß eine Boolesche Funktion eine eindeutige Darstellung besitzt. In diesem Fall stellen zwei Instanzen genau dann die gleiche Funktion dar, wenn die beiden Instanzen isomorph sind.

Synthese: Zu einer Darstellung von f, einer Darstellung von g und einem Booleschen Operator $\otimes$ berechne eine Darstellung von $f \otimes g$.

Ersetzung durch Konstante: Zu einer Darstellung von f, einer Variablen x_i aus der Variablenmenge und einer Konstanten $c \in \{0,1\}$ soll eine Darstellung für $f_{|x_i=c}$ berechnet werden[4].

Redundanztest: Zu einer Darstellung von f und einer Variablen x_i aus der Variablenmenge soll berechnet werden, ob die Variable x_i redundant ist. Wir nennen eine Variable x_i redundant, wenn $f_{|x_i=0} = f_{|x_i=1}$ gilt, andernfalls sagen wir, die Funktion f hängt essentiell von der Variablen x_i ab.

Die Ersetzung einer Variablen durch eine Konstante bezeichnen wir auch abkürzend als Konstantsetzung. In Abhängigkeit von der jeweils verwendeten Booleschen Konstanten

[4] Wir bezeichnen alle durch Ersetzung mindestens einer Variablen durch (möglicherweise verschiedene) Konstante aus einer Funktion f entstehenden Funktionen f_s als Subfunktionen von f.

erlauben wir uns auch gelegentlich kurz und prägnant von 0-Setzungen oder 1-Setzungen zu sprechen.

In diesem Kapitel werden alle im Rahmen dieser Arbeit berührten BDD-Modelle definiert und kurz diskutiert. Es werden, wo immer möglich, effiziente Algorithmen vorgestellt, insofern sie für die Transformationsalgorithmen aus Kapitel 3 notwendig sind. Außerdem werden zentrale Eigenschaften der verschiedenen BDD-Modelle diskutiert und abschließend tabellarisch gegenübergestellt.

Als gemeinsamer Ausgangspunkt aller BDD-Modelle können Branching Programme (BP) oder Binary Decision Diagrams (BDDs) betrachtet werden, die ein relativ altes und wohlbekanntes Arbeitsmittel der Informatik darstellen (Lee (1959)). Ein BDD ist eine graphbasierte Darstellung einer Booleschen Funktion: es besteht aus einem gerichteten, zusammenhängenden, azyklischen Graphen mit genau einer Quelle. Jeder Knoten außer den Senken ist mit einer Variablen markiert, die Senken tragen eine Boolesche Konstante als Markierung. Jeder innere Knoten hat genau zwei ausgehende Kanten, von denen eine mit 0 und die andere mit 1 markiert ist.

Ist G ein BDD, so bezeichnen wir mit $|G|$ die Größe von G. Wir gehen stets davon aus, daß zu einem BDD nur Knoten gehören, die von der Quelle aus erreichbar sind. Wir bezeichnen als Größe eines BDD die Anzahl der Knoten im Graphen; es ist aber für die Analyse unerheblich, ob wir damit konkret die Anzahl der Knoten oder die Anzahl der Kanten meinen, wenn wir uns nur für Ergebnisse in O-Notation interessieren; die Anzahl der Kanten liegt stets zwischen $|V| - 1$ und $2\,|V| - 2$, wenn V die Menge der Knoten ist und nicht nur aus genau einer Senke besteht.

Zur Berechnung des Funktionswertes zu einer gegebenen Belegung der Variablen durchläuft man ein solches BDD von der Quelle bis zu einer Senke. An jedem inneren Knoten entscheidet man sich je nach der Belegung der an dem Knoten notierten Variablen für den 0- oder 1-Nachfolger. Die schließlich erreichte Senke bezeichnet den von f auf dieser Eingabe berechneten Funktionswert.

Allgemeine BDDs stellen eine leicht zu implementierende und platzsparende Darstellung für Boolesche Funktionen dar. Ihre Größe kann in vielen Fällen viel kleiner als exponentiell in der Anzahl der Variablen sein, so daß sie etwa einer Wertetabelle oder einem Wertevektor deutlich überlegen sind. BDDs sind aber mit einer ganzen Reihe von Nachteilen behaftet, von denen einige hier genannt werden sollen.

Die Darstellung einer Booleschen Funktion ist nicht eindeutig: viele verschiedene BDDs können die gleiche Funktion repräsentieren. Eindeutigkeit ist eine wichtige Eigenschaft, weil dann zwei Darstellungen effizient daraufhin getestet werden können, ob sie die gleiche Funktion repräsentieren. Eng damit verknüpft ist der Umstand, daß man in konkreten Implementierungen dann die oft kritische Ressource Platz schonen kann, indem man gleiche Funktionen nur einmal darstellt.

Der Erfüllbarkeitstest für allgemeine BDDs ist NP-vollständig. Das liegt offenbar daran, daß es Pfade zu einer 1-Senke geben kann, die keiner Variablenbelegung entsprechen. Auf einem Pfad kann jede Variablenmarkierung mehrfach vorkommen, so daß für manche Pfade zu einer 1-Senke dieselbe Variable einmal über die 0-Kante und einmal über die

1-Kante verlassen wird. Solche Pfade nennt man null-chains. Offensichtlich läßt sich dieses Problem direkt beheben, wenn man zusätzlich fordert, daß auf jedem Pfad von der Quelle zu einer Senke jede Variable höchstens einmal als Knotenmarkierung vorkommen darf. Dann gibt es für jeden Pfad von der Quelle zu einer Senke auch mindestens eine zugehörige Belegung der Variablen, die erzwingt, daß bei der Auswertung genau dieser Pfad begangen wird. Folglich ist die dargestellte Funktion genau dann erfüllbar, wenn es einen Pfad von der Quelle zu einer 1-Senke gibt. Diese Überlegung führt zum ersten BDD-Modell, dem wir nähere Aufmerksamkeit widmen, den FBDDs.

2.1 Free Binary Decision Diagrams (FBDDs) und Ordered Binary Decision Diagrams (OBDDs)

Definition 2.1: Syntax: Ein *FBDD* (free binary decision diagram) ist ein gerichteter, azyklischer Graph mit genau einer Quelle. Alle Knoten außer den Senken besitzen zwei ausgehende Kanten, von denen eine mit 0 und eine mit 1 markiert ist. Den Knoten, den man über die mit 0 markierte Kante eines Knotens v erreicht, nennen wir *0-Nachfolger* von v, den über die mit 1 markierte Kante erreichten Knoten nennen wir *1-Nachfolger* von v. Alle Knoten außer den Senken sind mit einem Variablennamen markiert. Jede Senke ist entweder mit 0 oder mit 1 markiert. Auf jedem gerichteten Pfad von der Quelle zu einer Senke kommt jeder Variablenname höchstens einmal vor[5].

Semantik: Zu einem FBDD für die Funktion $f : \{0,1\}^n \to \{0,1\}$ und einer Variablenbelegung $a = (a_1, \ldots, a_n) \in \{0,1\}^n$ erhält man den Funktionswert $f(a)$ wie folgt: Anfangs ist der aktuelle Knoten die Quelle. Solange der aktuelle Knoten keine Senke ist, sei x_i die Markierung dieses Knotens. Wenn $a_i = 0$ ist, wird der 0-Nachfolger der neue aktuelle Knoten, sonst der 1-Nachfolger. Wenn der aktuelle Knoten eine Senke ist, bezeichnet die Knotenmarkierung den Funktionswert.

FBDDs haben eine Reihe erfreulicher Eigenschaften. So ist die Auswertung immer in Zeit $O(n)$ möglich: es genügt ein Durchlaufen des Graphen von der Quelle zu einer der Senken. Weil auf jedem solchem Pfad jede Variable höchstens einmal vorkommen darf, ist die Pfadlänge durch n nach oben beschränkt. Auch der Erfüllbarkeitstest ist effizient möglich, wie wir schon in der Einleitung dieses Kapitels diskutiert haben. Weil jedem Pfad von der Quelle zu einer Senke mindestens eine Variablenbelegung zugeordnet werden kann, genügt es, im Rahmen einer Tiefensuche festzustellen, ob eine 1-Senke von der Quelle aus erreicht werden kann. Die Laufzeit für den Erfüllbarkeitstest beträgt also $O(|G|)$ für ein FBDD G. Auch die Ersetzung von Variablen durch Konstante kann effizient im Rahmen einer Tiefensuche durchgeführt werden. Wenn $x_i := c$ gesetzt werden soll, so genügt es, jeden Zeiger, der in einen x_i-Knoten hineinführt, auf den c-Nachfolger des x_i-Knoten zu setzen. Die Ersetzung kann also für beide Booleschen Konstanten auf die gleiche Art durchgeführt werden: FBDDs unterscheiden 0 und 1 nicht weiter. Diese Eigenschaft ist

[5] Man nennt diese Eigenschaft von FBDDs „read-once“.

nicht selbstverständlich für BDD-Modelle; wir werden sehen, daß Varianten mit anderen Auswertungsregeln gerade bei der Ersetzung von Variablen durch Konstante eine klare Unterscheidung von 0-Setzungen und 1-Setzungen erfordern.

Nicht alle Operationen lassen sich so effizient wie die Auswertung, der Erfüllbarkeitstest und die Konstantsetzung ausführen; wir müssen jetzt zu schlechteren Nachrichten kommen. Als erstes ist zu nennen, daß FBDDs Boolesche Funktionen nicht eindeutig darstellen; eine Funktion f kann durch viele verschiedene FBDDs repräsentiert werden. Das ist auch offensichtlich: so spielt z. B. bei allen symmetrischen Funktionen die Reihenfolge der Tests der Variablen keine Rolle (vgl. etwa Abbildung 1).

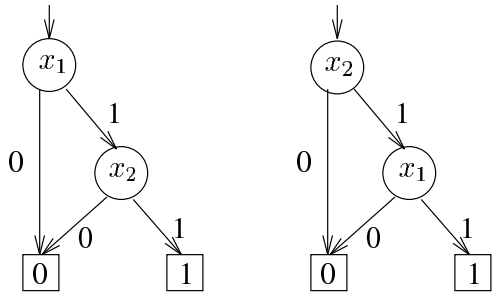


Abbildung 1: Zwei verschiedene FBDDs für die Funktion $x_1 \wedge x_2$

Ein einfacher Graphisomorphietest genügt also nicht, um einen Äquivalenztest zu realisieren. So muß es auch nicht verwundern, daß kein deterministischer Polynomialzeitalgorithmus für den Äquivalenztest bekannt ist; es gibt allerdings einen probabilistischen Algorithmus mit linearer Laufzeit, der zwei äquivalente Darstellungen sicher als äquivalent identifiziert, jedoch möglicherweise zwei verschiedene Funktionen fälschlich als gleich bezeichnet (Blum, Chandra und Wegmann (1980)).

Ein weiteres ernsthaftes Problem ist die Synthese. Wir definieren zwei Funktionen f und g auf den n^2 Variablen x_{ij} für $1 \leq i, j \leq n$. Die Variablen definieren eine $n \times n$-Matrix $X = (x_{ij})$, deren Einträge Boolesche Konstante sind. Es ist $f(X) = 1 :\Leftrightarrow X$ enthält eine 1-Zeile und $g(X) = 1 :\Leftrightarrow X$ enthält eine 1-Spalte. Beide Funktionen können durch je ein FBDD dargestellt werden, das nur $n^2 + 2$ Knoten enthält und also lineare Größe hat. Dazu sind jeweils nacheinander die Variablen einer Zeile für f und einer Spalte für g mit $\wedge$ zu verknüpfen analog zu Abbildung 2. Wir bemerken ohne Beweis, daß jedes FBDD für $f \vee g$ Größe $\Omega(2^{\frac{1}{2}n})$ hat[6].

In der Praxis werden BDD-Darstellungen von Funktionen häufig aus Schaltkreisen berechnet. Darstellungen für die Eingänge lassen sich leicht direkt angeben. Hat man für alle Vorgänger eines Gatters des Schaltkreises äquivalente BDDs berechnet, so kann man durch eine geeignete Synthese ein BDD für das betrachtete Gatter berechnen. Synthese spielt also in praktischen Anwendungen eine entscheidende Rolle. Aus diesem Grund (und weil echte Verifikation mit dem probabilistischen Äquivalenztest nicht möglich ist) stellen

[6]Ein entsprechender Beweis stammt von Bollig (1993).

FBDDs kein durchgängig geeignetes Modell dar. Damit ist die Einführung einer Variante mit besseren Eigenschaften motiviert.

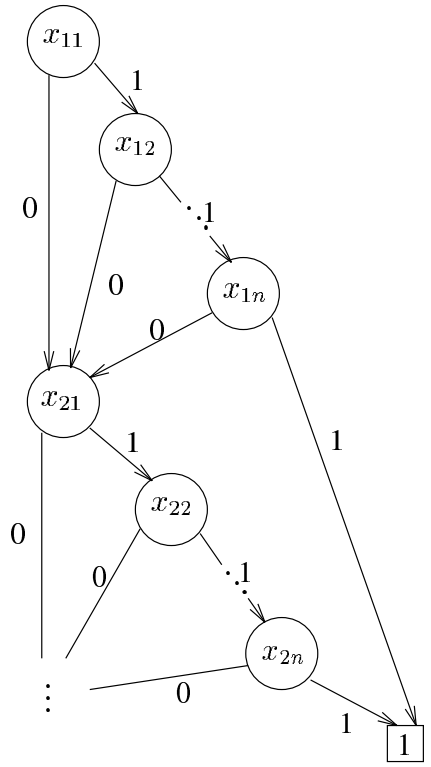


Abbildung 2: Ein FBDD für f

OBDDs sind eingeschränkte FBDDs, die von Bryant (1986) eingeführt wurden. Sie haben gegenüber FBDDs verschiedene Vorteile: bei der Synthese ist die Größe des Ergebnisses beschränkt durch das Produkt der Größen der beiden Eingabe-OBDDs. Ein gegebenes OBDD zu einer Booleschen Funktion f kann effizient auf minimale Größe verkleinert werden, zusätzlich ist diese Darstellung minimaler Größe eindeutig bis auf Isomorphie. Folglich sind solche reduzierten OBDDs eine eindeutige Darstellung Boolescher Funktionen, der Gleichheitstest ist also trivial in Linearzeit durchführbar. OBDDs sind die zur Zeit verbreitetste Datenstruktur für Boolesche Funktionen. Die in dieser Arbeit vorgestellten Transformationsalgorithmen benutzen OBDDs als das zentrale Modell, von dem aus und auf das hin die Algorithmen überwiegend ausgerichtet sind. Aus diesem Grund werden OBDDs hier in einiger Ausführlichkeit vorgestellt[7].

Definition 2.2: Syntax: Ein *OBDD* ist ein FBDD, dem eine Permutation π der Variablenmenge als Variablenordnung zugeordnet wird mit folgender Eigenschaft: Auf jedem gerichteten Pfad von der Quelle zu einer Senke wird die durch π gegebene Ordnung der Variablen respektiert, d. h. wenn ein Knoten u mit Markierung x_i vor einem Knoten v mit Markierung x_j im Pfad vorkommt, so gilt $\pi(x_i) < \pi(x_j)$.

[7]Eine deutlich ausführlichere Darstellung von OBDDs, die auch Implementierungsaspekte und vor allem Anwendungen berührt, findet man bei Bryant (1992).

Semantik: Für OBDDs ist dieselbe Auswertungsvorschrift wie für FBDDs anzuwenden.

Weil zu einem OBDD G stets eine Permutation π der Variablen $x_1, \ldots, x_n$ gehört, nennt man G auch ein πOBDD. Weil bei einem πOBDD G auf allen Pfaden von der Quelle zu einer Senke die Variablenordnung π respektiert wird, kann man die Knoten von G so in $n+1$ Schichten einteilen, daß die i-te Schicht ($1 \leq i \leq n$) nur Knoten mit Markierung x_i enthält, alle Senken zu Schicht $n+1$ gehören und alle Kanten nur von Schichten i zu Schichten j mit $i < j$ führen. In diesem Sinne kann man in πOBDDs sinnvoll von „oben" und „unten" sprechen, wobei die Quelle ganz oben angesiedelt wird. Diese graphische Vorstellung von BDDs läßt uns später dann auch Algorithmen die Begriffe „top-down" (von der Quelle abwärts) oder „bottom-up" (von den Senken aufwärts) zuordnen. Es ist klar, daß das Aufzählen der Knoten top-down (bottom-up) eine (umgekehrt) topologische Ordnung der Knoten ergibt.

Man kann zu einem gegebenen FBDD oder OBDD leicht beschreiben, welche Funktion es repräsentiert (vgl. auch Abbildung 3). Besteht das FBDD/OBDD nur aus einer Senke, so stellt es die entsprechende konstante Funktion dar. Sonst besteht es aus einer Quelle mit Markierung x_i, der 0-Nachfolger ist die Quelle eines FBDD/OBDD für die Funktion g und der 1-Nachfolger ist die Quelle eines FBDD/OBDD für die Funktion h[8]. Das FBDD/OBDD stellt die Funktion $f = \overline{x_i} g \vee x_i h$ dar[9].

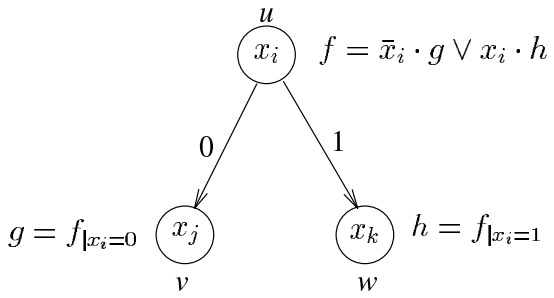


Abbildung 3: Funktionsrepräsentation im OBDD

Umgekehrt kann man dann folgende Aussage machen: Stellt ein FBDD/OBDD mit Quellenmarkierung x_i die Funktion f dar, so stellt das 0-Unter-FBDD/OBDD die Funktion $f_{|x_i=0}$ und das 1-Unter-FBDD/OBDD die Funktion $f_{|x_i=1}$ dar. FBDDs und OBDDs beruhen also auf der Shannon-Zerlegung:

$$f = \overline{x_i} f_{|x_i=0} \vee x_i f_{|x_i=1}.$$

Wir werden später sehen, daß auch andere Zerlegungen an den Knoten denkbar und sinnvoll sind[10].

[8] Der Umstand, daß die beiden Unter-FBDDs/OBDDs nicht notwendig disjunkt sind, spielt keine Rolle.

[9] Man beobachtet direkt, daß weder g noch h essentiell von x_i abhängen können, da gemäß Definition 2.1 bzw. Definition 2.2 die entsprechenden Unter-FBDDs/OBDDs keinen x_i-Knoten enthalten können. Es gilt also für alle $a \in \{0,1\}^n$ $g(a) = g_{|x_i=0}(a) = g_{|x_i=1}(a)$ und $h(a) = h_{|x_i=0}(a) = h_{|x_i=1}(a)$.

[10] Becker und Drechsler (1995) beschäftigen sich explizit mit der Betrachtung verschiedener Funktionszerlegungen an den Knoten.

Die Auswertung, der Erfüllbarkeitstest und die Ersetzung durch Konstante können bei OBDDs genau wie bei FBDDs durchgeführt werden; die Variablenordnung spielt für die Algorithmen keine Rolle.

Eine zentrale Eigenschaft von πOBDDs ist, daß sie effizient minimiert werden können und minimierte πOBDDs eine eindeutige Darstellung Boolescher Funktionen sind (bis auf Isomorphie). Darum ist ein effizienter Äquivalenztest möglich: es genügt ein Isomorphietest der beiden gegebenen πOBDDs, der einfach ist, weil die Quellen aufeinander abgebildet werden müssen, die Kanten gerichtet und alle Knoten markiert sind. Für den Nachweis der wichtigen Tatsache, daß minimierte πOBDDs eine eindeutige Darstellung Boolescher Funktionen sind, geben wir einen entsprechenden Satz mit Beweis von Wegener (1993) wieder.

Satz 2.3: Für jede Funktion $f : \{0,1\}^n \to \{0,1\}$ und jede Variablenordnung π gibt es bis auf Isomorphie ein eindeutiges πOBDD G minimaler Größe, das explizit mit Eigenschaften von f und π beschrieben werden kann.

Beweis: Wir nehmen o. B. d. A. an, daß π die Identität ist. Andernfalls benennen wir die Variablen um und operieren auf einer neuen Variablenmenge. Für konstante Funktionen genügt offensichtlich die entsprechende Senke, die als Quelle fungiert. Sei also f nicht konstant. Dann enthält G notwendig zwei verschieden markierte Senken; mehr als eine 0-Senke und eine 1-Senke sind offensichtlich nicht nötig. Wir bestimmen jetzt, wie viele verschiedene Knoten mit Markierung x_i es in G geben muß. Dazu betrachten wir für alle Vektoren $a = (a_1, \ldots, a_{i-1}) \in \{0,1\}^{i-1}$ die Subfunktionen $f_{|x_1=a_1,\ldots,x_{i-1}=a_{i-1}}$. Jeder Vektor a definiert einen Pfad von der Quelle zu einem Knoten, der die Funktion $f_{|x_1=a_1,\ldots,x_{i-1}=a_{i-1}}$ darstellt. Wenn diese Funktion nicht essentiell von x_i abhängt, kann der x_i-Test ausgelassen werden. Andernfalls muß der erreichte Knoten ein x_i-Knoten sein; ist dieser Knoten ein x_j-Knoten mit $j > i$, so kann wegen der Variablenordnung kein x_i-Knoten mehr vorkommen. Kommt aber kein x_i-Knoten für diese Subfunktion vor, so wird für wenigstens eine Eingabe ein falscher Wert berechnet. Falls zwei verschiedene Vektoren a, a' die gleiche Subfunktion erzeugen, so können die zugehörigen Pfade auch zum gleichen Knoten führen. Verschiedene Funktionen müssen durch verschiedene Knoten dargestellt werden, da keine Variablenmarkierung auf einem Pfad mehrfach vorkommen darf und somit keine Tests wiederholt werden können. G muß also mindestens so viele x_i-Knoten enthalten, wie es essentiell von x_i abhängende verschiedene Subfunktionen $f_{|x_1=a_1,\ldots,x_{i-1}=a_{i-1}}$ gibt. Daß diese Anzahl auch ausreicht, zeigen wir konstruktiv. Es sei $f_{|x_1=a_1,\ldots,x_{i-1}=a_{i-1}}$ eine essentiell von x_i abhängende Subfunktion, die durch den Knoten v mit Markierung x_i dargestellt werden soll. Wir definieren die Nachfolger von v. Falls $f_{|x_1=a_1,\ldots,x_{i-1}=a_{i-1},x_i=0}$ konstant ist, so wird der 0-Nachfolger die geeignete Senke. Andernfalls wird 0-Nachfolger der Knoten, der die Funktion $f_{|x_1=a_1,\ldots,x_{i-1}=a_{i-1},x_i=0}$ darstellt. Die Markierung dieses Knotens ist x_j, wenn j der kleinste Index ist, so daß $f_{|x_1=a_1,\ldots,x_{i-1}=a_{i-1},x_i=0}$ essentiell von x_j abhängt. Der 1-Nachfolger wird analog mittels $f_{|x_1=a_1,\ldots,x_{i-1}=a_{i-1},x_i=1}$ definiert.

Wir zeigen durch vollständige Induktion bottom-up, daß jeder Knoten die mit ihm assoziierte Funktion repräsentiert. Für die Senken ist das offensichtlich. Sei nun v ein Knoten

mit Markierung x_i, der mit der Funktion $f_{|x_1=a_1,\ldots,x_{i-1}=a_{i-1}}$ assoziiert ist. Nach Induktionsvoraussetzung stellt der 0-Nachfolger von v die Funktion $f_{|x_1=a_1,\ldots,x_{i-1}=a_{i-1},x_i=0}$ und der 1-Nachfolger die Funktion $f_{|x_1=a_1,\ldots,x_{i-1}=a_{i-1},x_i=1}$ dar. Folglich stellt nach unseren Beobachtungen oben v die Funktion

$$\overline{x_i} \cdot f_{|x_1=a_1,\ldots,x_{i-1}=a_{i-1},x_i=0} \vee x_i \cdot f_{|x_1=a_1,\ldots,x_{i-1}=a_{i-1},x_i=1} = f_{|x_1=a_1,\ldots,x_{i-1}=a_{i-1}}$$

dar. Also stellt die Quelle die mit ihr assoziierte Funktion f dar.

Folglich muß ein πOBDD für f mindestens soviele Knoten enthalten wie G. Wenn seine Knotenanzahl ebenso groß ist, muß es zu G isomorph sein, da wir bei der Festlegung der Kanten keine Wahl hatten. □

Weil es bis auf Isomorphie zu einer Funktion f ein eindeutiges minimales πOBDD gibt, ist die Reduktion eines πOBDD ein wohldefiniertes Problem. Tatsächlich genügt die Anwendung von zwei Reduktionsregeln, um ein gegebenes πOBDD vollständig zu reduzieren.

Satz 2.4: [11] Sei G ein πOBDD für f, in dem alle Knoten von der Quelle aus erreichbar sind und in dem die folgenden beiden Reduktionsregeln nicht anwendbar sind.

Eliminationsregel: Wenn die beiden Nachfolger eines Knotens v derselbe Knoten w sind, so kann v eliminiert werden; alle auf v zeigenden Kanten werden durch Kanten auf w ersetzt.

Verschmelzungsregel: Wenn zwei Knoten v und w die gleiche Markierung, den gleichen 0-Nachfolger und den gleichen 1-Nachfolger haben, so können die beiden Knoten verschmolzen werden. Zwei Senken mit gleicher Markierung können ebenfalls verschmolzen werden. Einer der beiden zu verschmelzenden Knoten wird entfernt, Kanten auf ihn werden durch Kanten auf den anderen Knoten ersetzt.

Dann ist G das reduzierte πOBDD für f.

Vor dem Beweis bemerken wir, daß x_i-Knoten, die nach der Eliminationsregel entfernt werden können, eine nicht essentiell von x_i abhängende Subfunktion repräsentieren. Knoten v und w, die gemäß Verschmelzungsregel verschmolzen werden können, stellen die gleiche Subfunktion dar.

Beweis: Zunächst ist offensichtlich, daß die Aussage für konstante Funktionen richtig ist. Wir nehmen also im folgenden an, daß f nicht konstant ist. Wir führen einen Beweis durch Widerspruch. Sei F das reduzierte πOBDD für f. Nach Satz 2.3 hat G mindestens soviele Knoten wie F. Haben G und F gleiche Knotenzahl, so sind sie gemäß Satz 2.3 isomorph. Wir können also annehmen, daß G mehr Knoten als F enthält. G und F enthalten beide jede Senke nur einmal; sonst könnten gleich markierte Senken verschmolzen werden. Wir wählen i maximal so, daß G mindestens einen x_i-Knoten mehr enthält als F. Weil $|G| >$

[11] Satz und Beweis sind Wegener (1993) entnommen.

$|F|$ gilt, gibt es notwendigerweise solch ein i. Weil F gerade so viele x_i-Knoten enthält, wie es unterschiedliche essentiell von x_i abhängende Subfunktionen gibt, enthält G einen x_i-Knoten, der entweder eine nicht essentiell von x_i abhängende Subfunktion h repräsentiert oder zwei x_i-Knoten, die dieselbe Subfunktion h repräsentieren. Im ersten Fall ist $h_{|x_i=0} = h_{|x_i=1}$; gemäß Wahl von i sind F und G unterhalb von i isomorph. Also sind 0- und 1-Nachfolger identisch, und der Knoten kann eliminiert werden. Im zweiten Fall folgt mit dem gleichen Argument, daß jeweils die Nachfolger gleich sind, so daß die Knoten verschmolzen werden können. Es ist also in jedem Fall im Widerspruch zur Voraussetzung eine der beiden Reduktionsregeln anwendbar. □

Man kann die Reduktion eines gegebenen OBDD sogar in Linearzeit durchführen: Man erreicht im Rahmen einer Tiefensuche, daß alle Knoten von der Quelle aus erreichbar sind. Dabei sammelt man die x_i-Knoten in Listen L_i und berechnet zu jedem Knoten eine Liste der eingehenden Kanten. Dann durchläuft man die Listen L_i bottom-up und führt die möglichen Anwendungen der Reduktionsregeln durch. Für die Verschmelzungen müssen die x_i-Knoten nach ihren Nachfolgerpaaren sortiert werden, damit gleiche Nachfolgerpaare effizient erkannt werden können. Der Reduktionsalgorithmus von Bryant (1986) reduziert ein OBDD G auf diese Art in Zeit $O(|G| \log |G|)$ auf Platz $O(|G|)$, wobei die Laufzeit durch das Sortieren der Knoten dominiert wird. Weil aber die Knoten nur nach den Markierungen ihrer Nachfolger sortiert werden müssen, es also je Nachfolger nur $n+1$ verschiedene Werte für das Sortierkriterium gibt, kann man das Sortieren auch etwas geschickter in Linearzeit erledigen. Um das zu realisieren, kann man eine 2-Phasen-Bucketsort-Technik verwenden. Der resultierende Algorithmus stammt von Sieling und Wegener (1993) und kann im Detail wie folgt beschrieben werden.

Eingabe für den Algorithmus ist ein Graph $G = (V, E)$ mit Knotennummern 1 bis $|V|$. Alle Knoten sind von der Quelle aus erreichbar. Wenn diese Bedingung nicht erfüllt ist, kann sie im Rahmen eines Tiefendurchlaufs leicht hergestellt werden. Jeder Knoten wird im Algorithmus mit seinem eindeutigen Bezeichner identifiziert. Dabei ist es für den Algorithmus wesentlich, daß als Bezeichner lückenlos die Zahlen von 1 bis $|V|$ verwendet werden. Ist das in G nicht so gegeben, kann auch diese Forderung im Rahmen des Tiefendurchlaufs hergestellt werden.

Wir bezeichnen zu einem Knoten v seine Markierung mit `v-label`. Sein 0-Nachfolger wird mit `v→links`, sein 1-Nachfolger mit `v→rechts` bezeichnet. Diese Bezeichnung deutet an, daß in graphischen Darstellungen von OBDDs häufig auf die Markierung der Kanten verzichtet wird. Man vereinbart dann, daß die linke Kante der 0-Kante und die rechte Kante der 1-Kante entspricht.

Der Algorithmus verwendet als zusätzliche Datenstrukturen (außer Listen L_i, die alle Knoten mit Markierung x_i enthalten) drei Arrays über der Knotenmenge V und eine zusätzliche Knotenliste. Im einzelnen sind das folgende Datenstrukturen:

- `num: array[1..`$|V|$`] of {1,...,`$|V|$`}`; in diesem Array wird zu v in `num[`v`]` der Repräsentant von v vermerkt. Eingangs ist also `num[`v`]=`v für alle Knoten; ist am Ende des Algorithmus `num[`v`]`$\neq v$, so kann der Knoten v aus G entfernt werden.

Der Begriff „Repräsentant“ kann zurecht im Sinne von „Repräsentant einer Äquivalenzklasse“ verstanden werden.

- `left: array[1..`$|V|$`] of Knotenliste`; in diesem Array werden zu einem Knoten v in `left[`v`]` alle Väter von v gespeichert, für die v 0-Nachfolger ist. Dabei werden schon nicht die Original-0-Nachfolger verwendet; es werden stattdessen die Repräsentanten der 0-Nachfolger herangezogen.

- `L-active: Knotenliste`; diese Liste enthält alle x_i-Knoten (i ist die Nummer der aktuellen Knotenliste), die echt verschiedene 0-Nachfolger haben. Echt verschieden bedeutet, daß die 0-Nachfolger verschiedene Repräsentanten haben.

- `right: array[1..`$|V|$`] of` $\{0,\ldots,|V|\}$; dieses Array enthält für einen Knoten v in `right[`v`]` den Repräsentanten des 1-Nachfolgers des aktuellen Knotens aus der Knotenliste `L-active`. Knoten aus `left[`v`]`, die hier den gleichen Repräsentanten vermerkt haben, sind äquivalent.

Der Zugriff auf die Listen folgt zum einen mit der Zuweisung des besonderen Werts `nil`, der eine leere Liste bezeichnet. Initialisierend soll die Zuweisung in konstanter Zeit möglich sein, enthält die Liste bereits i Elemente, soll die Zuweisung (die das vollständige Leeren der Liste bedeutet) in Zeit $O(i)$ durchführbar sein. Das Einfügen neuer Elemente in die Liste wird mit der Anweisung `Insert(Element,Liste)` durchgeführt; dabei ist uns gleichgültig, wo das Element eingefügt wird. Wichtig ist aber, daß jedes Einfügen nur konstante Zeit benötigt. Schließlich soll durch ein Konstrukt `for Element` $\in$ `Liste` die Variable `Element` nacheinander jedes Listenelement genau einmal repräsentieren; die Listenelemente bleiben dabei in der Liste erhalten und werden nicht etwa automatisch entfernt.

Wir werden nicht festlegen, wie Graphen (also Knoten und Kanten) im Rahmen einer Implementierung zu realisieren sind, wollen aber auch hier einige leicht zu erfüllende Anforderungen an die Laufzeit festhalten. Knoten mit den zu ihnen gehörigen Werten und Kanten sollen, wenn alle Daten zur Verfügung stehen, in konstanter Zeit angelegt und entfernt werden können. Insbesondere gehen wir bei der Analyse von Algorithmen (nicht explizit beim folgenden Algorithmus, jedoch bei vielen Algorithmen des zentralen Abschnitts 3) davon aus, daß entfernte Knoten keinen Speicherplatz mehr belegen.

Algorithmus 2.5 (OBDD-Reduktion):

```
/* Eingabe: πOBDD G=(V,E) für f (o.B.d.A. π=id) */
/* Ausgabe: reduziertes πOBDD für f */
1.   Erstelle Knotenlisten L_1,...,L_n,L_=0,L_=1
     für die Knoten mit Markierung x_1,...,x_n,0,1;
2.   For i:=1 to |V| do
          num[i]:=i; left[i]:=nil; right[i]:=0;
3.   L-Active:=nil;
```

```
4.   for i:=0 to 1 do
          j:=Head(L_=i);
          For v∈ L_=i do
               num[v]:=j;
5.   For i:=n downto 1 do
6.        For v∈ L_i do
7.             If num[v→links]=num[v→rechts]
               Then num[v]:=num[v→links];
               Else If left[num[v→links]]=nil
                    Then Insert(num[v→rechts],L-Active);
                    Insert(v,left[num[v→links]]);
8.        For v∈L-Active do
9.             For w∈Left[v] do
10.            If right[num[w→rechts]]=0
               Then right[num[w→rechts]]:=w;
               Else num[w]:=right[num[w→rechts]];
11.       For w∈Left[v] do
               right[num[w→rechts]]:=0;
12.       L-Active:=nil;
13.  For i:=1 to |V| do
          If num[v]≠v
          Then Eliminiere v;
          Else v→links:=num[v→links];
               v→rechts:=num[v→rechts];
```

Die Korrektheit folgt ziemlich direkt aus der Korrektheit der Eliminationsregeln; formal ist ein Beweis mittels vollständiger Induktion über die Knotenmarkierungen von unten nach oben zu führen. Nach den Initialisierungen werden überflüssige Senken entfernt (Zeile 4): gleichmarkierte Senken werden mit Hilfes des `num`-Arrays als äquivalent gekennzeichnet. Zentral ist die Hauptschleife, die bottom-up über alle Knotenmarkierungen läuft (Zeilen 5–12). Offenbar werden zunächst überflüssige Knoten gemäß Eliminationsregel entfernt (Zeile 7), anschließend werden äquivalente Knoten gemäß Verschmelzungsregel miteinander verschmolzen (Zeile 10). Man beachte, daß das eigentliche Entfernen erst am Ende durchgeführt wird (Zeile 13). Knoten werden zum Entfernen markiert, indem sie im `num`-Array eine andere als ihre eigene Knotennummer erhalten.

Vor der Laufzeitabschätzung schicken wir voraus, daß wir o. B. d. A. annehmen, daß das zu reduzierende OBDD für alle $1 \leq i \leq n$ einen x_i-Knoten enthält. Ist das nicht der Fall, operiert man auf einer kleineren Variablenmenge. Unter dieser Voraussetzung ist die Laufzeit in den Vor- und Nachbereitungen offenbar linear in der Anzahl der Knoten. Für den Kern des Algorithmus mache man sich klar, daß im wesentlichen jeder Knoten und seine Nachfolger höchstens zweimal betrachtet werden. Damit ist offensichtlich, daß die Laufzeit insgesamt linear in der Anzahl der Knoten ist.

Als für uns letzte wesentliche Operation in Zusammenhang mit OBDDs soll hier die Synthese besprochen werden, deren Beschreibung auf Bryant (1986) zurückgeht. Man darf sich merken, daß die Synthese im wesentlichen einem Paralleldurchlauf durch beide πOBDDs entspricht[12]. Dabei ist wesentlich, daß beide OBDDs der gleichen Variablenordnung π genügen. Wenn beim Paralleldurchlauf in beiden Graphen Senken erreicht sind, wird der anzuwendende Synthese-Operator herangezogen, um die passende Ergebnissenke zu berechnen. Sonst bestimmt die Knotenmarkierung, die gemäß Variablenordnung zuerst kommt, die Markierung des Knotens im Ergebnis-OBDD. Sind die Markierungen in beiden Knoten gleich, fahren wir rekursiv mit den 0- und 1-Nachfolgern fort. Sonst „warten" wir im tiefer gelegenen Knoten (tiefer in der Leveldarstellung gemäß Variablenordnung) und fahren nur mit dem anderen Knoten bei den Nachfolgern fort. Das Ergebnis-OBDD ergibt sich also in gewisser Weise als Produktgraph der Eingangs-OBDDs. Der Platzbedarf beträgt zunächst einmal $O(|G| \cdot |H|)$; das gleiche gilt für die Rechenzeit. Das muß im worst case auch so sein; es gibt Fälle, in denen das Ergebnis ebenso groß ist. Im Gegensatz zu FBDDs kann es hier nicht zu einem exponentiellen blow-up kommen; das stellt die gemeinsame Variablenordnung sicher. Um den Platzbedarf und den Zeitbedarf zu senken, kann man vermeiden, nicht erreichbare Knoten zu erzeugen. Dazu muß man erzeugte Knoten so abspeichern, daß sie bei Bedarf auch wieder gefunden werden können. Benutzt man ein Array, wird zwar kein Platz gespart, die Rechenzeit sinkt aber auf die Größe des Ergebnisgraphen (bis auf konstante Faktoren), wenn man die Zeit für die Arrayinitialisierung außer Acht läßt, was für konkrete Implementierungen berechtigt sein mag. Benutzt man einen AVL-Baum (oder etwas in der Zeit und im Platz vergleichbares), sinkt der Platzbedarf, es steigt aber die Rechenzeit (wegen der Suchzeiten im Dictionary). In der Praxis empfiehlt sich die Verwendung von Hashing oder einer Variante davon, die bei Kollisionen das Überschreiben vorhandener Hashtabelleneinträge vorsieht. Es ist auch möglich, die Reduktion gleich in die Synthese zu integrieren. Damit kann der Speicherplatzbedarf gesenkt werden, wenn Hashing verwendet wird (oder der reduzierte Graph *viel* kleiner ist). Die Rechenzeit sinkt aber nicht weiter im Vergleich zu der Version, welche die Konstruktion nicht erreichbarer Knoten vermeidet, da weiterhin alle erreichbaren Knoten betrachtet werden.

Wir wollen an dieser Stelle schon auf eine Besonderheit von Algorithmen, die auf BDDs operieren, hinweisen, die sie von Algorithmen vieler anderer Problemfelder unterscheidet. Es kommt bei BDD-Algorithmen häufig vor, daß die Größe der Ausgabe die Größe der Eingabe deutlich übersteigt. Bei der OBDD-Synthese ist etwa die Ausgabegröße $O(|G| \cdot |H|)$, obwohl die Eingabegröße nur $\Theta(|G| + |H|)$ beträgt. Es ist klar, daß damit jeder Algorithmus sowohl in der Laufzeit als auch im Platzbedarf in der Größenordnung nicht unter $|G| \cdot |H|$ liegen kann. Man sollte also bei BDD-Algorithmen die Laufzeit und das Platzverhalten in der Größe von Eingabe *und* Ausgabe messen.

[12] An dieser Stelle kann es sich lohnen, auf die Verwandtschaft zwischen OBDDs und endlichen Automaten hinzuweisen. Als Eingabe ist die Belegung der Variablen mit 0 oder 1 zu interpretieren, die Reihenfolge der Eingabe ist durch die Permutation π bestimmt. Im Gegensatz zu endlichen Automaten kann ein OBDD Teile der Eingabe überspringen (wenn kein so markierter Knoten auf dem Berechnungspfad vorhanden ist). Wie wir gesehen haben, können OBDDs ähnlich wie endliche Automaten effizient minimiert werden. Auch die Syntheseoperation weist große Ähnlichkeit auf.

2.2 Graphgesteuerte BDDs

Graphgesteuerte BDDs sind eine Verallgemeinerung von OBDDs, die von Sieling und Wegener (1995) eingeführt wurde[13]. Anstelle der Variablenordnung π tritt ein Orakelgraph G^O, dessen Gestalt zunächst formal definiert werden soll.

Definition 2.6: Ein *Orakelgraph* über den Variablen $x_1, \ldots, x_n$ ist ein gerichteter, azyklischer Graph mit genau einer Quelle und genau einer Senke, die mit „end" markiert ist. Jeder Knoten außer der Senke besitzt zwei ausgehende Kanten, von denen eine mit 0 und die andere mit 1 markiert ist. Außerdem ist jeder Knoten außer der Senke mit einem Variablennamen markiert. Auf jedem Weg von der Quelle zur Senke kommt jeder Variablenname genau einmal vor.

Definition 2.7: Syntax: Ein *graphgesteuertes BDD* oder *LBDD* (loosely structured graph driven binary decision diagram) ist ein FBDD G, dem ein Orakelgraph G^O zugeordnet ist mit folgender Eigenschaft: Auf jedem gerichteten Pfad von der Quelle zu einer Senke in G wird die durch G^O gegebene Variablenordnung respektiert: zu einer Belegung $a \in \{0,1\}^n$ der Variablen wird in G die Variablenfolge $v_G(a)$ und in G^O die Variablenfolge $v_{G^O}(a)$ durchlaufen. Wenn zwei Variablennamen x_i und x_j in beiden Folgen auftauchen, so kommen sie in derselben Reihenfolge vor.

Semantik: Die Auswertungsregel für graphgesteuerte BDDs ist mit der für FBDDs und OBDDs identisch.

Eine Teilmenge der Menge graphgesteuerter BDDs, deren Elemente eine stärker ausgeprägte Struktur besitzen und dadurch zum Teil einfachere und effizientere Algorithmen erlauben, wird noch gesondert bezeichnet.

Definition 2.8: Ein G^O-gesteuertes *WBDD* G (well structured binary decision diagram) ist ein LBDD, zu dem es eine Funktion $\alpha : V(G) \to V(G^O)$ gibt, so daß gilt: Für alle Knoten $v \in V(G)$ sind der Knoten v und sein Bild $\alpha(v) \in V(G^O)$ mit der gleichen Variablen markiert; für alle Eingaben $a \in \{0,1\}^n$, für die bei der Auswertung in G der Knoten v durchlaufen wird, wird in G^O der Knoten $\alpha(v)$ durchlaufen.

Die Auswertung kann für graphgesteuerte BDDs genau wie bei FBDDs und OBDDs durchgeführt werden, da die Semantik identisch ist und die Variablenordnung durch die Auswertung nicht berührt wird. Genau wie bei FBDDs und OBDDs ist der Erfüllbarkeitstest wiederum äquivalent zur Frage, ob es einen Pfad von der Quelle zu einer 1-Senke gibt und somit in Linearzeit durchführbar. Für die Betrachtung der restlichen Operationen ist es sinnvoll, sich zunächst der Minimierung zuzuwenden und insbesondere zu klären, ob es zu einer Booleschen Funktion f eine eindeutige Repräsentation minimaler Größe gibt.

Weil die Semantik von graphgesteuerten BDDs und OBDDs identisch ist, kann man die beiden Reduktionsregeln, die eine OBDD-Reduktion ermöglichen, auch auf graphgesteuerte BDDs anwenden, ohne die dargestellte Funktion zu verändern. Bevor wir uns diesem

[13] Die Darstellung des gesamten Abschnitts 2.2 folgt im wesentlichen Sieling (1995).

Gedanken näher zuwenden, wollen wir noch diskutieren, ob man einen gegebenen Orakelgraphen verkleinern kann, ohne daß dazu existierende LBDDs oder WBDDs verändert werden müßten, damit sich repräsentierte Funktionen nicht ändern.

Zu diesem Zweck betrachten wir, was bei Anwendung der OBDD-Reduktionsregeln auf einen Orakelgraph geschehen kann. Es ist ganz offensichtlich, daß die Eliminationsregel nicht angewendet werden darf: andernfalls gibt es im Widerspruch zu den in Definition 2.6 definierten Eigenschaften mindestens einen Pfad im Orakelgraphen, auf dem eine Variable nicht vorkommt. Die Verschmelzungsregel hingegen kann bedenkenlos angewendet werden, da sie nur isomorphe Subgraphen verschmilzt. Im ursprünglichen Orakelgraphen wird folglich auf allen Pfaden die gleiche Variablenordnung beschrieben wie in seiner durch Anwendung der Verschmelzungsregel entstandenen Modifikation.

Man kann zur Reduktion von Orakelgraphen eine modifizierte Version des OBDD-Reduktionsalgorithmus 2.5 verwenden, die auf die Elimination von Knoten verzichtet. Die Reduktion von Orakelgraphen ist also ebenfalls in Linearzeit möglich.

Wir kommen jetzt zur Reduktion von LBDDs und WBDDs. Wie bereits erwähnt können die OBDD-Reduktionsregeln auch auf LBDDs und WBDDs angewendet werden, ohne die dargestellte Funktion zu verändern, da graphgesteuerte BDDs und OBDDs die gleiche Semantik haben. Weil die Anwendung der Reduktionsregeln eine Variablenordnung intakt läßt, kann es bei LBDDs auch nicht zu Schwierigkeiten kommen. Bei WBDDs hingegen kann die Verschmelzungsregel nicht immer angewendet werden. Wenn an zwei Knoten v und w die gleiche Funktion repräsentiert wird, so dürfen sie nur verschmolzen werden, wenn dadurch die Funktion α nicht beschädigt wird. Es muß also $\alpha(v) = \alpha(w)$ gelten.

Der Umstand, daß die OBDD-Reduktionsregeln auf graphgesteuerte BDDs angewendet werden dürfen, sagt aber noch nichts darüber aus, ob LBDDs und WBDDs minimaler Größe zu einer Funktion f überhaupt eindeutig bestimmt sind. Selbst wenn das der Fall ist, bleibt unklar, ob die Anwendung der OBDD-Reduktionsregeln zur Reduktion ausreicht. Weil wir verschiedene Transformationsalgorithmen betrachten werden, die teilweise Reduktion als Unterprozedur einsetzen, werden wir uns diesen Problemen hier für WBDDs stellen und die einschlägigen Aussagen mit Beweisen aus Sieling (1995) wiedergeben. Für LBDDs werden wir uns darauf beschränken, die entsprechenden Aussagen ohne Beweis zu zitieren. Wir beginnen unsere Betrachtungen mit einigen Definitionen, die den Umgang mit graphgesteuerten BDDs erleichtern sollen.

Definition 2.9: Zu einem Orakelgraphen G^O und einem Knoten v aus G^O definieren wir den *Schnürknoten* $S(v)$ als den ersten Nachfolger von v, durch den alle gerichteten Pfade, die an v beginnen, laufen. Die Menge der Variablenmarkierungen, die zwischen dem Knoten v und $S(v)$ auftauchen (ohne die Markierung von $S(v)$ selbst) nennen wir $Var(v)$.

Man bemerke, daß es zu allen mit einer Variablen markierten Knoten aus einem Orakelgraphen stets einen Schnürknoten $S(v)$ gibt. Die mit „end“ markierte Senke ist jedenfalls ein Knoten, durch den alle Pfade gehen. Zur Definition von $Var(v)$ ist zu bemerken, daß auf jedem Pfad von v zu $S(v)$ alle Variablen aus $Var(v)$ auftauchen gemäß Definition 2.6.

Definition 2.10: Sei G ein gerichteter, azyklischer Graph; sei v ein Knoten in G. Wir nennen den Graphen $G(v)$, der den Knoten v und alle auf gerichteten Pfaden von v aus erreichbaren Knoten enthält, den *Subgraphen* von v.

Der Begriff des Subgraphen ist im Zusammenhang mit BDD-Varianten häufig nützlich. Es ist meist möglich, zu einem BDD G und einem Knoten v aus G den Subgraphen $G(v)$ wiederum als BDD für eine andere Funktion anzusehen.

Satz 2.11: [14] Sei $f : \{0,1\}^n \to \{0,1\}$ eine Boolesche Funktion und G^O ein Orakelgraph passend zu f. Es gibt ein bis auf Isomorphie eindeutiges WBDD G zu f und G^O minimaler Größe; G kann durch Eigenschaften von G^O und f beschrieben werden.

Beweis: Für konstante Funktionen f ist die Aussage trivial. Sei also f nicht konstant. Im ersten Beweisschritt konstruieren wir G und zeigen, daß G die Funktion f berechnet. Weil f nicht konstant ist, enthält G jedenfalls genau eine mit 0 und eine mit 1 markierte Senke. Wir wollen zu einem Knoten v mit Markierung x_i des Orakelgraphen die Menge $V(v)$ beschreiben, die alle x_i-Knoten w aus G enthält, für die $\alpha(w) = v$ gilt. Wir nehmen o. B. d. A. an, daß auf einem Pfad von der Quelle zu v die Variablen $x_1, \ldots, x_i$ getestet werden. Gemäß Definition 2.6 gilt das dann für alle Pfade von der Quelle bis zum Knoten v. Wir nennen die Menge aller Subfunktionen, die an Knoten in $V(v)$ berechnet werden können $A(v)$. Bei $A(v)$ handelt es sich um genau die Menge der Funktionen $f_{|x_1=a_1,\ldots,x_{i-1}=a_{i-1}}$, für die die partielle Eingabe $(a_1, \ldots, a_{i-1})$ zum Knoten v führt. Wir partitionieren $A(v)$ in die disjunkten Teilmengen $B(v)$ und $C(v)$. Dabei enthält $B(v)$ alle die Funktionen aus $A(v)$, die essentiell von x_i abhängen, alle anderen Funktionen gehören zu $C(v)$. Analog zu unseren Betrachtungen bei OBDDs ist auch hier offensichtlich, daß es für jede Funktion aus $B(v)$ einen x_i-Knoten geben muß. Anders als bei OBDDs können aber auch x_i-Knoten für Funktionen aus $C(v)$ erforderlich sein, um die durch G^O beschriebene Variablenordnung einzuhalten. Wir verwenden für alle die Funktionen aus $C(v)$ einen eigenen Knoten, die von einer Variablen aus $Var(v)$ essentiell abhängen. Damit ist die Knotenmenge von G beschrieben. Wir legen die Kanten in G wie folgt fest. Sei w ein Knoten aus $V(v)$ mit Markierung x_i; für w gilt $\alpha(w) = v$. Wir haben diesem Knoten w eine Funktion g aus $A(v)$ eindeutig zugeordnet. Wenn die Funktion $g_c := g_{|x_i=c}$ in G an einem Knoten w' dargestellt wird, für den $\alpha(w') = v'$ der c-Nachfolger von $\alpha(w)$ ist, wählen wir als c-Nachfolger von w diesen Knoten w'. Wenn es einen solchen Knoten aber nicht gibt, so hängt g_c nicht essentiell von einer Variablen aus $Var(v')$ ab; sonst entsteht direkt ein Widerspruch zur Definition von α. Darum können wir im Orakelgraphen von v' zu $S(v')$ gehen. Wir iterieren diesen Vorgang, bis wir entweder einen Knoten w' finden oder wir in G^O an der Senke ankommen. Dann ist g_c eine konstante Funktion, c-Nachfolger von v wird die entsprechende Senke.

Offensichtlich ist G ein G^O-gesteuertes WBDD. Den Nachweis, daß G die Funktion f repräsentiert, führt man leicht bottom-up mittels vollständiger Induktion. Die Senken repräsentieren offenbar die ihnen zugewiesenen Funktionen. Betrachten wir jetzt einen

[14] Satz und Beweis stammen wie gesagt aus Sieling (1995).

Knoten v mit Markierung x_i. Für seine beiden Nachfolger gilt per Induktionsannahme, daß sie die Funktionen $g_{|x_i=0}$ und $g_{|x_i=1}$ repräsentieren. Daraus folgt direkt die Behauptung.

Jetzt beginnen wir mit dem zweiten Beweisschritt, in dem wir zeigen, daß die Größe von G minimal ist. Wir führen einen Widerspruchsbeweis und nehmen an, daß H ein G^O-gesteuertes WBDD für f mit echt weniger Knoten als G ist. Dann gibt es einen Knoten v im Orakelgraphen, für den $\alpha_H^{-1}(v)$ weniger Knoten enthält als $\alpha_G^{-1}(v)$. Offensichtlich kann kein Knoten fehlen, der einer Funktion aus $B(v)$ zugeordnet ist. Sei nun g eine Funktion aus $C(v)$, für die G einen Knoten enthält und H nicht. Gemäß Definition von $C(v)$ hängt g essentiell von einer Variablen aus $Var(v)$ ab aber nicht von x_i; sei x_j die erste Variable, die auf einem Pfad von v nach $S(v)$ erreicht wird und von der g essentiell abhängt. Es gibt in $G^O(v)$ mindestens zwei Knoten mit Markierung x_j, weil $x_j \in Var(v)$ gilt. Seien z_1 und z_2 diese x_j-Knoten in G^O. Weil die Funktion g essentiell von x_j abhängt, gibt es in G einen x_j-Knoten w, an dem g dargestellt wird. Offenbar gibt es einen Knoten $z \in G^O(v)$, für den $\alpha(w) = z$ gilt. O. B. d. A. sei z_1 dieser Knoten. Wir wählen zwei partielle Eingaben a_1 und a_2, die in G^O zu z_1 und z_2 führen. Weil wir x_j so gewählt haben, daß es für einen gerichteten Pfad von v nach $S(v)$ die erste Variable ist, von der g essentiell abhängt, wird sowohl für a_1 als auch für a_2 in H der Knoten w erreicht. Also ist α_H nicht wohldefiniert; H ist im Widerspruch zur Annahme kein WBDD. Somit ist G das minimale G^O-gesteuerte WBDD für f.

Weil wir bei der Konstruktion von G keine Freiheiten hatten, ist G bis auf Isomorphie eindeutig. □

Nachdem wir nachgewiesen haben, daß WBDDs minimaler Größe eine eindeutige Darstellung für Boolesche Funktionen darstellen, wollen wir einen geeigneten Reduktionsalgorithmus finden. Dazu weisen wir nach, daß die Anwendung der OBDD-Eliminationsregel und die Anwendung der OBDD-Verschmelzungsregel auf Knoten v, w mit $\alpha(v) = \alpha(w)$ zur WBDD-Reduktion ausreicht. Wir nennen die OBDD-Eliminationsregel und die eingeschränkte Version der Verschmelzungsregel zusammen kurz WBDD-Reduktionsregeln.

Satz 2.12: [15] Sei G ein G^O-gesteuertes WBDD für f, in dem alle Knoten erreichbar sind und keine der WBDD-Reduktionsregeln anwendbar ist. Dann ist G isomorph zum minimalen G^O-gesteuerten WBDD für f.

Beweis: Für konstante Funktionen f ist die Aussage trivial, sei also f im folgenden nicht konstant. Wir betrachten die Knoten von G^O bottom-up (also in umgekehrt topologischer Reihenfolge) und beweisen mittels Induktion, daß die Knoten in $\alpha^{-1}(v)$ aus G genau zum minimalen WBDD gehören und G isomorph zum minimalen WBDD ist. Für die beiden Senken ist das offensichtlich, damit ist der Induktionsanfang gegeben. Wir betrachten einen Knoten v aus G^O; für alle Knoten in $G^O(v)$ außer v selbst können wir die Aussage per Induktionsannahme als bewiesen annehmen. Sei w ein x_i-Knoten aus $\alpha^{-1}(v)$, der zu G aber nicht zum minimalen G^O-gesteuerten WBDD für f gehört. Sei g die an w dargestellte Funktion. Wenn die Funktion g im minimalen WBDD gar nicht dargestellt wird, werden

[15] Auch hier stammen Satz und Beweis von Sieling (1995).

die Funktionen $g_{|x_i=0}$ und $g_{|x_i=1}$ im minimalen WBDD an *einem* Knoten dargestellt. Per Induktionsannahme gilt das auch für G, also kann w im Widerspruch zur Voraussetzung mit der Eliminationsregel entfernt werden. Wenn die Funktion g im minimalen WBDD auch dargestellt wird, so muß es zwei Knoten in G geben, an denen g dargestellt wird. Dann können diese zwei Knoten verschmolzen werden, da sie beide mittels α auf v abgebildet werden und W unterhalb von w isomorph zum minimalen G^O-gesteuerten WBDD ist. Es entsteht also in beiden Fällen ein Widerspruch zur Voraussetzung. □

Wir werden einen effizienten Reduktionsalgorithmus für WBDDs angeben, der in weiten Teilen dem Beweis oben folgt. Es werden zunächst alle nicht von der Quelle aus erreichbaren Knoten entfernt. Dann wird für alle Knoten v des Orakelgraphen $\alpha^{-1}(v)$ berechnet. Danach erfolgt die Anwendung der Reduktionsregeln, dabei werden die Knoten v aus G^O in umgekehrt topologischer Reihenfolge durchlaufen; in dieser Reihenfolge werden die Reduktionsregeln auf die Knoten in $\alpha^{-1}(v)$ angewendet. Die Anwendung kann analog zur OBDD-Reduktion in Linearzeit durchgeführt werden, wenn die Knotenmenge $\alpha^{-1}(v)$ zur Verfügung steht.

Wir brauchen hier also einen Algorithmus, der die effiziente Berechnung der Abbildung α erlaubt. Wir werden einen Algorithmus angeben, der das erledigt, wenn G tatsächlich G^O-gesteuert ist; andernfalls gibt der Algorithmus eine Fehlermeldung aus.

Um einen effizienten Algorithmus zu konstruieren, werden wir eine Hilfsdatenstruktur einführen, die auch für die Synthese von Nutzen ist. Dort hilft sie festzustellen, welcher der zwei Knoten $\alpha(v)$, $\alpha(w)$ im Orakelgraphen G^O für zwei Knoten v und w des G^O-gesteuerten BDD der Vorgänger des anderen ist.

Wir definieren zu einem Knoten v des Orakelgraphen und einer Variablen x_i die Menge $V(v, x_i)$ als Menge aller Knoten aus $G^O(v)$ mit Markierung x_i. Wir interessieren uns nur dafür, ob $V(v, x_i)$ leer ist, genau einen Knoten oder mehr als einen Knoten enthält. Dementsprechend definieren wir ein Array `succ` der Größe $n\left|G^O\right|$ mit

$$\texttt{succ}[\texttt{v}, \texttt{x}_\texttt{i}] := \begin{cases} 0 & \text{für } V(v, x_i) = \emptyset. \\ w & \text{für } V(v, x_i) = \{w\}. \\ 2 & \text{für } |V(v, x_i)| \geq 2. \end{cases}$$

Zur Berechnung der Einträge von `succ` durchlaufen wir G^O in umgekehrt topologischer Reihenfolge. Für den Knoten mit Marke „end“ ist $\texttt{succ}[\texttt{v}, \texttt{x}_\texttt{i}] = 0$ für alle x_i. Für alle anderen Knoten kann vorausgesetzt werden, daß die Werte der Nachfolger bereits berechnet sind. Sei v der betrachtete Knoten, seien v_0 und v_1 seine direkten Nachfolger. Es sind vier Fälle zu unterscheiden.

1. Es ist $\texttt{succ}[\texttt{v}, \texttt{x}_\texttt{i}] = \{v\}$ genau dann, wenn v mit x_i markiert ist.
2. Es ist $\texttt{succ}[\texttt{v}, \texttt{x}_\texttt{i}] = \{w\}$ genau dann, wenn $\texttt{succ}[\texttt{v}_0, \texttt{x}_\texttt{i}] = \{w\}$ und $\texttt{succ}[\texttt{v}_1, \texttt{x}_\texttt{i}] = \{w\}$ gelten.
3. Es ist $\texttt{succ}[\texttt{v}, \texttt{x}_\texttt{i}] = 0$ genau dann, wenn das auch für die Nachfolger gilt und v nicht mit x_i markiert ist.

4. In allen anderen Fällen ist $\texttt{succ}[\texttt{v}, \texttt{x}_\texttt{i}] = 2$.

Offenbar kann `succ` in Zeit $O\left(n\left|G^O\right|\right)$ auf Platz $O\left(n\left|G^O\right|\right)$ berechnet werden.

Wir können jetzt den Algorithmus zur Berechnung von α beschreiben. Zunächst berechnen wir das Array `succ`[16] zum Orakelgraphen G^O. Wir durchlaufen den Graphen G mit einer Tiefensuche. Dabei erreichen wir jeden Knoten über jede eingehende Kante genau einmal. Erreichen wir einen Knoten v mit Markierung x_i von einem Knoten v' aus, so ist $\alpha(v')$ schon berechnet. Einzige Ausnahme ist die Quelle von G, die keinen Vorgänger hat. Für die Quelle v finden wir $\alpha(v)$ als $\texttt{succ}[\texttt{w}, \texttt{x}_\texttt{i}]$, wenn w die Quelle des Orakelgraphen ist. Für alle anderen Knoten finden wir $\alpha(v)$ als $\texttt{succ}[\alpha(\texttt{v}'), \texttt{x}_\texttt{i}]$. Bezeichnet $\texttt{succ}[\alpha(\texttt{v}'), \texttt{x}_\texttt{i}]$ keinen Knoten, so ist G nicht G^O-gesteuert. Der Algorithmus hat Laufzeit $O\left(n\left|G^O\right| + |G|\right)$ und braucht Speicherplatz in dergleichen Größenordnung. Zur Berechnung der für die Reduktion benötigten Knotenlisten $\alpha^{-1}(v)$ genügt ein Durchlauf durch G.

Wir merken an, daß auch LBDDs minimaler Größe eine eindeutige Darstellung Boolescher Funktionen sind; zur Reduktion reicht die Anwendung der OBDD-Reduktionsregeln. Die Reduktion von LBDDs ist in Linearzeit auf linearem Speicherplatz möglich. Ein ausführlicher Beweis kann bei Sieling (1995) gefunden werden.

Weil jede Boolesche Funktion eine eindeutige Darstellung als reduziertes G^O-gesteuertes WBDD (LBDD) hat (zu einem festen Orakelgraphen G^O), ist der Äquivalenztest einfach als Isomorphietest der reduzierten Graphen zu implementieren. Er kann also in Linearzeit durchgeführt werden.

Wir kommen jetzt zur wichtigen Operation Synthese, die wir im Detail nur für LBDDs betrachten. Eingabe ist ein Orakelgraph G^O, ein G^O-gesteuertes LBDD F für die Funktion f, ein G^O-gesteuertes LBDD G für die Funktion g und ein Boolescher Operator $\otimes$. Gesucht ist ein G^O-gesteuertes LBDD für die Funktion $f \otimes g$. Die Beschreibung hier geht auf Sieling (1995) zurück.

Im Prinzip kann die Synthese wie bei OBDDs durchgeführt werden, es muß allerdings der verallgemeinerten Ordnung Rechnung getragen werden. Wenn man möchte, kann man den im folgenden vorgestellten Synthesealgorithmus als Verallgemeinerung der OBDD-Synthese betrachten. Schließlich läßt sich die bei OBDDs gegebene Variablenordnung π leicht in Graphform für graphgesteuerte BDDs notieren; der Orakelgraph besteht dann im Kern nur aus einer Knotenkette aus genau $n+1$ Knoten, wobei die Variablenmarkierungen von der Quelle bis zur Senke gerade in der durch π beschriebenen Reihenfolge auftauchen.

Der Synthesealgorithmus entspricht im Prinzip einem Paralleldurchlauf durch alle drei Graphen. Wir beginnen an den Quellen von G^O, F und G und konstruieren das Ergebnis-LBDD in DFS-Reihenfolge. Dieses Vorgehen vermeidet, nicht erreichbare Knoten zu erzeugen. Sei (u, v, w) das aktuelle Knoten-Tripel. Die 0-Nachfolger der Knoten u, v und w bezeichnen wir mit u_0, v_0 und w_0, analog nennen wir die 1-Nachfolger u_1, v_1 und w_1.

[16] Es gibt eine Reihe von Algorithmen, die von der Hilfsdatenstruktur `succ` profitieren. Darum macht es in realen Anwendungen Sinn, zu einem Orakelgraphen das Array `succ` nur einmal zu berechnen und stets im Speicher zu halten. Der Speicherbedarf steigt nur mäßig an, die Laufzeit vieler Algorithmen sinkt aber spürbar. Der WBDD-Reduktionsalgorithmus etwa bekommt dann Laufzeit $O(|G|)$.

Weil wir im Orakelgraphen niemals warten müssen, da dort keine Variable ausgelassen sein kann, bestimmt u stets die Markierung des neu zu erzeugenden Knotens. Ist u eine Senke, so sind v und w notwendig auch Senken (ist das nicht der Fall, ist wenigstens eines der beiden LBDDs F und G nicht G^O-gesteuert). Dann ist das Ergebnis die Senke, die sich durch Verknüpfung der Markierungen von v und w mit dem gegebenen Operator $\otimes$ ergibt. Wenn nur in F oder G eine Senke erreicht wird, kann schon dann eine Senke als Ergebnis bestimmt werden, wenn der Wert an der Senke den Funktionswert dominiert. Sonst erhält der neue Knoten die Markierung von u, und die Nachfolger bestimmen wir wie folgt:

1. Fall: u, v und w sind alle mit x_i markiert. Dann wird der 0-Nachfolger durch das Tripel (u_0, v_0, w_0) und der 1-Nachfolger durch das Tripel (u_1, v_1, w_1) bestimmt.

2. Fall: u und v sind mit x_i markiert, w trägt eine andere Marke x_j oder ist eine Senke. Dann wird der 0-Nachfolger durch das Tripel (u_0, v_0, w) und der 1-Nachfolger durch das Tripel (u_1, v_1, w) bestimmt.

3. Fall: u und w sind mit x_i markiert, v trägt eine andere Marke x_j oder ist eine Senke. Dann wird der 0-Nachfolger durch das Tripel (u_0, v, w_0) und der 1-Nachfolger durch das Tripel (u_1, v, w_1) bestimmt.

4. Fall: u ist mit x_i markiert, v und w tragen andere Marken x_j und x_k oder sind Senken. Dann wird der 0-Nachfolger durch das Tripel (u_0, v, w) und der 1-Nachfolger durch das Tripel (u_1, v, w) bestimmt.

Der Algorithmus konstruiert auf Platz $O\left(\left|G^O\right| |F| |G|\right)$ in Zeit $O\left(\left|G^O\right| |F| |G|\right)$ ein G^O-gesteuertes LBDD für $f \otimes g$, das im allgemeinen nicht reduziert ist. Man kann, um Speicherplatz zu sparen, ähnlich wie bei OBDDs die Reduktion in die Synthese integrieren. Auch hier ergibt sich ein trade-off zwischen Speicherplatz und Rechenzeit.

Wir erwähnen, daß die WBDD-Synthese in Zeit $O(|F| |G|)$ mit Platz $O\left(n \left|G^O\right| + |F| |G|\right)$ möglich ist, wenn die Hilfsdatenstruktur `succ` gegeben ist. Diese Hilfsdatenstruktur erlaubt es, in konstanter Zeit für einen Knoten zu entscheiden, mit welcher Variablen er markiert werden muß.

Für die Ersetzung von Variablen durch Konstante und den Redundanztest sind keine effizienten Algorithmen für LBDDs und WBDDs bekannt. Man kann zeigen, daß beide Probleme für LBDDs und WBDDs bezüglich polynomieller Reduktion genauso schwierig sind wie der Äquivalenztest für FBDDs[17].

Abschließend bemerken wir noch, daß viele Algorithmen effizienter und einige Probleme überhaupt erst effizient lösbar werden, wenn der Orakelgraph eine bestimmte Gestalt hat. Wir werden auf solche Spezialfälle zurückkommen, wenn wir uns im Abschnitt 3.3.4 mit der Transformation von graphgesteuerten BDDs beschäftigen.

[17] Entsprechende Aussagen und Beweise findet man bei Sieling (1995).

2.3 Zero-suppressed Binary Decision Diagrams (ZBDDs)

In der Praxis kommt es häufig vor, daß bei den verwendeten Booleschen Funktionen wesentlich mehr Eingabevektoren a den Funktionswert 0 als den Funktionswert 1 ergeben. Dieser Umstand rechtfertigt es, eine BDD-Variante zu definieren, in der die Booleschen Konstanten 0 und 1 verschiedene Rollen spielen. Minato (1993) hat in diesem Sinne ZBDDs definiert[18].

Definition 2.13: Syntax: Ein *ZBDD* (zero-suppressed binary decision diagram) entspricht syntaktisch einem OBDD.

Semantik: Zu einem gegebenen ZBDD für die Funktion $f : \{0,1\}^n \to \{0,1\}$ und einer Belegung der Variablen $x_1, \ldots, x_n$ berechnet man den Funktionswert $f(x_1, \ldots, x_n)$ wie folgt: Man durchläuft wie bei der OBDD-Auswertung den Graphen in Abhängigkeit von der Variablenbelegung von der Quelle bis zu einer Senke, wobei man sich merkt, welche Variablen nicht getestet werden (also nicht als Markierungen auf dem Pfad vorkommen). Ist die schließlich erreichte Senke eine 0-Senke, so ist der Funktionswert 0. Ist die erreichte Senke eine 1-Senke, so ist der Funktionswert genau dann 1, wenn alle nicht getesteten Variablen mit 0 belegt sind, sonst ist der Funktionswert 0.

Als erstes wollen wir feststellen, welche Funktion ein gegebenes ZBDD darstellt. Offenbar stellt die 0-Senke die konstante Nullfunktion dar. Die 1-Senke stellt aber nicht etwa die konstante Einsfunktion dar! Der Funktionswert eines ZBDD über den Variablen $x_1, \ldots, x_n$, das nur aus der 1-Senke besteht, ist genau dann 1, wenn alle Variablen mit 0 belegt sind. Die 1-Senke stellt also die Funktion $\overline{x_1} \cdot \overline{x_2} \cdot \ldots \cdot \overline{x_n}$ dar. Nehmen wir an, es ist ein ZBDD mit Quelle u gegeben. Die Markierung von u sei x_i, der 0-Nachfolger sei der Knoten v, der 1-Nachfolger der Knoten w. Wir nehmen an, daß alle drei Knoten Funktionen über den Variablen $x_1, \ldots, x_n$ repräsentieren. Es repräsentiere u die Funktion f, v die Funktion g und w die Funktion h (vgl. Abbildung 4).

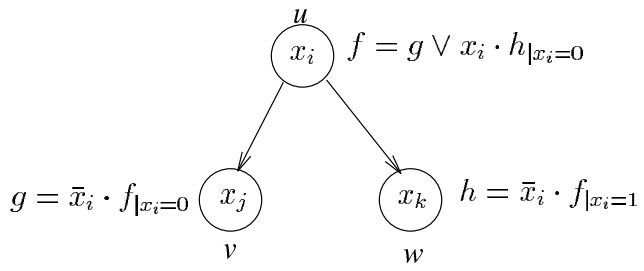


Abbildung 4: Funktionsrepräsentation im ZBDD

Wir nennen einen Pfad im ZBDD von der Quelle zu einer Senke einen Belegungspfad und vereinbaren, daß jeder Belegungspfad eine Belegung der Eingabevariablen definiert wie

[18]Die Darstellung hier basiert im wesentlichen auf Schröer und Wegener (1994). Minato selbst spricht von 0-Sup-BDDs.

folgt: Kommt auf dem Belegungspfad ein x_i-Knoten vor, der über die mit 1 markierte Kante verlassen wird, so ist $a_i := 1$, andernfalls ist $a_i := 0$.

Wir nennen die Menge der Variablenbelegungen, die zu Belegungspfaden im ZBDD gehören und in einer 1-Senke enden, A. Wir teilen die Menge A in zwei disjunkte Teilmengen A_0 und A_1; dabei enthält A_0 die Vektoren aus A, deren Belegungspfade über v führen und A_1 die Vektoren aus A, deren Belegungspfade über w führen. Offensichtlich kann die Menge A benutzt werden, um die Funktion f zu definieren, da A genau die Vektoren $a \in \{0,1\}^n$ enthält, für die $f(a) = 1$ gilt. Die Menge A_0 definiert auf diese Art und Weise die Funktion $\overline{x_i} \cdot f_{|x_i=0}$; die Menge A_1 definiert die Funktion $x_i \cdot f_{|x_i=1}$.

Wir betrachten jetzt zunächst das ZBDD mit Quelle v. Die Menge der Belegungspfade, die in einer 1-Senke enden, definiert eine Menge B_0. Offensichtlich ist $A_0 = B_0$, da alle Belegungspfade im ursprünglichen ZBDD genau eine 0-Kante mehr besitzen und 0-Kanten ebenso wie ausgelassene Variablen eine 0 in der Variablenbelegung implizieren. Also ist $g = \overline{x_i} \cdot f_{|x_i=0}$.

Jetzt betrachten wir das ZBDD mit Quelle w. Die Menge der Belegungspfade, die in einer 1-Senke enden, definiert eine Menge B_1. Offensichtlich ist $|A_1| = |B_1|$; zu jedem Vektor in A_1 gibt es ein Gegenstück in B_1, das sich lediglich an der i-ten Stelle unterscheidet. Es ist $a_i = 1$ für alle $a \in A_1$ und $b_i = 0$ für alle $b \in B_1$, da in allen Belegungspfaden zu Vektoren aus A_1 die 1-Kante der Quelle (einem x_i-Knoten) vorkommt, in allen Belegungspfaden zu Vektoren aus B_1 aber gar kein x_i-Knoten vorkommt. Also ist $h = \overline{x_i} \cdot f_{|x_i=1}$; die Darstellung in Abbildung 4 ist somit gerechtfertigt.

Nachdem wir die Struktur von ZBDDs geklärt haben, wollen wir kurz die Anwendung der wesentlichen Operationen auf ZBDDs besprechen. Außer der Reduktion werden aber keine Algorithmen vorgestellt, da sie sich für die Transformation als unnötig erweisen werden und deutlich komplizierter sind, als die entsprechenden Algorithmen für OBDDs. Eine vollständige Darstellung findet man bei Schröer und Wegener (1994); aus dieser Arbeit stammen auch Satz 2.14 und der zugehörige Beweis.

Um zu zeigen, daß auch ZBDDs eine eindeutige minimale Darstellung für Boolesche Funktionen sind und mit geeigneten Reduktionsregeln reduziert werden können, braucht man eine Aussage, die Satz 2.3 entspricht.

Satz 2.14: Für jede Funktion $f : \{0,1\}^n \to \{0,1\}$ und jede Variablenordnung π gibt es bis auf Isomorphie ein eindeutiges πZBDD G minimaler Größe.

Vor dem recht technischen Beweis werden wir einige Begriffe definieren, um ein geeignetes Instrumentarium zur Hand zu haben.

Definition 2.15: Die Funktion $f_i^{0\to 0}(x) := \overline{x_i} \cdot f_{|x_i=0}$ wird *Null-Unterfunktion* von f genannt.
Die Funktion $f_i^{1\to 0}(x) := \overline{x_i} \cdot f_{|x_i=1}$ wird *Eins-Unterfunktion* von f genannt.

In Abbildung 4 ist also g die Null-Unterfunktion von f und h die Eins-Unterfunktion von f.

Definition 2.16: Eine Funktion f heißt *0-redundant* bezüglich der Variablen x_i, wenn alle Vektoren a, für die $f(a) = 1$ gilt, an der i-ten Stelle eine 0 haben.
Ein Knoten v in einem ZBDD heißt 0-redundant bezüglich der Variablen x_i, wenn v mit x_i markiert ist und die an v repräsentierte Funktion 0-redundant ist.

Beweis von Satz 2.14: O. B. d. A. sei die Variablenordnung π die Identität. Wir wollen bestimmen, wie viele Knoten mit Markierung x_i $(i > 1)$ es in einem πZBDD für f geben muß. Für jeden Vektor $a \in f^{-1}(1)$ mit $a_i = 1$ gibt es einen Pfad von der Quelle zur 1-Senke, auf dem ein x_i-Knoten liegt. Ein Eingabevektor kann nicht durch mehrere Pfade vertreten sein, da er an jedem Knoten eindeutig entweder der 0-Kante oder der 1-Kante zugeordnet ist. Offenbar können verschiedene Funktionen nicht durch den gleichen Knoten dargestellt werden. Also muß es mindestens so viele x_i-Knoten geben, wie es verschiedene Unterfunktionen $f_{1,\ldots,i-1}^{a_1 \to 0,\ldots,a_{i-1} \to 0}$ gibt.

Diese Anzahl von x_i-Knoten genügt auch. Offenbar können gleiche Unterfunktionen durch gleiche Knoten dargestellt werden. Funktionen die 0-redundant sind, brauchen keine x_i-Knoten; x_i-Knoten für 0-redundante Funktionen haben Nachfolger, die die gleiche Funktion repräsentieren.

Für x_1 ist offensichtlich, daß das πZBDD genau dann einen x_1-Knoten besitzen muß, wenn f nicht 0-redundant bezüglich x_1 ist.

Jetzt ist gezeigt, daß die Knotenzahl eines minimalen πZBDD für f durch Bezug auf f und die Variablenordnung π beschrieben werden kann. Wir zeigen nun durch eine eindeutige Konstruktion, daß es bis auf Isomorphie nur genau ein minimales πZBDD für f und π gibt.

Sei i der kleinste Index, so daß f nicht 0-redundant bezüglich x_i ist. Wenn es kein solches i gibt, ist f entweder konstant 0 oder $\overline{x_1} \cdot \ldots \cdot \overline{x_n}$ und wird dementsprechend entweder durch die 0-Senke oder durch die 1-Senke repräsentiert. Andernfalls benutzen wir einen Knoten v mit Markierung x_i als Quelle des πZBDD und bestimmen seine Nachfolger. Wiederholte Anwendung dieser Regel führt dann zur Erzeugung des gesamten minimalen πZBDD.

0-Nachfolger von v ist

- die 0-Senke, falls $(f_i^{0 \to 0})^{-1}(1) = \emptyset$ gilt.
- die 1-Senke, falls $(f_i^{0 \to 0})^{-1}(1) = \{0^n\}$ gilt.
- ein Knoten, der die Funktion $f_{i,\ldots,j-1}^{a_i \to 0,\ldots,a_{j-1} \to 0}$ repräsentiert, so daß diese Funktion für $j > i$ nicht 0-redundant ist und j der minimale Index ist, der diese Eigenschaften hat.

1-Nachfolger von v ist

- die 1-Senke, falls $(f_i^{0\to 0})^{-1}(1) = \{0^n\}$ gilt.
- ein Knoten, der die Funktion $f_{i,\ldots,j-1}^{a_i\to 1, a_{i+1}\to 0,\ldots,a_{j-1}\to 0}$ repräsentiert, so daß diese Funktion für $j > i$ nicht 0-redundant ist und j der minimale Index ist, der diese Eigenschaften hat.

Weil die Konstruktion dem entspricht, was wir oben über die an ZBDD-Knoten repräsentierten Funktionen gesagt haben, stellt die Quelle des so konstruierten πZBDD tatsächlich die Funktion f dar. □

Wir suchen jetzt Reduktionsregeln, die ähnlich wie bei OBDDs eine (effiziente) Reduktion erlauben. Die Verschmelzungsregel kann offenbar unverändert beibehalten werden, da sie nur isomorphe Subgraphen verschmilzt. Die Eliminationsregel muß allerdings an die veränderte Semantik angepaßt werden. Wie wir bereits gesehen haben, ist es möglich, 0-redundante Knoten zu entfernen. Ein Knoten ist 0-redundant, wenn sein 1-Nachfolger die 0-Senke ist[19]. Es können also alle die Knoten entfernt werden, deren 1-Nachfolger die 0-Senke ist. Man kann zur Reduktion von ZBDDs eine leicht veränderte Variante von Algorithmus 2.5 verwenden, wobei die einschlägigen Reduktionsregeln wieder bottom-up angewendet werden. Wir geben hier den angepaßten Algorithmus wieder; die verwEndeten Datenstrukturen ändern sich nicht.

Algorithmus 2.17 (ZBDD-Reduktion):

```
/* Eingabe: πZBDD G=(V,E) für f (o.B.d.A. π=id) */
/* Ausgabe: reduziertes πZBDD für f */
1.   Erstelle Knotenlisten L_1,...,L_n,L_=0,L_=1
     für die Knoten mit Markierung x_1,...,x_n,0,1;
2.   For i:=1 to |V| do
          num[i]:=i; left[i]:=nil; right[i]:=0;
3.   L-Active:=nil;
4.   for i:=0 to 1 do
          j:=Head(L_=i);
          For v∈ L_=i do
               num[v]:=j;
     sink0:=num[Head(L_=0)];
```

[19] Die Umkehrung gilt offenbar nicht. Das entspricht der Situation bei OBDDs: Knoten mit zwei gleichen Nachfolgern sind redundant, d. h. sie können ohne Änderung der dargestellten Funktion entfernt werden. Das kann aber auch für andere Knoten gelten.

```
5.   For i:=n downto 1 do
6.        For v∈ L_i do
7.             If num[v→rechts]=sink0
               Then num[v]:=num[v→links];
               Else If left[num[v→links]]=nil
                    Then Insert(num[v→links],L-Active);
                    Insert(v,left[num[v→links]]);

8.        For v∈L-Active do
9.             For w∈Left[v] do
10.            If right[num[w→rechts]]=0
               Then right[num[w→rechts]]:=w;
               Else num[w]:=right[num[w→rechts]];
11.       For w∈Left[v] do
               right[num[w→rechts]]:=0;
12.       L-Active:=nil;
13.  For i:=1 to |V| do
          If num[v]≠v
          Then Eliminiere v;
          Else v→links:=num[v→links];
               v→rechts:=num[v→rechts];
```

Die Laufzeit des Algorithmus beträgt $O(|G| + n)$, ist also für alle praktisch relevanten Fälle linear in der Größe des ZBDD. Der Summand n wird durch die Initialisierung der leeren Listen hervorgerufen; es ist bei ZBDDs im Gegensatz zu OBDDs nicht ohne weiteres möglich anzunehmen, daß für alle $1 \leq i \leq n$ auch ein x_i-Knoten vorkommt. Auch wenn eine Funktion essentiell von x_i abhängt, muß deshalb noch kein x_i-Knoten vorkommen.

Wir werden noch an verschiedenen Stellen Gelegenheit haben zu überlegen, ob wir tatsächlich ohne jede Beschränkung der Allgemeinheit voraussetzen können, daß die beteiligten BDDs mindestens Größe n haben. Man sollte diese Problematik zumindest in praktischer Hinsicht nicht zu Ernst nehmen und sich folgendes ins Gedächtnis rufen: In praktischen Anwendungen sind BDDs typischerweise *viel* größer als n. Wenn die beteiligten BDDs so klein werden, daß der Summand n eine Rolle in der Rechenzeitabschätzung spielt, darf man für die Praxis getrost annehmen, daß der betroffene Algorithmus sehr effizient ist. Will man in Hinblick auf eine makellose Theorie lineare Laufzeit erreichen und spielt tatsächlich der Summand n in der Laufzeitabschätzung eine Rolle, so kann man auch getrost wie folgt argumentieren: Zu einem ZBDD gehört ebenso wie zu einem OBDD eine Variablenordnung π; verlangt man, daß die Variablenordnung Teil der Eingabe ist — eine nicht unberechtigte Forderung —, so ist eine Laufzeit von $O(n + |G|)$ linear in der Größe der Eingabe. Schwierigkeiten kann man bekommen, wenn man rekursive Algorithmen betrachtet, für die auf den unteren Rekursionsebenen praktisch immer Graphen mit sehr wenig Knoten betrachtet werden; in diesen Fällen sollte dem Summanden n Aufmerk-

samkeit gezollt werden. Wir werden in Abschnitt 3.2 dieses Problem antreffen und mit Sorgfalt lösen.

Nach diesem kurzen Ausflug fassen wir jetzt noch kurz die wesentlichen Fakten über Algorithmen für die restlichen Operationen zusammen. Die Synthese mit dem Syntheseoperator $\otimes$ kann nur effizient durchgeführt werden, wenn $0 \otimes 0 = 0$ gilt. Ist das nicht gegeben, muß die Synthese mit einem Operator $\overline{\otimes}$ durchgeführt werden, so daß $a \otimes b = \overline{\overline{a} \overline{\otimes} \overline{b}}$ gilt. Offenbar gilt dann $0 \overline{\otimes} 0 = 0$. Das gewünschte Ergebnis kann anschließend durch Negation erreicht werden. Die Laufzeit der Synthese hängt stark davon ab, welche Dictionary-Datenstruktur verwendet wird; die Wahl beeinflußt gleichzeitg den Platzbedarf. Laufzeit und Speicherplatz hängen ab von der Summe der Anzahl der betrachteten Knotenpaare und kopierten Knoten. Diese Summe kann im ungünstigsten Fall Größe $O(|G|\,|H|)$ haben, wenn $G \otimes H$ für ZBDDs G und H berechnet werden soll. Laufzeit und Speicherplatz haben die gleiche Größenordnung, wenn Zwischenergebnisse in Arrays gehalten werden. Verzichtet man ganz auf das Speichern von Teilergebnissen, kommt man mit minimalem Platz $O(|F| + n)$ aus, wenn F das reduzierte Ergebnis-ZBDD bezeichnet. Die Laufzeit beträgt in diesem Fall allerdings $O(2^n)$.

Die Ersetzung von Variablen durch Konstante kann wie bei OBDDs in Linearzeit durchgeführt werden. Es gibt auch in der Durchführung der Ersetzung einer Variablen x_i durch die Konstante c keinen Unterschied zwischen OBDDs und ZBDDs, wenn auf jedem Pfad von der Quelle zu einer Senke ein mit x_i markierter Knoten auftaucht oder $c = 1$ gilt. Für 0-Setzungen müssen also unter Umständen neue Knoten erzeugt werden, der Aufwand ist offensichtlich etwas größer. Weil sich die Größe eines ZBDD durch das Einfügen von x_i-Knoten allerdings höchstens verdoppeln kann (wenn schon vorher beide Senken vorhanden waren), bleibt der Aufwand in O-Notation linear in der Größe des Graphen.

Der Redundanztest ist ähnlich wie die Ersetzung von Variablen durch Konstante etwas komplizierter als bei OBDDs, kann aber ebenfalls in Linearzeit durchgeführt werden. Wir wollen hier kurz erwähnen, daß sich der Redundanztest leicht auf Ersetzung einer Variablen durch eine Konstante und Äquivalenztest zurückführen läßt. Die Variable x_i ist redundant, wenn die Darstellungen für $f_{|x_i=0}$ und $f_{|x_i=1}$ äquivalent sind.

2.4 Ordered Functional Decision Diagrams (OFDDs)

OBDDs basieren auf der Shannon-Zerlegung und haben sich in der Praxis für viele Funktionen bewährt. Es gibt jedoch viele Funktionen, für die keine Darstellung als OBDD gefunden wird, die klein genug ist, um praktisch handhabbar zu sein. Es gibt Fälle in der Praxis, wo mit Funktionen gearbeitet wird, die auf der Reed-Muller-Expansion (RME) basieren. Für solche Funktionen haben Kebschull, Schubert und Rosenstiel (1992) OFDDs eingeführt.

Die RME einer Funktion ist eine $\oplus$-Summe von Monomen aus unnegierten Literalen. Es gibt für jede Funktion genau eine Darstellung als RME: Die Existenz der Darstellung ist gesichert, weil man die RME aus der Darstellung in disjunktiver Normalform (DNF) erhält, indem man „$\vee$“ durch „$\oplus$“ ersetzt, negierte Literale $\overline{x_i}$ durch $x_i \oplus 1$ ersetzt, aus-

multipliziert und so weit wie möglich vereinfacht. Die Eindeutigkeit ergibt sich aus einem einfachen Abzählargument. Es gibt bekanntlich genau 2^{2^n} Boolesche Funktionen über n Variablen. Offensichtlich gibt es genau 2^n Monome (mit dem Monom 1), also gibt es genau 2^{2^n} RME-Darstellungen.

Definition 2.18: Syntax: Die Syntax eines *OFDD* (ordered functional decision diagram) ist identisch mit der Syntax eines OBDD.

Semantik: Ein OFDD, das aus genau einer Senke besteht, stellt die entsprechende konstante Funktion dar. Ein OFDD, das eine Quelle mit Marke x_i hat, stellt die Funktion $f = g \oplus x_i \cdot h$ dar, wenn der 0-Nachfolger der Quelle die Funktion g und der 1-Nachfolger die Funktion h darstellt.

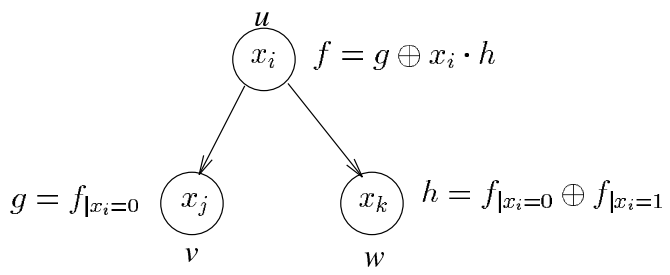


Abbildung 5: Funktionsrepräsentation im OFDD

Die Definition (und auch die Abbildung 5) zeigen, daß OFDDs auf der RME basieren:

$$f = f_{|x_i=0} \oplus x_i \cdot (f_{|x_i=0} \oplus f_{|x_i=1}).$$

Die Auswertung von OFDDs läßt sich am besten rekursiv beschreiben. Senken stellen die entsprechende Konstante dar, damit ist der Abbruch der Rekursion geklärt. Besteht das OFDD nicht nur aus einer Senke, so liegt im wesentlichen die Situation aus Abbildung 5 vor. Man bestimmt dann als den Funktionswert, der am Knoten u berechnet wird, den rekursiv berechneten Funktionswert vom Knoten v, wenn $x_i = 0$ gilt; gilt $x_i = 1$, so ergibt sich der Funktionswert am Knoten u als $\oplus$-Summe der rekursiv an den Knoten v und w berechneten Funktionswerte. Wenn man sicherstellt, daß keine Knoten mehrfach betrachtet werden, kann die Auswertung in Zeit $O(|G|)$ erledigt werden; man vergleiche das Ergebnis mit der bisher üblichen Zeit $O(n)$.

Wir benötigen auch hier eine strukturelle Aussage, um Aussagen über Eindeutigkeit der Darstellung und Reduktion machen zu können. Die Aussage findet sich in ähnlicher Form allerdings ohne Beweis bei Wegener (1994/95). Der Beweis hier ist in seiner Struktur den Beweisen zu den entsprechenden Aussagen 2.3 (für OBDDs), 2.11 (für WBDDs) und 2.14 (für ZBDDs) ähnlich.

Satz 2.19: Für jede Boolesche Funktion $f : \{0,1\}^n \to \{0,1\}$ und jede Variablenordnung π gibt es bis auf Isomorphie ein eindeutiges πOFDD G minimaler Größe, das explizit mit Eigenschaften von f und π beschrieben werden kann.

Beweis: Wir nehmen wie stets o. B. d. A. an, daß die Variablenordnung π die Identität ist. Für konstante Funktionen f ist die Aussage trivial. Wir betrachten nur Knoten mit Variablenmarkierungen; es ist offensichtlich daß es höchstens zwei Senken geben muß und jede Senke nur vorzukommen braucht, wenn sie von der Quelle aus erreichbar ist.

Wir wollen zeigen, daß G genau so viele x_{i+1}-Knoten enthält, wie es unterschiedliche Funktionen

$$f_{S_i} := \bigoplus_{1 \leq j \leq i : c_j = 0 \text{ für } j \notin S_i, c_j \in \{0,1\} \text{ für } j \in S_i} f_{|x_1 = c_1, \ldots, x_i = c_i}$$

für Teilmengen $S_i \subseteq \{1, \ldots, i\}$ gibt, die essentiell von x_{i+1} abhängen für $0 \leq i < n$. Wir zeigen zuerst mit Hilfe einer eindeutigen Konstruktion, daß man tatsächlich mit dieser Anzahl von Knoten auskommen kann. Im zweiten Schritt weisen wir nach, daß es kein πOFDD für f mit echt weniger Knoten geben kann.

Wir beschreiben top-down die Konstruktion eines πOFDD H für f. Die Konstruktion ist rekursiv durchzuführen, dabei ist darauf zu achten, daß für gleiche Funktionen nur ein repräsentierendes πOFDD konstruiert wird. Die Rekursion bricht ab, wenn ein πOFDD für eine konstante Funktion berechnet werden soll: πOFDDs für konstante Funktionen sind entsprechende Senken.

Hängt die Funktion f nicht essentiell von x_1 ab, so benutzen wir das rekursiv zu berechnende πOFDD für $f_{|x_1=0}$ als H. Andernfalls erzeugen wir einen x_1-Knoten, an dem f repräsentiert werden soll. 0-Nachfolger des Knotens wird ein πOFDD für $f_{|x_1=0}$, 1-Nachfolger des Knotens wird ein πOFDD für $f_{|x_1=0} \oplus f_{|x_1=1}$. Auf diese Art und Weise werden ausschließlich πOFDDs für Funktionen f_{S_i} erzeugt, die essentiell von x_{i+1} abhängen.

Wir nehmen jetzt im zweiten Schritt an, daß G' ein πOFDD für f mit echt weniger Knoten als H ist. Wir vergleichen die πOFDDs G' und H ebenenweise von oben nach unten; dabei können wir für jeden Knoten bestimmen, welche Funktion an ihm repräsentiert wird. Wird an einem x_i-Knoten in G' eine Funktion repräsentiert, die nicht essentiell von x_i abhängt, kann x_i aus G' entfernt werden; Kanten auf den x_i-Knoten werden durch Kanten auf seinen 0-Nachfolger ersetzt. Dadurch wird G' nur noch kleiner; wir dürfen also o. B. d. A. davon ausgehen, daß dieser Fall nicht vorkommt. Als direkte Folge ergibt sich, daß die Quellen gleiche Markierungen tragen: H hat als Markierung der Quelle das x_i mit i minimal, so daß f essentiell von x_i abhängt. G' kann gemäß Vereinbarung also an der Quelle als Markierung nur x_j mit $j \geq i$ haben. Ist $j > i$, so erzwingt die Variablenordnung π, daß in G' überhaupt kein x_i-Knoten vorkommt. Dann berechnet G' auch nicht f im Widerspruch zur Voraussetzung.

Beim Graphdurchlauf können Knoten anfangs derart bijektiv aufeinander abgebildet werden, daß entsprechende Knoten durch Kanten verknüpft sind, gleiche Markierungen tragen und identische Funktionen berechnen. Dann gibt es irgendwo einen ersten Knoten, an dem identische Funktionen berechnet werden, für dessen Nachfolger aber verschiedene Funktionen berechnet werden. Einen solchen Knoten muß es geben, wie man leicht einsieht. Die Quellen repräsentieren gemäß Voraussetzung die gleiche Funktion; gibt es überhaupt keinen solchen Knoten, so sind G' und H im Widerspruch zur Voraussetzung isomorph. Seien u und u' diese Knoten in H und G'. Nachfolger von u und u' seien u_0, u_1

einerseits und u'_0, u'_1 andererseits. Analog zu den Überlegungen oben zur Markierung zur Quelle sieht man ein, daß die Knoten u und u' gleiche Markierungen x_i tragen müssen. Wir nehmen an, daß an den Knoten (ihren Namen entsprechend) die Funktionen g_u, g_{u_0}, g_{u_1}, $g_{u'}$, $g_{u'_0}$ und $g_{u'_1}$ dargestellt werden. Nach Voraussetzung ist einerseits $g_u = g_{u'}$ und andererseits $g_{u_0} \neq g_{u'_0}$ oder $g_{u_1} \neq g_{u'_1}$ (oder beides). Wir setzen $g := g_u$ $(= g_{u'})$. Gemäß Definition 2.18 gilt $g = g_{u_0} \oplus x_i \cdot g_{u_1}$ und $g = g_{u'_0} \oplus x_i \cdot x_i g_{u'_1}$. Wir betrachten zuerst den Fall $x_i = 0$. Dann folgt direkt $g_{u_0} = g_{u'_0}$. Für $x_i = 1$ ergibt sich ebenso direkt $g_{u_1} = g_{u'_1}$, also ein Widerspruch zur Voraussetzung, daß die Funktionen an den Nachfolgern verschieden sind.

Es kann also kein πOFDD für f mit weniger Knoten geben, als H sie hat. H ist das eindeutige minimale πOFDD G für f. □

Die Reduktion von OFDDs ist somit ein wohldefiniertes Problem; wir stehen also wieder vor der Aufgabe, geeignete Reduktionsregeln zu finden. Offenbar können auch in OFDDs isomorphe Subgraphen verschmolzen werden, die Verschmelzungsregel kann also angewendet werden. Aus Satz 2.19 wissen wir, daß ein x_i-Knoten eliminiert werden kann, wenn die an ihm repräsentierte Funktion f nicht essentiell von x_i abhängt. Wenn das der Fall ist, gilt $f_{|x_i=0} = f_{|x_i=1}$, also auch $f_{|x_i=0} \oplus f_{|x_i=1} = 0$. Wenn die Reduktion bottom-up durchgeführt wird, ist beim Erreichen eines x_i-Knotens das unter dem Knoten befindliche OFDD bereits reduziert. Der Knoten kann also entfernt werden, wenn der 1-Nachfolger die 0-Senke ist. Man kann folglich denselben Reduktionsalgorithmus anwenden, der schon bei ZBDDs Anwendung fand (Algorithmus 2.17), da die Reduktionsregeln für ZBDDs und OFDDs identisch sind. Die Laufzeit des Reduktionsalgorithmus ist linear in der Größe der Eingabe. Man kann analog zu OBDDs durch einen Widerspruchsbeweis zeigen, daß ein OFDD, auf das keine der beiden Reduktionsregeln für OFDDs (ZBDDs) anwendbar ist, isomorph zum reduzierten OFDD ist. Damit ist klar, daß der Äquivalenztest als Graphisomorphietest auf den reduzierten OFDDs G und H implementiert werden kann. Laufzeit und Platzbedarf liegen bei $O(|G| + |H|)$.

Der Redundanztest läßt sich leicht in Zeit $O(|G|)$ ohne zusätzlichen Platzbedarf implementieren. Schließlich ist die Variable x_i genau dann redundant, wenn sie nicht als Knotenmarkierung im reduzierten OFDD vorkommt.

Die Synthese ist für πOFDDs ein schweres Problem. Soll zu einem πOFDD F für f und einem πOFDD G für g ein πOFDD H für $h := f \otimes g$ berechnet werden, so benötigt man als 0-Nachfolger der Quelle von H ein πOFDD für $f_0 \otimes g_0$ und als 1-Nachfolger ein πOFDD für $(f_0 \otimes g_0) \oplus (f_1 \otimes g_1)$. Für den 0-Nachfolger kann man also analog zur OBDD-Synthese rekursiv verfahren. Für den 1-Nachfolger ist die Lage aber nicht so einfach: die benötigten Teile stehen in keinem der beiden Eingabe-OFDDs zur Verfügung und müssen erst berechnet werden. Es ist also nicht klar, wie die Größe der Ausgabe dabei kontrolliert werden soll.

Wird als Syntheseoperator allerdings $\oplus$ verwendet, sieht die Lage günstiger aus, wie man sich leicht klarmacht. Sei x_i die der Variablenordnung gemäß erste Variable, von der f oder g essentiell abhängen. Im Hinblick auf die πOFDDs von oben kann man dann folgende Rechnung aufmachen:

$$
\begin{aligned}
h = f \oplus g &= f_{|x_i=0} \oplus x_i(f_{|x_i=0} \oplus f_{|x_i=1}) \oplus g_{|x_i=0} \oplus x_i(g_{|x_i=0} \oplus g_{|x_i=1}) \\
&= (f_{|x_i=0} \oplus g_{|x_i=0}) \oplus x_i \left((f_{|x_i=0} \oplus f_{|x_i=1}) \oplus (g_{|x_i=0} \oplus g_{|x_i=1})\right) \\
&= h_{|x_i=0} \oplus x_i \left(h_{|x_i=0} \oplus h_{|x_i=1}\right)
\end{aligned}
$$

Dabei werden $f_{|x_i=0}$ und $f_{|x_i=0} \oplus f_{|x_i=1}$ in F und $g_{|x_i=0}$ und $g_{|x_i=0} \oplus g_{|x_i=1}$ in G direkt dargestellt. Es genügt also, ganz analog zur OBDD-Synthese einen Paralleldurchlauf durch beide Graphen zu unternehmen. Wir formulieren hier konkret und ausführlich eine rekursive Prozedur `syn(a,b)`, die zu einem πOFDD F für f mit Quelle a und einem πOFDD G für g mit Quelle b ein πOFDD H für $h := f \oplus g$ berechnet. Wir setzen voraus, daß sowohl F als auch G zwei verschiedene Senken enthalten. Wenn das nicht der Fall ist, müssen vor Aufruf der Synthese die fehlenden Senken eingefügt werden. Dadurch erhöht sich die OFDD-Größe höchstens um 1 in jedem OFDD.

Mit label(v) bezeichnen wir die Knotenmarkierung des Knotens v. Wir unterscheiden vier Fälle. Es gilt label(v)<label(w) genau dann, wenn die Markierung von v gemäß Variablenordnung π vor der Markierung von w kommt oder w eine Senke ist.

1.Fall: a und b sind Senken Dann ist das Ergebnis die entsprechende Senke, deren Markierung sich aus der $\oplus$-Summe der Markierungen von a und b ergibt.

2.Fall: label(a)=label(b) Dann ist das Ergebnis ein πOFDD mit einer mit label(a) markierten Quelle. 0-Nachfolger ist die Quelle von `syn(`a_0`,`b_0`)`, 1-Nachfolger ist die Quelle von `syn(`a_1`,`b_1`)`.

3. Fall: label(a)<label(b) Dann ist das Ergebnis ein πOFDD mit einer mit label(a) markierten Quelle. 0-Nachfolger ist die Quelle von `syn(`a_0`,`b`)`, 1-Nachfolger ist die Quelle von `syn(`a_1`,`b`)`.

4. Fall: label(a)>label(b) Dann ist das Ergebnis ein πOFDD mit einer mit label(b) markierten Quelle. 0-Nachfolger ist die Quelle von `syn(`a`,`b_0`)`, 1-Nachfolger ist die Quelle von `syn(`a`,`b_1`)`.

Offenbar wird für jedes Knotenpaar aus dem Kreuzprodukt der Knotenmenge höchstens ein Knoten erzeugt. Die Größe des Ergebnisses ist also durch $O(|G|\,|H|)$ nach oben beschränkt. Die Laufzeit der Synthese kann ebenso wie bei OBDDs mit Hilfe geeigneter Dictionary-Datenstrukturen kontrolliert werden, wobei es auch hier einen trade-off zwischen Speicherplatz und Rechenzeit gibt.

Die Ersetzung von Variablen durch Konstante werden wir zu diesem Zeitpunkt nicht besprechen. Wir verschieben das in den Abschnitt 3.4, da gerade dieses Thema eng mit dem Transformationsproblem verknüpft ist.

Weil wir an Transformationsalgorithmen interessiert sind, macht es Sinn, nach Beziehungen zwischen verschiedenen BDD-Modellen zu fragen. Im Fall der OFDDs gibt es eine strukturelle Beziehung zu OBDDs, die sich in verschiedener Hinsicht als interessant und nützlich erweisen wird. Wir definieren dazu zunächst den Begriff „quasi-reduziert“, dem wir in Abschnitt 3.2.1 in Definition 3.6 noch begegnen werden.

Definition 2.20: Ein πOFDD G für eine Funktion $f : \{0,1\}^n \to \{0,1\}$ heißt *quasi-reduziert*, wenn auf G die Verschmelzungsregel nicht angewendet werden kann und auf jedem Pfad von der Quelle zu einer Senke jede Variablenmarkierung genau einmal vorkommt.

Man kann aus einem beliebigen πOFDD G für f leicht ein quasi-reduziertes πOFDD für f machen. Dazu wendet man zunächst solange die Verschmelzungsregel an, bis es keine zu verschmelzenden Knoten mehr gibt. Anschließend wendet man eine inverse Eliminationsregel für OFDDs an (dargestellt in Abbildung 6), bis auf jedem Pfad von der Quelle zu einer Senke jede Variablenmarkierung genau einmal vorkommt. Dabei muß man darauf achten, daß keine Knoten eingefügt werden, die mit bereits existierenden Knoten verschmolzen werden können. Andernfalls muß man nach Beendigung aller Einfügungen nochmals solange wie möglich die Verschmelzungsregel anwenden.

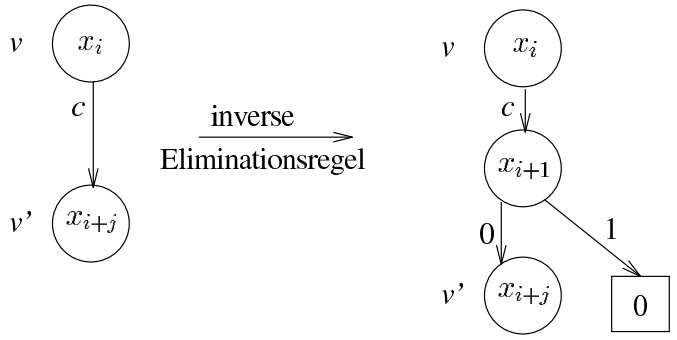


Abbildung 6: Inverse Eliminationsregel für OFDDs

Man kann zeigen, daß das quasi-reduzierte OFDD für f bis auf Isomorphie eindeutig bestimmt ist. Außerdem kann man nachweisen, daß nach erschöpfender Anwendung der Verschmelzungsregel auch nach beliebiger Anwendung der Eliminationsregel die Verschmelzungsregel nicht wieder anwendbar wird. Wir verzichten hier auf den technischen Beweis und verweisen dafür auf Abschnitt 3.2.1, wo ein entsprechender Beweis für ZBDDs geführt ist. Dieser Beweis etabliert die gleichen Aussagen für OFDDs, da die Aussagen nur von den Reduktionsregeln abhängen, die für ZBDDs und OFDDs identisch sind.

Für die Formalisierung der Beziehungen zwischen OBDDs und OFDDs brauchen wir einen Operator τ, wie er etwa bei Becker, Drechsler und Werchner (1995) eingeführt wird.

Definition 2.21: Zu einer Funktion $f : \{0,1\}^n \to \{0,1\}$ wird $\tau(f)$ definiert durch die Funktionswerte $\tau(f)(a)$ für Vektoren $a = (a_1, \ldots, a_n) \in \{0,1\}^n$ mit

$$\tau(f)(a) := \bigoplus_{b=(b_1,\ldots,b_n)\in\{0,1\}^n : \forall 1\leq i\leq n : b_i \leq a_i} f(b).$$

Gemäß Definition ist beispielsweise $\tau(f)((0,\ldots,0)) = f(0,\ldots,0)$ oder $\tau(f)((1,\ldots,1)) = \bigoplus_{b\in\{0,1\}^n} f(b)$. Man kann also $\tau(f)(a)$ als die $\oplus$-Summe aller $f(b)$ beschreiben, wobei b an allen Stellen, an denen in a eine Null auftaucht auch eine Null stehen hat. Alle anderen

Stellen in b werden mit 0 oder 1 besetzt. Wir notieren diese Eigenschaft von b ab jetzt kurz als $b \leq a$.

Man macht sich leicht klar, daß wegen der Kommutativität von $\oplus$ für zwei Boolesche Funktionen f und g stets $\tau(f \oplus g) = \tau(f) \oplus \tau(g)$ gilt. Es gilt nämlich für alle $a \in \{0,1\}^n$:

$$\begin{aligned}
\tau(f \oplus g)(a) &= \bigoplus_{b \leq a} (f \oplus g)(b) \\
&= \bigoplus_{b \leq a} (f(b) \oplus g(b)) \\
&= \left(\bigoplus_{b \leq a} f(b) \right) \oplus \left(\bigoplus_{b \leq a} g(b) \right) \\
&= \tau(f)(a) \oplus \tau(g)(a) \\
&= (\tau(f) \oplus \tau(g))\,(a)
\end{aligned}$$

Damit läßt sich leicht nachrechnen, daß τ zu sich selbst invers ist. Es gilt für alle $a \in \{0,1\}^n$:

$$\begin{aligned}
\tau(\tau(f))(a) &= \left(\bigoplus_{b \leq a} \tau(f)(b) \right) \\
&= \bigoplus_{b \leq a} \bigoplus_{c \leq b} f(c) \\
&= \bigoplus_{c \leq b \leq a} f(c).
\end{aligned}$$

Für jedes $c \neq a$ gibt es stets gerade viele b mit $c \leq b \leq a$, so daß sich diese Terme in der $\oplus$-Summe für $c \neq a$ gerade aufheben. Also ist

$$\bigoplus_{c \leq b \leq a, c \neq a} f(c) = 0$$

und somit

$$\tau(\tau(f))(a) = \bigoplus_{c = a} f(c) = f(a).$$

Gemäß Definition ist τ ein Operator auf der Menge der Booleschen Funktionen über n Variablen, also einer endlichen Menge. Weil — wie eben gesehen — τ zu sich selbst invers ist, ist τ bijektiv.

Nach diesen Vorüberlegungen können wir folgenden Satz beweisen, der zusammen mit seinem Beweis Wegener (1994/95) entnommen ist.

Satz 2.22: Sei G ein quasi-reduziertes OFDD. Sei f_{OFDD} die Funktion, die das OFDD G darstellt. Sei f_{OBDD} die Funktion, die G darstellt, wenn man den Graphen als OBDD interpretiert. Dann gilt $f_{\text{OFDD}} = \tau(f_{\text{OBDD}})$ und $f_{\text{OBDD}} = \tau(f_{\text{OFDD}})$.

Beweis: Offenbar genügt es, eine der beiden Gleichungen zu zeigen. Die andere Gleichung folgt direkt durch Anwendung von τ auf beiden Seiten der Gleichung mit der Tatsache, daß $\tau(\tau(f)) = f$ gilt.

Sei jetzt also f_{OBDD} die Funktion, die der Graph G repräsentiert, wenn man ihn als OBDD interpretiert. Sei $a \in \{0,1\}^n$ eine Eingabe. Wir berechnen $f_{\text{OBDD}}(a)$, indem wir an der Quelle startend einen Pfad bis zu einer Senke durchlaufen. Dabei folgen wir an einem x_i-Knoten der 0-Kante, wenn $a_i = 0$ gilt; wenn $a_i = 1$ gilt, folgen wir der 1-Kante. Der Funktionswert ergibt sich als die Marke der erreichten Senke.

Wir interpretieren G jetzt als OFDD und berechnen $f_{\text{OFDD}}(a)$ für den gleichen Vektor a auf ähnliche Weise. Wir starten mit der Quelle als aktuellem Knoten. Wenn der aktuelle Knoten mit x_i markiert ist und $a_i = 0$ gilt, verfahren wir wie bei OBDDs. Wenn aber $a_i = 1$ ist, so berechnen wir die Funktionswerte für $a_i = 0$ und $a_i = 1$ und verknüpfen die Ergebnisse mit $\oplus$. Wenn a genau k Einsen enthält, sind somit genau die 2^k Funktionswerte zu verknüpfen, die sich für Vektoren b mit $b \leq a$ ergeben, wie man leicht induktiv folgert. Also ist $f_{\text{OFDD}} = \tau(f_{\text{OBDD}})$. □

2.5 Zusammenfassung

Es gibt eine Reihe verschiedener BDD-Modelle, die sich zum Teil in der zugrundeliegenden Syntax, zum Teil in ihrer Semantik unterscheiden. Die vorgestellten BDD-Modelle haben alle verschiedene Stärken und Schwächen: zusammenfassend kann man sagen, daß mit wachsender Effizienz der gewünschten Operationen die Menge der Funktionen, die auf polynomiellem Platz dargestellt werden können, abnimmt[20]. Es gibt keine allgemeine Methode, um sicher vorhersagen zu können, welches BDD-Modell für eine konkrete Funktion am besten geeignet ist. Man darf also davon ausgehen, daß die Verwendung verschiedener BDD-Modelle sinnvoll ist.

Wir halten hier in Form einer Tabelle die Laufzeiten der verschiedenen Operationen fest. Die Bezeichnung RP deutet an, daß es einen RP-Algorithmus gibt, d. h. einen polynomiellen probabilistischen Algorithmus mit einseitigem Fehler. Das Kürzel „exp." bedeutet, daß exponentielle Laufzeit erforderlich ist. Im Gegensatz dazu bedeutet „(exp.)", daß die besten zur Zeit bekannten Algorithmen exponentielle Laufzeit im worst case benötigen; über die Existenz von Polynomialzeitalgorithmen ist damit zunächst einmal nichts ausgesagt[21].

[20] Die Frage, welche Funktionen in einem Modell effizient darstellbar sind, haben wir hier weitgehend ausgeklammert.

[21] Für LBDDs und WBDDs findet man bei Sieling (1995) komplexitätstheoretische Aussagen, die geeignet sind, die Existenz deterministischer Polynomialzeitalgorithmen für einige Probleme unwahrscheinlich zu machen.

Modell	Ausw.	Erfüll.	Minimierung	Äquivalenz	Synthese	$x_i := c$	Redun.
FBDD	$O(n)$	$O(\|G\|)$	(exp.)	RP	exp.	$O(\|G\|)$	(exp.)
OBDD	$O(n)$	$O(\|G\|)$	$O(\|G\|)$	$O(\|G\| + \|H\|)$	$O(\|G\|\,\|H\|)$	$O(\|G\|)$	$O(\|G\|)$
LBDD	$O(n)$	$O(\|G\|)$	$O(\|G\|)$	$O(\|G\| + \|H\|)$	$O(\|G^O\|\,\|G\|\,\|H\|)$	exp.	RP
WBDD	$O(n)$	$O(\|G\|)$	$O(\|G^O\| + \|G\|)$	$O(\|G\| + \|H\|)$	$O(\|G\|\,\|H\|)$	exp.	RP
ZBDD	$O(n)$	$O(\|G\|)$	$O(\|G\|)$	$O(\|G\| + \|H\|)$	$O(\|G\|\,\|H\|)$	$O(\|G\|)$	$O(\|G\|)$
OFDD	$O(\|G\|)$	$O(\|G\|)$	$O(\|G\|)$	$O(\|G\| + \|H\|)$	exp.	$O(\|G\|^2)$	$O(\|G\|)$

Tabelle 1: Übersicht der BDD-Modelle

3 Transformationsalgorithmen

3.1 Die Transformationsalgorithmen von Savický und Wegener

Wir haben in Abschnitt 2.1 grob beschrieben, wie eine πOBDD-Synthese durchzuführen ist[22]. Kern der Transformationen von Savický und Wegener (1994) ist eine spezielle Synthese, die sowohl zeit- als auch platzeffizient durchgeführt werden kann. Der Synthesealgorithmus berechnet zu dem reduzierten πOBDD G für g, dem reduzierten πOBDD H für h und einer Variablen x_i ein reduziertes πOBDD für $f := \text{ite}(x_i, g, h)$. Dabei ist $\text{ite}(x_i, g, h) = x_i g \vee \overline{x_i} h$; ite ist also ein ternärer Operator, der die Funktion if-then-else berechnet: Ist $x_i = 1$, ist $\text{ite}(x_i, g, h) = g$, sonst ist $\text{ite}(x_i, g, h) = h$. Für den Synthesealgorithmus dürfen g und h beide nicht essentiell von x_i abhängen.

Wir nehmen hier weiterhin an, daß alle Funktionen über den Variablen $x_1, \ldots, x_n$ definiert sind. Ist ein Knoten v mit der Variablen x_i markiert, sagen wir auch kurz, der Index von v sei i ($\text{ind}(v) = i$)[23]. Der folgende Satz, der den Grundstein des folgenden Transformationsalgorithmus legt, stammt mitsamt Beweis von Savický und Wegener (1994).

Satz 3.1: Seien $G = (V, E(G))$ und $H = (W, E(H))$ reduzierte πOBDDs für die Funktionen g und h. Beide Graphen enthalten keinen Knoten, der mit x_i markiert ist. Sei $f := \text{ite}(x_i, g, h)$ Das reduzierte πOBDD $F = (U, E(F))$ für f kann in Zeit $O(|F| \log |F|)$ auf Platz $O(|F|)$ konstruiert werden.

Beweis: Als erstes konstruieren wir reduzierte πOBDDs G' für $x_i g$ und H' für $\overline{x_i} h$. Weil πOBDDs für x_i und $\overline{x_i}$ konstante Größe haben, könnte man den normalen Reduktionsalgorithmus benutzen, ohne in Konflikt mit den zu beweisenden Zeit- und Platzschranken zu kommen. Für die extrem einfachen Funktionen kann man allerdings besser einen direkten Algorithmus verwenden, der an die besondere Gestalt der betrachteten Graphen angepaßt ist. Wir betrachten exemplarisch die Konstruktion von G'; das πOBDD H' kann analog konstruiert werden.

Wir nehmen o. B. d. A. an, daß π die Identität ist. Sei $k > i$, sei v ein Knoten in G mit Markierung x_k. Es gibt einen Pfad zur 1-Senke von v aus, da G reduziert ist. Ist $x_i = 0$,

[22] Der Standard-Synthesealgorithmus kann bei Bryant (1986) nachgelesen werden.

[23] Die Schreibweise $\text{ind}(v)$ unterscheidet sich etwas von der bereits verwendeten Notation $\text{label}(v)$. Ist $\text{label}(v) = x_i$, so ist $\text{ind}(v) = i$.

darf die 1-Senke jedoch nicht erreicht werden. Wir fügen darum oberhalb von v einen neuen Knoten w mit Markierung x_i ein. 0-Nachfolger von w ist die 0-Senke, 1-Nachfolger ist v. Alle direkten Vorgänger von v mit einer Markierung, deren Index kleiner als i ist, verlieren ihre Kante zu v. Dafür erhalten sie eine Kante zu w. Hat v keinen solchen Vorgänger, gibt es einen Knoten oberhalb von v mit Index größer i, der einen solchen Vorgänger hat, oder aber v ist Quelle von G. Im ersten Fall genügt es, die Konstruktion für den Vorgänger von v durchzuführen, im zweiten Fall konstruieren wir eine neue Quelle mit Markierung x_i, deren 0-Nachfolger die 0-Senke ist und die als 1-Nachfolger v hat.

Offensichtlich ist die Größe von G' linear in der Größe von G (gleiches gilt für die Größe von H' bezogen auf die Größe von H), die Konstruktion kann natürlich in Linearzeit durchgeführt werden.

Wir wenden jetzt den Standard-Synthesealgorithmus an, der die Synthese unter Vermeidung der Erzeugung nicht erreichbarer Zustände durchführt, um F'' aus G' und H' zu konstruieren. Danach wird aus F'' mit dem Linearzeitreduktionsalgorithmus das πOBDD F gewonnen.

Zum Beweis des Satzes genügt es zu zeigen, daß F mindestens halb so groß ist wie F'' ($|F| \geq \left\lfloor \frac{|F''|}{2} \right\rfloor$); das soll hier geschehen:

Für alle Knoten (v, w) aus F'' gibt es eine partielle Eingabe $a = (a_1, \ldots, a_j) \in \{0,1\}^j$, $0 \leq j \leq n$, so daß der Knoten v in G' und der Knoten w in H' mittels a erreicht wird. Man beachte, daß G' und H' reduziert sind. Sei nun (v, w) ein beliebiger, aber fester Knoten aus F'', und sei $a = (a_1, \ldots, a_j)$ eine solche partielle Eingabe für (v, w). Ist der Index von v höchstens i, so ist notwendigerweise $j < i$, da der Wert von x_i für alle Knoten von G' entscheidend ist, also auf allen Wegen in G' ein x_i-Knoten vorkommt. Gemäß Konstruktion können für eine solche partielle Eingabe a in H' nur Knoten mit Index kleiner i oder die 0-Senke erreicht worden sein. Analoge Überlegungen gelten, wenn der Index von w höchstens i ist. Im verbleibenden Fall sind der Index von v und der Index von w beide echt größer als i. Wenn v und w nicht beides 0-Senken sind, ist j mindestens i. Ist $a_i = 0$, so ist v die 0-Senke in G'. Ist $a_i = 1$, so ist w die 0-Senke in H'. Nach diesen vorbereitenden Betrachtungen kann die Auswirkung der Anwendung der beiden Reduktionsregeln auf F'' betrachtet werden. Wir betrachten zuerst die Eliminationsregel.

1. Fall: Der Knoten (v, w) ist mit x_j markiert, dabei ist $j < i$. Wir nehmen o. B. d. A. an, daß auch v mit x_j markiert ist. Wie wir uns oben überlegt haben, ist w entweder die 0-Senke oder mit einem Index von höchstens i markiert. Setzen wir x_i gleich 1, wird die am Knoten w repräsentierte Funktion 0 und die am Knoten v repräsentierte Funktion zu g_v, der in G bei v dargestellten Funktion. Weil G reduziert ist, hängt die Funktion essentiell von x_j ab. Weil der Knoten (v, w) für $x_i = 1$ ebenfalls g_v repräsentiert, hängt die am Knoten (v, w) repräsentierte Funktion essentiell von x_j ab, der Knoten (v, w) kann also nicht eliminiert werden.

2. Fall: Der Knoten (v, w) ist mit x_i markiert. Die Elimination von (v, w) ist möglich. Die Anzahl der x_i-Knoten in F'' ist aber höchstens um 1 größer als die Anzahl aller x_j-Knoten mit $j < i$ in F''.

3. Fall: Der Knoten (v, w) ist mit x_j markiert, dabei ist $j > i$. Wir nehmen o. B. d. A. an, daß auch v mit x_j markiert ist. Wie wir uns oben überlegt haben, ist w die 0-Senke in H'. Also repräsentiert (v, w) in F'' dieselbe Funktion wie v in G'. Weil v in G' nicht eliminiert werden konnte, kann (v, w) nicht in F'' eliminiert werden.

4. Fall: Der Knoten (v, w) ist eine Senke. Senken werden niemals eliminiert.

Wir betrachten jetzt die Verschmelzungsregel. Es seien (v, w) und (v', w') verschiedene Knoten mit gleicher Markierung von F'', die nicht eliminiert werden können. Wir betrachten keine Senken, da wir stets sicherstellen, daß F'' nur eine 0-Senke und eine 1-Senke enthält.

1. Fall: Die Knoten (v, w) und (v', w') sind mit x_j markiert, dabei ist $j \leq i$. Weil (v, w) und (v', w') verschieden sind, dürfen wir o. B. d. A. annehmen, daß die Knoten v und v' verschieden sind. Mittels Widerspruchsbeweis zeigen wir, daß die Knoten nicht verschmolzen werden können. Wir nehmen dazu also an, daß die Knoten (v, w) und (v', w') verschmolzen werden können. Dann werden an (v, w) und (v', w') identische Funktionen repräsentiert, die essentiell von x_j abhängen. Wie wir uns oben überlegt haben, ist der Index von v höchstens i oder v ist die 0-Senke. Gleiche Überlegungen gelten für v', w und w'. Wir setzen x_i auf 1. Die Funktionen, die an w und w' repräsentiert werden, werden nun durch 0 ersetzt. Der Knoten v ist die 0-Senke oder repräsentiert die Funktion $x_i g^*$ für eine Subfunktion $g^* \neq 0$ von g. Der Knoten v' ist die 0-Senke oder repräsentiert die Funktion $x_i g^{**}$ für eine Subfunktion $g^{**} \neq 0$ von g. Weil v und v' verschieden sind und die Funktionen, die an v und v' repräsentiert werden für $x_i = 1$ identisch sind, repräsentieren v und v' $x_i g^*$ und $x_i g^{**}$, und es gilt $g^* = g^{**}$. Dann müssen aber v und v' in G' verschmolzen werden, ein Widerspruch, da G' reduziert ist und die Knoten v und v' verschieden sind.

2. Fall: Die Knoten (v, w) und (v', w') sind mit x_j markiert, dabei ist $j > i$. Wir nehmen o. B. d. A. an, daß v ebenfalls mit x_j markiert ist. Unseren obigen Überlegungen zufolge ist dann w die 0-Senke in H'. Wir unterscheiden zwei Unterfälle.

1. Unterfall: v' ist mit x_j markiert. Dann ist w' die 0-Senke in H'. Verschmelzen ist nicht möglich, da sonst v und v' in G' verschmolzen worden wären; G' ist aber reduziert.

2. Unterfall: w' ist mit x_j markiert. Dann ist v' die 0-Senke in G'. Jetzt ist Verschmelzen nicht auszuschließen, da v und w' möglicherweise die gleiche Funktion repräsentieren.

Wir betrachten, wie viele Knoten höchstens eliminiert oder verschmolzen werden können. Die Eliminationsregel kann höchstens auf x_i-Knoten angewendet werden, deren Anzahl ist aber nach oben begrenzt durch die Zahl der x_j-Knoten mit $j < i$ abzüglich 1. Die Verschmelzungsregel kann wiederum nur auf x_j-Knoten mit $j > i$ angewendet werden. Es ist aber gemäß der obigen Fallunterscheidung nicht möglich, drei oder mehr Knoten

miteinander zu verschmelzen. Also kann die Anzahl der x_j-Knoten mit $j > i$ höchstens halbiert werden. Folglich kann die Knotenzahl insgesamt höchstens halbiert werden. Es ist also $|F| \geq \left\lfloor \frac{|F''|}{2} \right\rfloor$. □

Diese ite-Synthese ist der erste nicht triviale Fall, wo bei einer OBDD-Synthese eine $O(|F| \log |F|)$ Laufzeit- und $O(|F|)$ Platzschranke garantiert werden können, wobei beide Schranken nur von der Größe des reduzierten Ergebnis-OBDD abhängen. Dieses schon für sich interessante Resultat wird so wertvoll, weil es mit Gewinn für einen effizienten Transformationsalgorithmus verwendet werden kann. Die Transformation eines FBDD in ein πOBDD, die hier als nächstes vorgestellt wird, basiert im Kern auf der Effizienz der ite-Synthese.

3.1.1 Transformation eines FBDD in ein πOBDD

Die Transformation eines FBDD in ein πOBDD entpuppt sich als realisierbar mit fortgesetzter Anwendung der ite-Synthese aus Satz 3.1. Weiterer erheblicher Aufwand ist interessanterweise weder im Algorithmus noch in der Analyse nötig.

Satz 3.2: Zu einem FBDD F für eine Boolesche Funktion f auf den Variablen $x_1, \ldots, x_n$ und einer Variablenordnung π kann das reduzierte πOBDD G für f in Zeit $O(|F|\,|G| \log |G|)$ auf Platz $O(|F|\,|G|)$ berechnet werden. Ein Algorithmus kann so gestaltet werden, daß er zusätzlich eine ganze Zahl max als Eingabe bekommt und G ausgibt, wenn $|G| \leq max$ gilt; andernfalls gibt er „G zu groß“ aus. Dabei kann für den Algorithmus in den Platz- und Laufzeitschranken $|G|$ durch $\min\{max, |G|\}$ ersetzt werden.

Beweis: O. B. d. A. können wir annehmen, daß F für jede Variable x_i, $1 \leq i \leq n$, auch einen Knoten enthält, sonst operiert man mit einer kleineren Variablenmenge. Wir können außerdem o. B. d. A. annehmen, daß π die Identität ist. Im ersten Schritt werden im Rahmen eines DFS-Durchlaufs alle von der Quelle aus nicht erreichbaren Knoten eliminiert. Alle nicht gestrichenen Knoten werden anschließend in umgekehrt topologischer Reihenfolge bearbeitet. Für jeden dieser Knoten v wird zu der dort repräsentierten Funktion f_v das reduzierte πOBDD $G(v)$ berechnet. Für die beiden Senken ist das trivial. Wir dürfen jetzt nach Induktionsvoraussetzung annehmen, daß das πOBDD $G(v)$ für einen Knoten v mit 0-Nachfolger v_0 und 1-Nachfolger v_1 konstruiert werden soll. Für die dort repräsentierten Funktionen f_{v_0} und f_{v_1} sind die reduzierten πOBDDs $G(v_0)$ und $G(v_1)$ bereits konstruiert. Weil F ein FBDD ist und somit der read-once Eigenschaft genügt, enthalten weder $G(v_0)$ noch $G(v_1)$ einen x_i-Knoten. Offensichtlich ist $f = \text{ite}(x_i, f_{v_1}, f_{v_0})$, also kann zur Konstruktion von $G(v)$ die ite-Synthese aus Satz 3.1 benutzt werden, es genügen also Zeit $O(|G(v)| \log |G(v)|)$ und Platz $O(|G(v)|)$. Alle so konstruierten πOBDDs $G(v)$ sind jeweils nicht größer als G, da $f(v)$ jeweils eine Subfunktion von f ist. Weil insgesamt $|F|$ πOBDDs konstruiert werden, ergeben sich die behaupteten Platz- und Zeitschranken. Der Abbruch nach höchstens max konstruierten Knoten läßt sich leicht integrieren. □

3.1.2 Transformation eines π'OBDD in ein πOBDD

Transformationen, bei denen das BDD-Modell nicht verlassen wird und lediglich die zugrundeliegende Variablenordnung (in Form einer Permutation π oder eines Orakelgraphen G^O) gewechselt wird, nennen wir *Umordnung*. Die Transformation eines π'OBDD in ein πOBDD (also die OBDD-Umordnung) ist ein Spezialfall der Transformation eines FBDD in ein OBDD, da jedes OBDD als ein spezielles FBDD betrachtet werden kann; zusätzlich zur read-once Eigenschaft wird auf jedem Weg von der Quelle zur Senke eine bestimmte Variablenreihenfolge eingehalten. Der Algorithmus aus Abschnitt 3.1.1 kann durchaus zur Lösung des Problems verwendet werden, es liegt allerdings nahe, nach Verbesserungen zu fragen, die von der stärkeren Struktur der Eingabe profitieren. Eine Verbesserung der im allgemeinen Algorithmus erzielten Rechenzeit bieten Savický und Wegener nicht, allerdings können sie in diesem Fall einen platzoptimalen Algorithmus formulieren. Da bei typischen OBDD-Anwendungen gerade Platz die kritische Systemressource darstellt, ist diese Verbesserung als interessant und praktisch relevant zu betrachten. Wir sollten hier nochmals erwähnen, daß bei BDD-Algorithmen die Ausgabe häufig wesentlich größer als die Eingabe sein kann. Wir nennen einen Algorithmus deshalb optimal, wenn er linear in der Größe von Eingabe *und* Ausgabe ist.

Satz 3.3: Sei F ein π'OBDD für die Boolesche Funktion f auf den Variablen $x_1, \ldots, x_n$ und π eine Permutation. Das reduzierte πOBDD G für f kann in Zeit $O(|F|\,|G| \log |G|)$ auf Platz $O(|F| + |G|)$ berechnet werden. Analog zu Satz 3.2 kann ein Algorithmus so implementiert werden, daß er eine Größenschranke max als zusätzliche Eingabe verwendet. Der Algorithmus gibt G aus, wenn $|G| \leq max$ gilt und sonst die Meldung „G zu groß“. Dabei kann in den Zeit- und Platzschranken wie oben $|G|$ durch $\min\{max, |G|\}$ ersetzt werden.

Beweis: Zuerst wird F mit dem Linearzeitalgorithmus von Sieling und Wegener (1993) (Algorithmus 2.5) reduziert, das Ergebnis heißt ebenfalls F. Wir dürfen o. B. d. A. annehmen, daß F für alle i einen x_i-Knoten enthält, sonst operieren wir auf einer kleineren Variablenmenge. Das πOBDD G wird top-down konstruiert; es kann sowohl in BFS- als auch in DFS-Reihenfolge vorgegangen werden. Die Konstruktion wird derart durchgeführt, daß nur Knoten erzeugt werden, die nicht eliminiert werden können. Erzeugte Knoten können möglicherweise verschmolzen werden, in solchen Fällen wird eine Verschmelzung durchgeführt, bevor die Nachfolger dieser Knoten erzeugt werden. Damit ist klar, daß die Konstruktion platzoptimal durchgeführt werden kann.

Um von jedem Knoten effizient zurück zur Quelle gehen zu können, werden alle Kanten auch in umgekehrter Richtung abgespeichert. Wir führen folgenden Algorithmus aus.

Algorithmus 3.4 (π'OBDD$\rightarrow \pi$OBDD):

```
/* Eingabe: π'OBDD G für f */
/* Ausgabe: reduziertes πOBDD H für f (o.B.d.A. π = id) */
1.  Initialisiere H mit zwei Senken und einer Quelle, die mit
    x_1 markiert wird.
2.  L_1:={Quelle von H};
```

```
3.  for i:=2 to n do L_i:=nil;
4.  for i:=1 to n do
5.      Bestimme für alle Knoten w_1,...,w_r(i),
        ob sie die gleichen Subfunktionen von f repräsentieren.
        In diesem Fall verschmilz diese Knoten.
6.      Seien w_1,...,w_s(i) die restlichen Knoten in L_i;
7.      For j:=1 to s(i) do
8.          for c:=0 to 1 do
9.              Entscheide, ob der c-Nachfolger von w_j in H eine
                Senke oder einen Knoten mit Marke x_t, i < t ≤ n ist.
10.             Erzeuge entweder die Kante zur Senke oder die Kante zu
                dem neu zu erzeugenden x_t-Knoten.
11.             Füge x_t zusammen mit dem Hinweis auf den Vater
                zur Liste L_t hinzu.
```

Wenn die fehlenden Lücken noch richtig aufgefüllt werden, gelten für das so konstruierte πOBDD folgende Aussagen:

1. Das πOBDD H repräsentiert f.
2. Es werden keine eliminierbaren Knoten erzeugt.
3. Alle verschmelzbaren Knoten werden verschmolzen.

Weil das erzeugte πOBDD f repräsentiert und reduziert ist, wird wie behauptet H konstruiert.

Wir beschreiben zunächst den Schritt 5 ausführlicher. Wir identifizieren jeden Knoten v_j mit $1 \leq j \leq r(i)$ mit einem reduzierten π'OBDD $G_{j,i}$, das wir aus G erhalten, indem wir einige Variablen konstant setzen und das resultierende π'OBDD reduzieren. Der Konstantsetzung muß ein Pfad von der Quelle zu v_j entsprechen, das ist aber leicht berechenbar, da wir den Pfad zur Quelle abgespeichert haben. Also ist die Konstantsetzung genau wie die Reduktion in Linearzeit durchführbar. Das resultierende π'OBDD $G_{j,i}$ wird durch eine Folge von $O(|G|)$ Zahlen codiert; diese Codierung wird im Rahmen einer Tiefensuche berechnet, wobei immer der 0-Nachfolger vor dem 1-Nachfolger besucht wird. Jeder Knoten wird repräsentiert durch seine DFS-Nummer, seine Markierung, seinen 0-Nachfolger und seinen 1-Nachfolger. Wir bezeichnen die Codierung eines OBDD als ein Codewort und bemerken, daß die Berechnung eines Codeworts in Linearzeit möglich ist. Die π'OBDDs für $G_{j,i}$ und $G_{j',i}$ können nur verschmolzen werden, wenn ihre Codewörter identisch sind, weil die Knoten genau dann die gleiche Funktion repräsentieren, wenn die reduzierten πOBDDs $G_{j,i}$ und $G_{j',i}$ isomorph sind. Wir definieren eine Ordnung auf den π'OBDDs, indem wir die lexikographische Ordnung auf den Codewörtern benutzen. Wir speichern die Knoten $v_1, \ldots, v_{r(i)}$ in einem AVL-Baum, der leer initialisiert wird. Wenn zwei gleiche Knoten gefunden werden, werden diese Knoten in H verschmolzen. Um Platz zu sparen, werden nicht die Codewörter sondern nur die Knoten gespeichert. Die Codewörter werden

stets neu berechnet, wenn sie benötigt werden; jede Neuberechnung kostet Zeit $O(|G|)$. Es müssen also zu jedem Zeitpunkt höchstens zwei Codewörter gleichzeitig im Speicher gehalten werden. Es sind nur $O(r(i))$ Operationen im AVL-Baum durchzuführen; weil $r(i)$ die Anzahl der Kanten in H bezeichnet, die zu x_i-Knoten führen, gibt es ingesamt $O(|H|)$ Operationen im AVL-Baum. Jede einzelne Operation benötigt höchstens Zeit $O(|G|)$, der Gesamtzeitbedarf beträgt also höchstens $O(|G|\,|H| \log |H|)$. Die Zeit für die Verschmelzungen ist nach oben durch $O(|H|)$ beschränkt, da die umgekehrten Kanten verfügbar sind und jede Kante höchstens einmal betrachtet wird.

Jetzt beschreiben wir Schritt 9 ausführlicher. Für den c-Nachfolger von w_j mit $1 \leq j \leq s(i)$ konstruieren wir das reduzierte π'OBDD $G_{j,i,c}$ durch Ersetzung einiger Variablen durch Konstante — dabei wird wie oben ein Pfad von der Quelle zum Knoten benötigt —; außerdem wird $x_i := c$ gesetzt. Danach wird in Linearzeit reduziert. Wenn $G_{j,i,c}$ nur aus einer Senke besteht, ist der c-Nachfolger die entsprechende Senke in H. Andernfalls sei t die kleinste Zahl, so daß $G_{j,i,c}$ einen x_t-Knoten enthält. Die Zeit für alle Operationen zusammen ist offensichtlich durch $O(|G|\,|H|)$ nach oben beschränkt.

Wir betrachten noch die Benutzung der Platzschranke max. Dazu genügt es, die in H erzeugten Kanten zu zählen. In einem πOBDD H gibt es, wenn H eine nicht-konstante Funktion repräsentiert, genau $2\,|H| - 4$ Kanten, da jeder innere Knoten fan-out 2 hat, die beiden Senken fan-out 0. Wenn die $(2max - 3)$-te Kante in H erzeugt wird, kann die Konstruktion abgebrochen werden, sonst wird H wie beschrieben konstruiert. □

Wir werden eine Verallgemeinerung dieses Algorithmus in Abschnitt 3.3.4 angeben, mit der die Transformation eines baumgesteuerten BDD (ein graphgesteuertes BDD mit baumförmigem Orakelgraphen, vgl. Definition 3.21) platzoptimal durchgeführt werden kann. In Abschnitt 3.4.4 werden wir ebenfalls eine Variante dieses Algorithmus angeben, mit der praktisch jedes Transformationsproblem platzoptimal gelöst werden kann; die Platzoptimalität ist dort allerdings mit exponentieller Laufzeit für jede Eingabe verbunden.

3.2 Transformationen mit ZBDDs

Wir beginnen unsere Bemühungen um Transformationsalgorithmen mit ZBDDs mit der Konstruktion eines Algorithmus für die Transformation eines πOBDD in ein πZBDD. Die Transformation zwischen einem OBDD und einem ZBDD folgt notwendig einem gänzlich anderem Paradigma als die Transformationsalgorithmen von Savický und Wegener. OBDDs und ZBDDs haben im Gegensatz zu FBDDs und OBDDs eine unterschiedliche Semantik. Zum Ausgleich dafür unterliegen beide einer Variablenordnung. Wir gehen in diesem Abschnitt davon aus, daß ein Wechsel der Variablenordnung bei der Transformation nicht vorgesehen ist. Wir werden später überlegen, was sich ändert, wenn man diese vereinfachende Annahme fallen läßt.

Um die Transformationsalgorithmen verstehen zu können, ist etwas Vorarbeit nötig. Man braucht Kenntnisse über die Verbindungen zwischen OBDDs und ZBDDs. Dabei können wir (sozusagen als Abfallprodukt dieser für die Transformation genutzten Beziehungen)

gleich etwas über die maximalen Größenunterschiede zwischen reduzierten πOBDDs und reduzierten πZBDDs aussagen.

Wir nehmen wie immer o. B. d. A. an, daß die Variablenordnung π die Identität ist. Sei weiter $f : \{0,1\}^n \to \{0,1\}$ eine beliebige Boolesche Funktion.

Definition 3.5: Ein *vollständiger Entscheidungsbaum* für f ist ein πOBDD mit folgender Gestalt: Der Baum besteht aus n Ebenen, die von oben nach unten von 1 bis n durchnumeriert sind. Auf der Ebene i befinden sich 2^{i-1} Knoten, die alle mit x_i markiert sind. Kanten gibt es nur zwischen benachbarten Ebenen, alle Kanten führen (wegen der Variablenordnung notwendig) von oben nach unten. Die Knoten der untersten Ebene erhalten derart Kanten zu den Senken, daß das OBDD die Funktion f repräsentiert.

Wir fordern (etwas unkonventionell) von einem vollständigen Entscheidungsbaum nur, daß er in bezug auf die mit Variablen markierten Knoten baumförmig ist. Wir lassen also durchaus zu, daß ein vollständiger Entscheidungsbaum weniger als 2^n Senken hat.

Gemäß Definition ist ein vollständiger Entscheidungsbaum für f ein OBDD mit genau $2^{n+1} - 1$ Knoten, wenn man die Senken nicht mitzählt; das OBDD ist natürlich (jedenfalls für $n > 2$) nicht reduziert. Der Entscheidungsbaum ist aber auch ein ZBDD für die gleiche Funktion, das (jedenfalls für $n > 2$) ebenfalls nicht reduziert ist. Das liegt daran, daß jeder Pfad zu einer 1-Senke im vollständigen Entscheidungsbaum einen vollständigen Eingabevektor definiert. Die besondere Rolle der 0 bei Auslassungen von Variablentests in ZBDDs spielt hier also keine Rolle.

Weil bei OBDDs und ZBDDs die gleiche Verschmelzungsregel als Bestandteil eines Reduktionsverfahrens dienen kann, liegt nach vollständiger Anwendung der Verschmelzungsregel immer noch *ein* Graph vor, der sowohl als OBDD wie auch als ZBDD für die Funktion f interpretiert werden kann. Der Graph ist offenbar im allgemeinen nicht reduziert, weder für OBDDs noch für ZBDDs.

Die Eliminationsregeln sind für OBDDs und ZBDDs bekanntlich verschieden. Bei der Anwendung der jeweiligen Eliminationsregel, um ein minimales OBDD bzw. ZBDD zu erhalten, ergeben sich also Unterschiede.

Man darf somit festhalten: Ausgehend von einem vollständigen Entscheidungsbaum ergeben sich bezüglich OBDD und ZBDD Unterschiede einzig durch Anwendung der unterschiedlichen Eliminationsregeln.

Der Sachverhalt soll durch eine Definition festgehalten werden, deren Sinngehalt im Anschluß daran behauptet und bewiesen wird. Die folgende Definition 3.6 und der darauf sich beziehende Satz 3.7 sind Schröer und Wegener (1994) entnommen. Wir geben hier für Satz 3.7 einen strukturell einfacheren Beweis an.

Definition 3.6: Ein *quasi-reduziertes* OBDD (ZBDD) für eine Boolesche Funktion $f : \{0,1\}^n \to \{0,1\}$ entsteht aus einem vollständigen Entscheidungsbaum für die Funktion f durch erschöpfendes Anwenden der Verschmelzungsregel.

Weil Knoten nur verschmolzen aber nie gelöscht werden, ist auf jedem Pfad von der Quelle zur Senke stets jede Variablenmarkierung genau einmal zu finden; darum sind das quasi-reduzierte πOBDD und das quasi-reduzierte πZBDD für eine Funktion f auch identisch.

Die Definition 3.6 behauptet implizit, daß ein „erschöpfendes Anwenden der Verschmelzungsregel" auf Entscheidungsbäume möglich ist; nach Anwendung der Eliminationsregel kann die Verschmelzungsregel also nicht wieder angewendet werden. Insbesondere der Umstand, daß die Eliminiationsregel auch durch Anwenden der Verschmelzungsregel nicht wieder anwendbar ist, kann nicht als selbstverständlich bezeichnet werden. Wir halten diese Tatsache deshalb hier explizit mit Satz und Beweis fest.

Satz 3.7: Auf ein quasi-reduziertes OBDD (ZBDD) kann die Verschmelzungsregel nicht mehr angewendet werden; das gilt auch für alle die OBDDs (ZBDDs), die durch Anwendung der Eliminationsregel aus dem quasi-reduzierten OBDD (ZBDD) hervorgehen.

Wir formulieren zunächst eine einfache Hilfsaussage.

Lemma 3.8: Die Anwendung einer Reduktionsregel auf ein OBDD (ZBDD) entfernt genau einen Knoten und läßt bei allen anderen Knoten die Markierungen und die dargestellten Funktionen unverändert.

Beweis: Die Aussage folgt direkt aus den Eigenschaften der Reduktionsregeln, unabhängig davon ob man die OBDD- oder ZBDD-Reduktionsregeln betrachtet. Bei der Anwendung der Verschmelzungsregel, die sich für beide Modelle nicht unterscheidet, ist es gleichgültig, welcher Knoten entfernt und welcher Knoten in seinen Eigenschaften unberührt belassen wird. □

Beweis von Satz 3.7: Sei T ein vollständiger Entscheidungsbaum für eine Boolesche Funktion $f : \{0,1\}^n \to \{0,1\}$. Sei $V = \{s_0, s_1, v_1, \ldots, v_l\}$ die Knotenmenge des OBDD (ZBDD) G, das durch erschöpfende Anwendung der Verschmelzungsregel auf T entsteht. Wir setzen o. B. d. A. voraus, daß G genau zwei verschieden markierte Senken s_0 und s_1 hat. Wir bezeichnen für $1 \leq i \leq l$ mit m_i die Markierung des Knotens v_i und mit f_i die am Knoten v_i repräsentierte Funktion. Wir beweisen zunächst eine Aussage über die in G repräsentierten Funktionen.

Lemma 3.9: Für alle $1 \leq j \neq k \leq l$ gilt, daß für alle Knoten $v_j, v_k \in V$ mit $m_j = m_k = x_i$ notwendig $f_j \neq f_k$ gilt.

Beweis: Wir führen einen Beweis durch Widerspruch und nehmen an, daß v_j und v_k zwei Knoten aus V mit $m_j = m_k = x_i$ sind und $f_j = f_k$ gilt. Diese Knoten seien so gewählt, daß i maximal bezüglich der zugrundeliegenden Variablenordnung ist. Weil die Funktionen f_j und f_k gleich sind, gilt das auch für ihre Subfunktionen bezüglich $x_i = 0$ und $x_i = 1$. Weil auf T nach Voraussetzung noch niemals die Eliminationsregel angewendet wurde, gilt für die 0-Nachfolger von v_j und v_k, daß sie gleiche Markierungen tragen oder beides Senken sind. Das gilt auch für die 1-Nachfolger. Weil die Subfunktionen gleich sind, müssen die an den 0-Nachfolgern bzw. an den 1-Nachfolgern repräsentierten Funktionen gleich sein.

Wegen der Maximalität von i haben v_j und v_k notwendig denselben 0-Nachfolger und denselben 1-Nachfolger. Dann können v_j und v_k verschmolzen werden im Widerspruch zur Voraussetzung, daß die Verschmelzungsregel erschöpfend angewendet wurde. □

Die Verschmelzungsregel kann gemäß ihrer Definition nur auf zwei Knoten mit gleicher Knotenmarkierung angewendet werden, wenn diese Knoten die gleiche Funktion repräsentieren. Weil die Anwendung der Eliminationsregel die an Knoten repräsentierten Funktionen nicht ändert gemäß Lemma 3.8 und alle Knoten mit gleicher Markierung in G verschiedene Funktionen repräsentieren gemäß Lemma 3.9, kann die Verschmelzungsregel auch nach beliebiger Anwendung der Eliminationsregel nicht mehr angewendet werden. □

Wie haben im zweiten Kapitel gesehen, daß die Algorithmen auf BDD-Varianten häufig effizient (oft linear) in der Größe der beteiligten Graphen sind. Aus diesem Grund und weil die Graphen oft sehr groß — gemessen an der Speicherkapazität aktueller Rechner — werden, ist in praktischer Hinsicht vor allem die Größe der Datenstruktur für Boolesche Funktionen von Interesse; es soll also an dieser Stelle festgestellt werden, wie groß die Unterschiede zwischen OBDDs und ZBDDs überhaupt werden können. Interessanterweise ist es möglich, klare Grenzen anzugeben, die zeigen, daß die Differenzen in einem zwar erheblichen, für praktische Zwecke aber nicht unüberschaubaren Rahmen bleiben. Die Grenzen sind scharf in dem Sinne, daß sie sowohl obere Schranken für die Größenunterschiede darstellen, es aber auch Funktionen gibt, bei denen diese Unterschiede tatsächlich auftreten; es handelt sich also gleichzeitig um untere Schranken. Der folgende Satz ist zusammen mit seinem Beweis Schröer und Wegener (1994) entnommen.

Satz 3.10: Sei f eine beliebige Boolesche Funktion $f : \{0,1\}^n \to \{0,1\}$. Sei O_f das reduzierte OBDD für f und Z_f das reduzierte ZBDD jeweils für die Variablenordnung $x_1, \ldots, x_n$. Dann gilt immer $|Z_f| \leq (n+1)\,|O_f|$.

Beweis: Sei G das quasi-reduzierte BDD für die gegebene Funktion f und die gegebene Variablenordnung. Wir werden schrittweise alle Knoten von G rot oder blau färben. Blau gefärbt werden nur solche Knoten, die gemäß Eliminationsregel für OBDDs nicht entfernt werden können. Wir bezeichnen die Anzahl der blau gefärbten Knoten mit b, die Anzahl der rot gefärbten Knoten mit r. Am Ende des schrittweisen Färbens gilt für die Anzahlen r und b, daß O_f mindestens b Knoten hat; Z_f kann nicht mehr Knoten haben als G, hat also höchstens $r+b$ Knoten. Es ist folglich $\frac{|Z_f|}{|O_f|} \leq \frac{r+b}{b}$. Es genügt demnach zu zeigen, daß $r \leq nb$ ist.

Wir färben die Knoten in G schrittweise rot und blau, dabei bezeichnen wir die Anzahlen nach dem i-ten Schritt mit r_i und b_i. Die Färbung wird so durchgeführt, daß jeweils $r_i \leq nb_i$ gilt. Es wird also ein Induktionsbeweis geführt.

Im ersten Schritt betrachten wir einen beliebigen Berechnungspfad in G und färben die Senke blau, alle anderen Knoten des Pfades rot. Da G aus quasi-reduziert ist, gilt $r_1 = n$ und $b_1 = 1$. Damit ist $r_i \leq nb_i$ erfüllt für $i = 1$.

Jetzt kommen wir zum Induktionsschritt. Sind alle Knoten gefärbt, ist der Beweis geführt. Sonst wählen wir einen Berechnungspfad in G, auf dem noch mindestens ein ungefärbter Knoten liegt. Es liegen auf dem Pfad auch gefärbte Knoten, zumindest die Quelle ist gefärbt. Nicht gefärbte Knoten werden rot gefärbt. Rot gefärbte Knoten, die einen nicht gefärbten Nachfolger haben, werden blau gefärbt. Das ist korrekt, da sie offenbar zwei nicht identische Nachfolger haben, also nicht eliminiert werden können. Es gibt höchstens n ungefärbte Knoten auf dem Pfad, also gilt $r_{i+1} \leq r_i + n$. Weil blau gefärbte Knoten ihre Farbe immer behalten, gilt $b_{i+1} \geq b_i + 1$. Nach Induktionsvoraussetzung gilt $r_i \leq nb_i$. Damit gilt $r_{i+1} \leq nb_{i+1}$. □

Der Größenunterschied ist also im wesentlichen durch einen Faktor n beschränkt. Das gilt auch in umgekehrter Richtung. Die Aussage mit Beweis ist ebenfalls Schröer und Wegener (1994) entnommen.

Satz 3.11: Seien die Voraussetzungen wie in Satz 3.10 gegeben. Dann gilt stets $|O_f| \leq (n+1)\,|Z_f| + 1$.

Beweis: Der Beweis wird von der gleichen Idee geleitet wie der Beweis von Satz 3.10. Die Unterschiede in der Durchführung ergeben sich durch die Unterschiede in den jeweiligen Eliminationsregeln. □

Aus der Tatsache, daß ein quasi-reduziertes πOBDD für f gleichzeitig ein quasi-reduziertes πZBDD für f ist (und umgekehrt) ergibt sich offensichtlich ein Hinweis auf einen Transformationsalgorithmus. Wir betrachten zunächst die Transformation eines πOBDD in ein πZBDD.

3.2.1 Transformation eines πOBDD in ein πZBDD

Für die Konstruktion eines πZBDD aus einem πOBDD folgen wir den Hinweisen, die sich durch die strukturelle Verknüpfung von OBDDs und ZBDDs ergeben, wie wir sie eben festgestellt haben. Zunächst gehen wir naiv vor, dann werden wir unser Vorgehen optimieren.

Wir berechnen zu einem reduzierten πOBDD zu f das quasi-reduzierte πOBDD zu f. Dann reduzieren wir es mit der ZBDD-Eliminationsregel und erhalten das reduzierte πZBDD für f. Der Algorithmus erzeugt kein allzu großes Zwischenergebnis: der maximale Größenunterschied zwischen reduzierten πOBDDs und reduzierten πZBDDs für f aus Satz 3.10 wurde bewiesen durch Betrachtung des quasi-reduzierten BDD. Damit ist auch gezeigt, daß der Größenunterschied zwischen einem reduzierten OBDD (ZBDD) und dem quasi-reduzierten BDD nicht größer sein kann.

Wir wollen jetzt in einem ersten Schritt überlegen, wie man konkret aus einem reduzierten πOBDD für f das quasi-reduzierte πOBDD für f berechnet. Danach werden wir durch verschiedene Modifikationen an dem Algorithmus einen Algorithmus konstruieren, der direkt eine πOBDD$\rightarrow$ πZBDD-Transformation durchführt.

Bei der Anwendung der OBDD-Eliminationsregel werden Knoten entfernt, die als 0- *und* 1-Nachfolger denselben Knoten v haben. Die Umkehrung dieser Regel muß also solche Knoten einfügen. Will man in jedem Schritt genau einen Knoten einfügen, ist nicht eindeutig bestimmt, welche Markierung der neue Knoten erhalten soll. Führt eine Kante von einem x_i-Knoten zu einem x_{i+j}-Knoten ($j > 0$), so kann zwischen den beiden Knoten ein x_{i+l}-Knoten eingefügt werden, wobei $0 < l < j$ frei gewählt werden kann. Wir legen fest, daß in solchen Fällen ein x_{i+1}-Knoten einzufügen ist und sprechen von nun ab von *der* inversen Eliminationsregel (obwohl es wie besprochen andere gibt).

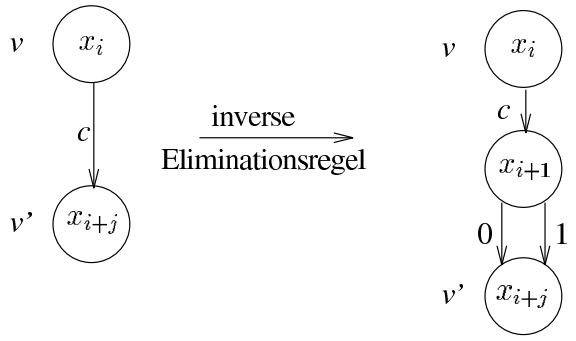


Abbildung 7: Inverse Eliminationsregel für OBDDs

Die einmalige Anwendung der Eliminationsregel ist offenbar in konstanter Zeit möglich, wenn der Vater des einzufügenden Knotens gegeben ist. Beim Einfügen ist darauf zu achten, daß keine Knoten eingefügt werden, die mit anderen Knoten verschmolzen werden können; sonst entsteht nicht das quasi-reduzierte OBDD. Dabei sind nur neue Knoten zu berücksichtigen: Alle neuen Knoten haben gleichen 0- und 1-Nachfolger und sind also redundant, bei allen anderen Knoten ist das nicht der Fall. Neue und „alte“ Knoten können folglich niemals miteinander verschmolzen werden.

Satz 3.12: Zu einem πOBDD G für eine Boolesche Funktion f kann das quasi-reduzierte πOBDD für f in Zeit $O(n\,|G|\log(n\,|G|))$ auf Platz $O(n\,|G|)$ berechnet werden.

Wir werden den Beweis führen, indem wir einen Algorithmus mit den behaupteten Eigenschaften als korrekt nachweisen. Zunächst präsentieren wir den Algorithmus.

Wir gehen analog zur Reduktion ebenenweise von unten nach oben vor (bottom-up). Wir halten dabei alle alten Knoten in Knotenlisten L_i, die neuen Knoten werden in Knotenlisten N_i gesammelt (jeweils $1 \leq i \leq n$). Knoten mit gleicher Markierung werden in derselben Liste gesammelt. Weil wir mit einem reduzierten OBDD beginnen, brauchen wir die Senken nicht zu betrachten. Ist das Eingabe-OBDD nicht reduziert, kann das in Linearzeit auf linearem Platz geändert werden.

Bei der Notation des folgenden Algorithmus bezeichnen wir diesmal den 0-Nachfolger eines Knoten v mit v_0, den 1-Nachfolger mit v_1, um bequem in der Schleife ab Zeile 5 über beide Nachfolger eines Knotens laufen zu können.

Algorithmus 3.13 (OBDD→quasi-reduziertes OBDD):

```
/* Eingabe: reduziertes πOBDD G für f (o.B.d.A. π = id) */
/* Ausgabe: quasi-reduziertes πOBDD für f */
1.   for i:=1 to n do L_i:=N_i:=nil; (* leere Listen *)
2.   for v∈V do füge v in Liste L_ind(v) ein;
3.   for i:=n-1 downto 1 do
4.        for v∈ L_i do
5.             for c:=0 to 1 do
6.                  if label(v)+1<label(v_c) then
7.                       w:=Knoten aus N_{i+1} mit gleichen
                         Nachfolgern wie v_c;
8.                       if w existiert nicht then
9.                            Erzeuge solchen Knoten w;
10.                           Füge w in N_{i+1} ein;
11.                           Verfahre für w wie für v_c;
12.                      Mache w zum c-Nachfolger von v;
13.  if Quelle ist nicht x_1-Knoten then
14.       Füge Knotenkette vor der Quelle ein;
```

In den Zeilen 13 und 14 wird eine Sonderbetrachtung nur für die Quelle durchgeführt. Das ist notwendig, weil der Algorithmus neue Knoten stets nur als Nachfolger vorhandener Knoten einfügt. Die Quelle hat als einziger Knoten keinen Vorgänger, sodaß oberhalb der Quelle fehlende Knoten nicht eingefügt werden. Wir werden noch öfter Algorithmen begegnen, die Schwierigkeiten mit dieser Eigenart der Quelle haben. Man kann sich die Sonderbehandlung der Quelle aber mit einem kleinen Trick ersparen. Wenn die Quelle nicht mit der gemäß Variablenordnung π ersten Variablen markiert ist, kann man einen redundanten Knoten als neue Quelle einfügen, der mit der ersten Variablen markiert ist. Oberhalb dieser neuen Quelle können keine Einfügungen mehr erforderlich sein, eine Sonderbehandlung der Quelle ist also nicht mehr erforderlich. Wir werden von diesem Kunstgriff gelegentlich Gebrauch machen.

Bevor wir uns hier mit dem Nachweis der Korrektheit und der Abschätzung des Ressourcenbedarfs beschäftigen, wollen wir den Algorithmus anhand eines kleinen Beispiels verdeutlichen.

Eingabe für den Algorithmus ist das reduzierte OBDD für eine Boolesche Funktion $f : \{0,1\}^7 \to \{0,1\}$ aus Abbildung 8.

Nach Beendigung der Initialisierung in den Zeilen 1 und 2 sind alle Listen N_i leer, die Liste L_7 enthält den x_7-Knoten d, die Liste L_5 enthält den x_5-Knoten c, die Liste L_4, den x_4-Knoten b, die Liste L_3 die Quelle a; alle anderen Listen sind leer.

Der erste betrachtete Knoten ist der x_5-Knoten. Man erzeugt zunächst einen x_6-Knoten e zwischen c und d; für diesen Knoten ist nichts weiter zu tun. Anschließend erzeugt man einen neuen Knoten f mit Marke x_6. Weil dieser Knoten als Nachfolger die 1-Senke erhält, ist unter ihm noch ein weiterer neuer Knoten g mit Marke x_7 einzufügen.

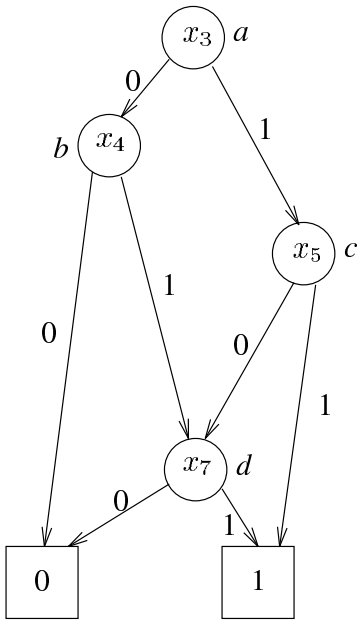


Abbildung 8: Reduziertes OBDD

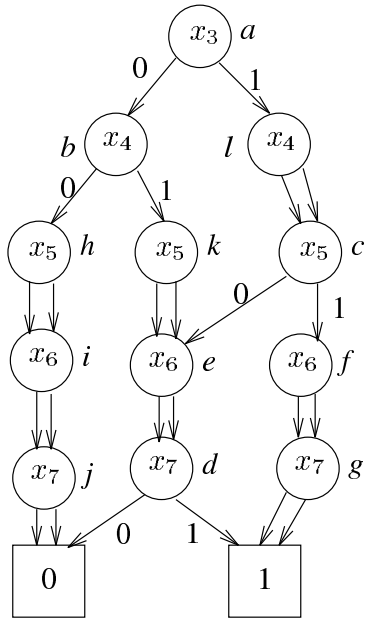


Abbildung 9: OBDD nach vollständigem Durchlauf durch alle L_i

Nach Beendigung der `for`-Schleife aus Zeile 3 hat man dann das OBDD in Abbildung 9 erzeugt. Man erkennt sofort, daß es sich dabei noch nicht um das quasi-reduzierte OBDD handeln kann. Es kommt weder ein x_1-Knoten noch ein x_2-Knoten vor; im quasi-reduzierten OBDD muß aber auf jedem Pfad von der Quelle zur Senke jede Knotenmarkierung genau einmal vorkommen (genau wie bei Orakelgraphen, vgl. Definition 2.6). Darum wird in der Zeile 14 eine „Knotenkette" vor der Quelle eingefügt. Damit ergibt sich als Endergebnis das quasi-reduzierte OBDD für f mit dem Graphen aus Abbildung 10.

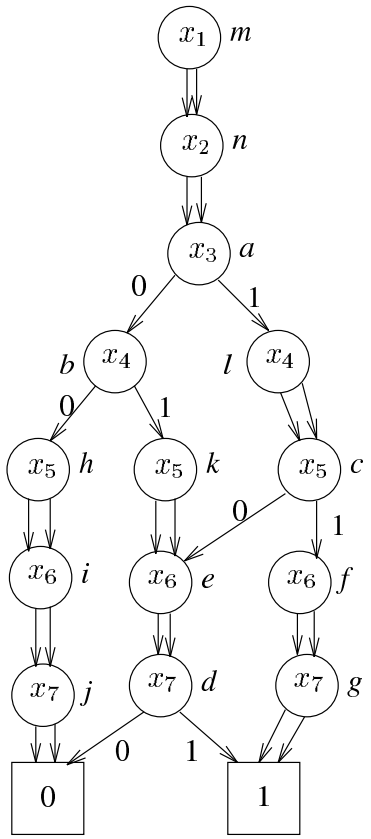


Abbildung 10: Quasi-reduziertes OBDD

In diesem Beispiel vergrößert sich das OBDD um zehn Knoten von Größe 6 auf Größe 16. Das Beispiel ist natürlich nicht typisch: viele Variablen kommen gar nicht als Markierungen im OBDD vor, außerdem hat das reduzierte OBDD extrem wenig Knoten.

Beweis von Satz 3.12: Wir wollen jetzt die Korrektheit von Algorithmus 3.13 nachweisen. Dazu weisen wir nach, daß nach vollständiger Bearbeitung der Anweisungen der `for`-Schleife (Zeilen 3 bis 12) mit dem Wert i folgende Aussage gilt: Alle Knoten mit Markierung x_j, $j \geq i$, die von Knoten, die bis zu diesem Zeitpunkt betrachtet wurden, aus erreichbar sind, können so mit einer bijektiven Abbildung ρ auf Knoten des quasi-reduzierten OBDD abgebildet werden, daß die Kantenverbindungen sich entsprechen. Das garantiert nach vollständiger Bearbeitung der Schleife, daß alle Knoten des OBDD zum quasi-reduzierten OBDD gehören und keiner fehlt, da stets alle Knoten von der Quelle aus

erreichbar sind; ausgenommen sind lediglich Knoten, die in der Zeile 14 als Knotenkette oberhalb der Quelle eingefügt werden. Die Existenz des Isomorphismus ρ garantiert, daß damit das quasi-reduzierte πOBDD berechnet ist.

Die Aussage ist für die Senken offensichtlich. Wir nehmen jetzt also gemäß Induktionsannahme an, daß die Knoten in den Ebenen $i+1$ und darunter zum quasi-reduzierten OBDD gehören, soweit sie erreichbar sind von x_{i+1}-Knoten aus. Der Algorithmus bearbeitet jetzt alle Knoten mit Marke x_i. Der Test in Zeile 6 stellt sicher, daß nur dort Knoten eingefügt werden, wo auch tatsächlich eine Lücke in der Variablenmarkierung vorhanden ist. Die Zeile 7 garantiert, daß keine Knoten eingefügt werden, die wieder verschmolzen werden könnten. Die Zeilen 9 und 10 erzeugen den benötigten neuen Knoten und bringen ihn in der passenden Liste unter, damit die nächsten Tests in Zeile 7 korrekt durchgeführt werden können. Die Zeile 11 stellt sicher, daß Bereiche des OBDD, die bislang von Knoten mit Marke x_{i+1} nicht erreichbar waren, jetzt aber von x_i-Knoten aus erreichbar sind, auch korrekt behandelt werden: es werden die noch fehlenden Knoten bis nach unten eingefügt. Damit ist die Aussage gezeigt.

Wir schätzen jetzt noch Laufzeit und Speicherplatzbedarf ab. Es werden offensichtlich nur Knoten erzeugt, niemals welche entfernt. Der Speicherplatzbedarf des Algorithmus ist also linear in der Größe des quasi-reduzierten OBDD. Der Platzbedarf aller verwendeten Datenstrukturen liegt in der gleichen Größenordnung. Außer in Zeile 7 ist der Zeitbedarf jeden Schrittes konstant; verwendet man etwa einen AVL-Baum als Datenstruktur, so kann leicht eine Gesamtlaufzeit $O(|Q| \log |Q|)$ garantiert werden, wenn Q das quasi-reduzierte OBDD bezeichnet. Damit ergibt sich gemessen in der Größe der Eingabe Zeit $O(n\,|G| \log(n\,|G|))$ und Platz $O(n\,|G|)$, wie man direkt aus dem Beweis zu Satz 3.10 folgern kann. □

Die Effizienz von Algorithmen auf BDD-Varianten sollte in der Regel in der Größe von Ein- *und* Ausgabe gemessen werden, da die Ausgabe häufig deutlich größer sein kann als die Eingabe. Betrachtet man unter dieser Perspektive den von Algorithmus 3.13 induzierten Transformationsalgorithmus für die Transformation eines πOBDD in ein πZBDD, der im ersten Schritt das Eingabe-OBDD reduziert, im zweiten Schritt mit Algorithmus 3.13 das quasi-reduzierte BDD aufbaut und im dritten Schritt dann dieses BDD zum reduzierten ZBDD reduziert, so erkennt man, daß der Algorithmus zwar polynomiell in Zeit und Platz, keineswegs aber optimal ist. In den Fällen, in denen sowohl das reduzierte πOBDD als auch das reduzierte πZBDD klein sind, das quasi-reduzierte BDD für die gleiche Funktion aber relativ groß ist, wird im zweiten Schritt ein Zwischenergebnis erzeugt, dessen Größe um einen Faktor $n+1$ größer sein kann als Ein- oder Ausgabe. Ein einfaches Beispiel ist die konstante Nullfunktion. Sowohl OBDD als auch ZBDD bestehen nur aus der 0-Senke, das quasi-reduzierte BDD enthält offenbar $n+1$ Knoten. Aber auch im betrachteten Beispiel werden offenbar unnötige Knoten erzeugt. Die Knotenkette, die aus den Knoten h, i und j besteht, kann vollständig entfernt werden, wie man sich leicht klarmacht. Die Knoten können gemäß ZBDD-Eliminationsregel in der Reihenfolge j, i und h bottom-up entfernt werden.

Betrachtet man Algorithmus 3.13 isoliert, so erkennt man leicht, daß der Algorithmus platzoptimal ist. Doch auch isoliert betrachtet ist der Algorithmus nicht makellos: Man findet ähnlich wie beim Reduktionsalgorithmus von Bryant (1986) eine Laufzeit der Größenordnung $O(|Q| \log |Q|)$. Es ist zu überlegen, ob analog zur Verbesserung der Laufzeit der OBDD-Reduktion nach Sieling und Wegener (1993) nicht auch hier eine lineare Laufzeit erreicht werden kann. Wir werden hier keine Verbesserung von Algorithmus 3.13 angeben; stattdessen wenden wir uns gleich dem Transformationsproblem insgesamt zu und lösen das Zeit- und das Platzproblem gleichzeitig.

Satz 3.14: Man kann zu einem (reduzierten) πOBDD G für eine Boolesche Funktion f das reduzierte πZBDD G' für f in Zeit $O(|G| + |G'|)$ auf Platz $O(|G| + |G'|)$ zeit- und platzoptimal berechnen.

Beweis: Wie erläutert, gibt es zwei Probleme, die für den Beweis der Aussage aus dem Weg geräumt werden müssen. Dem Platzproblem kann man beikommen, wenn man beim Einfügen von Knoten darauf achtet, keine Knoten einzufügen, die gemäß ZBDD-Eliminationsregel wieder entfernt werden können. Außerdem müssen alle Knoten, die im reduzierten OBDD zu finden sind und gemäß ZBDD-Eliminationsregel zu löschen sind, entfernt werden, bevor unter solchen Knoten neue Knoten eingefügt werden. Man erkennt, daß dann im Graphen zwischenzeitlich zwei verschiedene Typen von Knoten vorhanden sind. Zum einen gibt es OBDD-Knoten, die „normal“ behandelt werden können. Dann gibt es andererseits auch schon ZBDD-Knoten. An diesen Knoten dürfen keine Einfügungen nach inverser OBDD-Eliminationsregel mehr vorgenommen werden, da ja hier schon Knoten des reduzierten ZBDD vorliegen, für die eine Anwendung der inversen OBDD-Eliminationsregel die dargestellte Funktion ändert. Man muß in der Lage sein zu erkennen, bis wohin neue Knoten eingefügt werden dürfen und wo das Einfügen aufgrund des Erreichens eines ZBDD-Knotens zu stoppen hat. Um das bewerkstelligen zu können, notieren wir an jeder *Kante*, bis zu welcher Marke hier Einfügungen erfolgen dürfen.

Dem Rechenzeitproblem kann man begegnen, indem man das Einfügen so organisiert, daß nicht zu viele Knoten auf Äquivalenz überprüft werden müssen. Es kommt darauf an, jede Überprüfung in konstanter Zeit durchführen zu können; dabei darf allerdings der Speicherplatz für Hilfsdatenstrukturen nicht zu groß werden.

Wir lösen beide Probleme wie folgt. Die Transformation wird in zwei getrennten Phasen durchgeführt. In der ersten Phase wird das OBDD bottom-up durchlaufen. Dabei werden alle Knoten, die gemäß ZBDD-Eliminationsregel entfernt werden können, entfernt. Außerdem wird festgestellt, wo Knoten eingefügt werden müssen. Wenn mehr als ein Knoten fehlt, wie das in Abbildung 7 für $j \geq 2$ der Fall ist, wird nur der erste Knoten als fehlend notiert. Die weiter unten fehlenden Knoten werden später (in der zweiten Phase) behandelt. Solche einzufügenden Knoten werden aber nicht direkt eingefügt. Die Notwendigkeit, sie einzufügen, wird in Anforderungslisten A_i gesammelt, wobei A_i nur Anforderungen über einzufügende x_i-Knoten enthält.

Wir wollen eliminierbare Knoten entfernen, bevor durch sie Einfügeanforderungen entstehen. Das ist für die 1-Kanten solcher Knoten auch richtig; ein Knoten kann gemäß

ZBDD-Eliminationsregel entfernt werden, wenn sein Nachfolger die 0-Senke ist. Alle als Nachfolger einer solchen 1-Kante eingefügten OBDD-Knoten bilden letztlich eine Knotenkette zur 0-Senke (wie die Knoten h, i und j in Abbildung 9). Also können solche Knoten selbst wieder eliminiert werden, ihr Einfügen ist überflüssig. Das gilt für die 0-Kante aber nicht unbedingt. Eine Möglichkeit, das hier entstehende Problem zu beheben, besteht im Prinzip darin, die Einfügeanforderungen *vor* dem Entfernen eines Knotens aufzunehmen; so ein Vorgehen muß jedoch als ungünstig bezeichnet werden, weil mit jeder Einfügeanforderung der anfordernde Knoten, der Vater des neuen Knotens wird, vermerkt wird. Zum Zeitpunkt des Einfügens existiert dann dieser Knoten nicht mehr. Als Vater müssen dann alle Väter des entfernten Knotens dienen. Um mit einer Einfügeanforderung nicht mehr als einen Vater vermerken zu müssen, werden wir dem Problem auf andere Weise begegnen. Wir werden in der Initialisierung im reduzierten OBDD alle Knoten daraufhin überprüfen, ob sie gemäß ZBDD-Eliminationsregel eliminierbar sind und ob an ihrer 0-Kante eine Einfügeanforderung entsteht. Wenn das der Fall ist, werden wir dort *einen* neuen Knoten einfügen gemäß inverser OBDD-Eliminationsregel. Die Größe des OBDD kann sich dadurch höchstens verdoppeln, so daß in der Analyse in O-Notation keine Probleme erzeugt werden.

In der zweiten Phase des eigentlichen Transformationsalgorithmus werden die Einfügeanforderungslisten A_i bearbeitet. Dabei wird erkannt, wenn weitere Knoten fehlen; die entsprechenden Anforderungen werden dann eingefügt. Weil der Graph ebenenweise von oben nach unten durchlaufen wird, werden alle einzufügenden Knoten rechtzeitig in die entsprechende Liste eingetragen.

Wir verwenden ein Array `H` der Größe $|G|$ (für ein reduziertes OBDD G), um zu notieren, zu welchen Nachfolgern bereits neue Knoten eingefügt wurden. Neu eingefügte Knoten haben immer den gleichen 0- und den gleichen 1-Nachfolger. Es gibt genau dann einen neuen Knoten w mit 0- und 1-Nachfolger v, wenn `H[`v`]`$= w$ ist. Weil die Anforderungen ebenenweise bearbeitet werden, genügt *ein* solches Array, um alle zur Zeit relevanten Einträge im Speicher halten zu können. Es kann zwar mehrere neue Knoten geben, die Nachfolger v haben; es gibt jedoch immer höchstens einen neuen Knoten mit Markierung x_i, der Nachfolger v hat.

Man beachte, daß die erste Initialisierung des Arrays `H` Zeit $O(|G|)$ benötigt. Man kann es sich nicht leisten, diese Initialisierung für jede Ebene einmal durchzuführen, wenn man einen zeitoptimalen Algorithmus erhalten will. Man kann allerdings den Zeitbedarf für weitere Initialisierungen in Grenzen halten, wenn man nur genau die Einträge wieder löscht, die man auch beschrieben hat. Die Anforderungslisten werden also nicht nur einmal benötigt, um in dem aktuellen Durchgang die nötigen Einfügungen durchführen zu können. Sie werden anschließend ein zweites Mal benutzt, um den Ausgangszustand des Hilfsarrays `H` wieder herstellen zu können.

Wir verwenden in der Beschreibung des Algorithmus die Bezeichnung `v→links` für den 0-Nachfolger des Knotens v und `v→rechts` für den 1-Nachfolger des Knotens v. Mit `v-label` bezeichnen wir die Variablenmarke des Knotens v. Dementsprechend bezeichnet also `v→links-label` die Variablenmarke des 0-Nachfolgers von v. Mit `v-Marke-links`

und `v-Marke-rechts` bezeichnen wir die Marken, die an den Kanten des Knotens v notiert sind; diese Kantenmarkierungen sind nicht zu verwechseln mit der Beschriftung der Kanten mit 0 bzw. 1, die zu jedem OBDD gehören. Die hier verwendeten zusätzlichen Kantenmarkierungen werden ausschließlich in diesem Algorithmus verwendet.

Die Notation des Algorithmus folgt den gleichen Richtlinien wie das bei den Algorithmen aus dem einführenden Kapitel 2 der Fall ist. Wir wollen hier trotzdem noch einmal die Verwendung der Listenoperationen genau festlegen. Wir verwenden eine Prozedur `Insert(x,L)`, die das Datenelement x in die Liste L einträgt. Wir verwenden verschiedene Typen von Datenelementen in Listen; dabei geht aus der Notation jedesmal eindeutig hervor, welcher Datentyp eingefügt wird. Wir gehen davon aus, daß jedes Einfügen in konstanter Zeit erledigt werden kann. Der lesende Zugriff auf die Listen erfolgt in Form von `For` $x \in$`L do`. Dann soll x nacheinander alle Elemente der Liste repräsentieren, jedes Listenelement genau einmal. Wir entfernen niemals Datenelemente aus einer Liste, so daß wir uns hier keine Sorgen um die Laufzeit machen müssen. Wir setzen allerdings voraus, daß am Ende des Algorithmus eine Liste mit i Elementen in Zeit $O(i)$ geleert werden kann, um den gesamten Hilfsspeicherplatz wieder freigeben zu können.

Jeder Knoten wird identifiziert mit seinem Namen. Als Namen für Knoten verwenden wir lückenlos die Zahlen von 1 bis $|G|$; wenn das eingangs nicht gegeben ist, kann das leicht im Rahmen einer Tiefensuche erreicht werden.

Algorithmus 3.15 (πOBDD$\rightarrow$ πZBDD):

```
/* Eingabe: πOBDD G für f (o.B.d.A. π = id) */
/* Ausgabe: πZBDD G' für f (reduziert) */
1.   Reduziere G=(V,E);
2.   For v ∈ V do
3.        If v→rechts=0-Senke ∧ v-label+1<v→links-label
4.             Füge neuen Knoten w ein;
5.             w→links:=w→rechts:=v→links;
6.             w-label=v-label+1;
7.             v→links:=w;
8.   For v ∈ V do
9.        Insert(v,L_ind(v));
10.       If v→links≠Senke Then
11.            Insert(v,P(v→links));
12.            v-Marke-links:=v→links-label;
13.       Else v-Marke-links:=n+1;
14.       If v→rechts≠Senke Then
15.            Insert(v,P(v→rechts));
16.            v-Marke-rechts:=v→rechts-label;
17.       Else v-Marke-rechts:=n+1;
```

```
18.  For i:=1 to |G| do H[i]:=nil;
19.  For i:=n downto 1 do /* bottom-up-Durchlauf */
20.       For v ∈ L_i do
21.            If v→rechts=0-Senke Then
22.                 For u∈P(v) do
23.                      If u→rechts=v Then
24.                           u→rechts:=v→links;
25.                           u-Marke-rechts:=v-label;
26.                      If u→links=v Then
27.                           u→links:=v→links;
28.                           u-Marke-links:=v→label;
29.                      Entferne v;
30.            Else /* v nicht entfernt */
31.                 If   v→links≠0-Senke ∧
32.                      v-Marke-links>v-label+1
33.                 Then Insert((v,v→links,v-Marke-links),A(i+1));
34.                 If   v→rechts≠0-Senke ∧
35.                      v-Marke-rechts>v-label+1
36.                      Then Insert((v,v→rechts,v-Marke-rechts),A(i+1));
37.  For i:=2 to n do /* top-down-Durchlauf */
38.       For (v,v-Sohn,Marke)∈A(i) do
39.            If H[v-Sohn]=nil Then
40.                 H[v-Sohn]:=w;
41.                 w-label:=v-label+1;
42.                 w→links:=w→rechts:=v-Sohn;
43.                 w-Marke-links:=w-Marke-rechts:=Marke;
44.                 If Marke>w-label+1
45.                 Then Insert((w,v-Sohn,Marke),A(w-label+1);
46.            If v→links=v-Sohn Then v→links:=H[v-Sohn]
               Else v→rechts:=H[v-Sohn];
47.       For (v,v-Sohn,Marke)∈A(i) do H[v-Sohn]:=nil;
48.  If Quellenlabel≠1 ∧ Quelle≠0-Senke
49.  Then Füge Knotenkette vor Quelle ein;
```

Wir beschreiben noch einmal den Algorithmus im Detail. Der Algorithmus zerfällt grob gesprochen in vier Phasen: in den Zeilen 1 bis 18 werden die notwendigen Initialisierungen durchgeführt: das Eingabe-OBDD wird reduziert (Zeile 1). In den Zeilen 2 bis 7 werden die Einfügungen an 0-Kanten eliminierbarer Knoten durchgeführt. Die x_i-Knoten werden in L_i-Listen eingefügt, es werden zu jedem Knoten v Listen seiner Vorgänger erstellt, die Kantenmarken werden initialisiert (in den Zeilen 7–17). Schließlich wird auch noch das Array `H` initialisiert (in Zeile 18).

In den Zeilen 19 bis 36 wird der Graph bottom-up durchlaufen (in der Schleife in Zeile 19), Löschungen nach ZBDD-Eliminationsregel werden durchgeführt (Zeilen 21–29), Einfügungen gemäß inverser OBDD-Eliminationsregel werden in den Anforderungslisten `A(i)` vermerkt (Zeilen 31–36). In den Zeilen 37 bis 47 werden die Anforderungslisten abgearbeitet. Um das Mehrfacheinfügen äquivalenter Knoten zu vermeiden, wird in jeder Runde ein Hilfsarray `H[1..`$|G|$`]` verwendet: es ist `H[v]` $= w$ genau dann, wenn ein Knoten w mit Nachfolger v erzeugt wurde; sonst ist der Arrayeintrag `nil`. Die Zeile 47 bewerkstelligt die Initialisierung von `H` für die nächste Runde. In den Zeilen 48 und 49 schließlich wird nötigenfalls vor der Quelle eine Knotenkette nach inverser OBDD-Eliminationsregel eingefügt; dabei wird nicht die aktuelle Quelle betrachtet sondern die (gemessen an der Variablenordnung π) erste (oder kleinste) Markierung der Quelle, die im Laufe der Transformation aufgetreten ist. Das stellt sicher, daß nur Knoten auf den „Ebenen“ eingefügt werden, die von dem Algorithmus gar nicht betrachtet worden sind.

Offensichtlich werden nur Knoten eingefügt, die nicht wieder entfernt werden können. Ausgenommen davon sind möglicherweise die Zusatzknoten, die während der Initialisierung in den Zeilen 2 bis 7 eingefügt werden. Wir haben uns aber schon oben überlegt, daß es höchstens $|G|$ viele Knoten dieser Art geben kann. Der Platzbedarf für Knoten und Kanten beträgt also $O(|G|+|G'|)$. Die Größe aller benötigten Hilfsdatenstrukturen liegt in der gleichen Größenordnung. Offensichtlich wird keine Kante öfter als dreimal betrachtet, der Zeitbedarf liegt also ebenfalls bei $O(|G|+|G'|)$. Folglich ist der Transformationsalgorithmus optimal in Zeit und Platz. □

3.2.2 Transformation eines πZBDD in ein πOBDD

Es ist nicht weiter erstaunlich, daß die Transformation eines πZBDD in ein πOBDD ebenso effizient durchgeführt werden kann, wie die Transformation in umgekehrter Richtung. Gegenüber Algorithmus 3.15 gibt es nur geringfügige Änderungen.

Satz 3.16: Man kann zu einem (reduzierten) πZBDD G für eine Boolesche Funktion f das reduzierte πOBDD G' in Zeit $O(|G|+|G'|)$ und Platz $O(|G|+|G'|)$ zeit- und platzoptimal berechnen.

Beweis: Man beginnt damit, eine inverse Eliminationsregel für ZBDDs festzulegen. Wir gehen dabei ganz analog zur inversen OBDD-Eliminationsregel vor und verwenden ein Vorgehen, wie es aus Abbildung 11 hervorgeht.

Dann kann praktisch der Algorithmus 3.15 unverändert übernommen werden. Wir listen hier ausschließlich die Änderungen auf, die durchgeführt werden müssen, um einen Transformationsalgorithmus für die Gegenrichtung zu erhalten.

Das initialisierende Einfügen der Hilfsknoten muß jetzt gemäß inverser ZBDD-Eliminationsregel erfolgen. Außerdem betrifft es natürlich nur Knoten, die gemäß OBDD-Eliminationsregel eliminierbar sind.

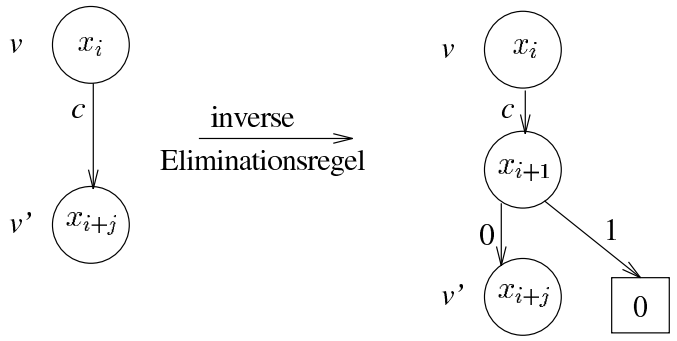


Abbildung 11: Inverse Eliminationsregel für ZBDDs

Damit ergeben sich Änderungen wie folgt:

```
3. If v→rechts=v→links ∧ v-label+1<v-links-label Then
5. w→links:=v→links; w→rechts:=0-Senke;
7. v→links:=w; v→rechts:=w;
```

In Zeile 21 ist die veränderte Eliminationsregel zu berücksichtigen. Ein Knoten kann im OBDD eliminiert werden, wenn seine beiden Nachfolger gleich sind. Es muß also heißen:

```
21. If v→rechts=v→links Then
```

Beim Notieren der Anforderungen für einzufügende Knoten in den Zeilen 31–36 ändert sich nichts: die inversen Eliminationsregeln sind zwar verschieden, die Situationen, in denen die verschiedenen inversen Eliminationsregeln angewendet werden müssen, sind aber gleich.

Die Durchführung der inversen Eliminationsregel unterscheidet sich natürlich. Darum ist Zeile 42 wie folgt zu ersetzen.

```
42. w→links:=v-Sohn; w→rechts:=0;
```

Das Einfügen fehlender Knoten vor der Quelle in Zeile 49 ist weiterhin nötig. Allerdings ist die Gestalt der einzufügenden Knotenkette eine andere: gemäß inverser Eliminationsregel sind die Knoten nur noch über die 0-Kanten miteinander verbunden, alle 1-Kanten der Knoten der Knotenkette zeigen auf die 0-Senke.

Man sieht ohne weiteres ein, daß sich der so modifizierte Algorithmus im Laufzeit- und Platzverhalten nicht von der ursprünglichen Fassung 3.15 unterscheidet. Die Transformation ist also auch in dieser Richtung zeit- und platzoptimal möglich. □

3.2.3 Transformation eines π'ZBDD in ein πZBDD

Wir beschäftigen uns jetzt mit dem Wechsel der Variablenordnung bei ZBDDs; hier liegt es nahe, sich an Algorithmus 3.4 zu orientieren, der das gleiche Problem für OBDDs platzoptimal löst. Es gibt jedoch bei ZBDDs ein entscheidendes Problem, das die Effizienz des

Ansatzes zunächst in Frage stellt. Sowohl um entscheiden zu können, welche Knoten erzeugt werden müssen, als auch um zu entscheiden, ob zwei erzeugte Knoten verschmolzen werden können, werden durch Folgen von Konstantsetzungen aus dem Eingabe-OBDD OBDDs für Subfunktionen berechnet. Dabei handelt es sich stets um Subfunktionen, die möglicherweise im Ergebnis-OBDD dargestellt werden müssen. Bei OBDDs ist offensichtlich, daß die Größe dieser Zwischen-OBDDs jeweils nicht die Größe des Eingangs-OBDD übersteigen kann. Das ist bei ZBDDs zunächst einmal nicht gegeben. Durch *eine* Konstantsetzung kann sich die Größe um den Faktor 1,5 erhöhen[24]. Damit ist klar, daß bei Durchführung einer Folge von Konstantsetzungen nicht garantiert werden kann, daß die Größe des Ergebnisses einer Konstantsetzung die Größe der Eingabe (auch in O-Notation) nicht übersteigt.

Wir untersuchen zunächst genauer, welche Funktionen in einem ZBDD für f überhaupt dargestellt werden müssen. Gegeben ist ein πZBDD G für eine Boolesche Funktion $f : \{0,1\}^n \to \{0,1\}$. Die Quelle von G sei mit x_i markiert. Wie wir uns schon in Abschnitt 2.3 überlegt haben, müssen dann die Funktionen $\overline{x_i} f_{|x_i=0}$ und $\overline{x_i} f_{|x_i=1}$ dargestellt werden.

Bei der π'ZBDD$\to$ πZBDD-Transformation stellt sich also gar nicht direkt das Problem, ZBDDs für Subfunktionen von f zu berechnen. Es ist offenbar so, daß stets noch zusätzlich eine $\wedge$-Verknüpfung mit $\overline{x_i}$ durchzuführen ist. Das macht die Situation auf den ersten Blick nicht besser. Man kann die benötigten ZBDDs berechnen, indem man zunächst eine Konstantsetzung durchführt (also ein ZBDD für $f_{|x_i=c}$ berechnet) und das Ergebnis anschließend mittels ZBDD-Synthesealgorithmus mit einem ZBDD für $\overline{x_i}$ mit $\wedge$ verknüpft. Die Anwendung der Größenschranken für die ZBDD-Synthese ergibt, daß das Ergebnis dann Größe $O(n\,|G|)$ haben kann, wenn G das Eingabe-ZBDD bezeichnet. Tatsächlich kann man aber einen besseren Algorithmus angeben, der beide Schritte integriert. Dazu überlegen wir uns zunächst, was die Ausführung der beiden Schritte nacheinander bewirkt. Bei der Berechnung des ZBDD für $f_{|x_i=c}$ wird zunächst in G auf allen Pfaden von der Quelle zu einer Senke jeweils ein x_i-Knoten gemäß inverser ZBDD-Eliminationsregel eingefügt, wenn nicht schon einer vorhanden ist. Das ändert die dargestellte Funktion nicht. Dann werden alle $\overline{c}$-Kanten, die x_i-Knoten verlassen, auf den c-Nachfolger des Knotens umgesetzt. Anschließend kann der so berechnete Graph noch reduziert werden. Man beachte, daß im Gegensatz zu OBDDs die x_i-Knoten danach nicht notwendig eliminiert werden können. Die anschließende Synthese mit dem ZBDD für $\overline{x_i}$ sorgt dann aber dafür, daß alle x_i-Knoten entfernt werden können. Das ergibt sich zum einen aus dem Umstand, daß bei allen x_i-Knoten die zwei ausgehenden Kanten auf den gleichen Knoten verweisen und das reduzierte ZBDD für $\overline{x_i}$ keinen x_i-Knoten enthält (Abbildung 12).

[24]Einen Beweis dieser Tatsache findet man bei Schröer und Wegener (1994).

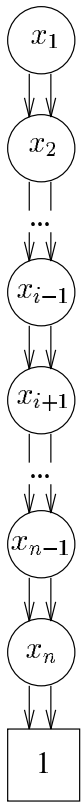


Abbildung 12: Reduziertes ZBDD für $\overline{x_i}$

Zum anderen kann man sehen, daß ein reduziertes ZBDD für $\overline{x_i} f_{|x_i=c}$ typischerweise als c-Nachfolger eines x_i-Knotens auftaucht, an dem die Funktion f repräsentiert wird. Damit ist klar, daß wegen der read-once Eigenschaft kein zweiter x_i-Knoten auftauchen darf. Diese Überlegungen motivieren folgenden Algorithmus, dessen Korrektheit natürlich noch nachgewiesen werden muß. Eingabe ist ein πZBDD G für f, wobei wir o. B. d. A. davon ausgehen, daß G bereits reduziert ist.

Algorithmus 3.17 (πZBDD für $f \rightarrow \pi$ZBDD für $\overline{x_i} f_{|x_i=c}$):
/* Eingabe: πZBDD G für f */
/* Ausgabe: reduziertes πZBDD für $\overline{x_i} f_{|x_i=c}$ */

1. Erweitere G so um x_i-Knoten, daß auf jedem Pfad von der Quelle zu einer Senke ein x_i-Knoten vorkommt.
2. Ersetze Zeiger auf x_i-Knoten durch Zeiger auf den c-Nachfolger des Knotens.
3. Reduziere das Ergebnis.

Zunächst halten wir fest, daß das Ergebnis wiederum ein reduziertes πZBDD ist: Es liegt ein πZBDD vor, weil keiner der Schritte die syntaktische Korrektheit von G verletzt und die Variablenordnung beachtet bleibt. Der dritte Schritt garantiert, daß das Ergebnis reduziert vorliegt. Wie wir schon bei den Überlegungen zu Konstantsetzungen in ZBDDs gesehen haben, ändert der erste Schritt die dargestellte Funktion nicht. Die Korrektheit

des zweiten Schrittes kann man sich wie folgt klar machen. Wir unterscheiden nach der Markierung x_j der Quelle von G zwei Fälle: Ist $j \geq i$, steht die Markierung der Quelle also gemäß Permutation der Variablen nicht vor der betrachteten Variablen x_i, so ist die Quelle des ZBDD nach Durchführung des ersten Schrittes mit x_i markiert. Dann stellt der c-Nachfolger der Quelle, der im zweiten Schritt Quelle des Ergebnisses wird, die Funktion $\overline{x_i} \cdot f_{|x_i=c}$ dar, wie wir uns schon in Abschnitt 2.3 klargemacht haben. Sei jetzt also $j < i$, die Markierung der Quelle steht also gemäß Permutation der Variablen vor der betrachteten Variablen x_i. Die Ausführung des ersten Schrittes sei abgeschlossen, die Quelle sei weiterhin mit x_j markiert. Wir nehmen per Induktionsannahme an, daß nach Abschluß des zweiten Schrittes der 0-Nachfolger der Quelle die Funktion $\overline{x_i} \cdot (\overline{x_j} \cdot f_{|x_j=0})_{|x_i=c}$ und der 1-Nachfolger der Quelle die Funktion $\overline{x_i} \cdot (\overline{x_j} \cdot f_{|x_j=1})_{|x_i=c}$ darstellen. Das ist gerechtfertigt: Die Induktion wird ebenenweise bottom-up geführt, spätestens an den Senken greift der erste Fall; der Induktionsanfang ist somit gesichert. Wir haben uns in Abschnitt 2.3 überlegt, daß die Quelle eines ZBDD mit Markierung x_j die Funktion $g \vee x_j \cdot h_{|x_j=0}$ darstellt, wenn g die durch den 0-Nachfolger und h die durch den 1-Nachfolger dargestellten Funktionen bezeichnen. Damit wird also folgende Funktion an der Quelle dargestellt:

$$\begin{aligned}
& \overline{x_i} \left(\overline{x_j} \cdot f_{|x_j=0}\right)_{|x_i=c} \vee x_j \cdot \left(\overline{x_i} \cdot (\overline{x_j} \cdot f_{|x_j=1})_{|x_i=c}\right)_{|x_j=0} \\
= \; & \overline{x_i} \cdot \overline{x_j} \cdot f_{|x_j=0,x_i=c} \vee \overline{x_i} \cdot x_j \cdot f_{|x_j=1,x_i=c} \\
= \; & \overline{x_i} \left(\overline{x_j} \cdot f_{|x_j=0} \vee x_j \cdot f_{|x_j=1}\right)_{|x_i=c} \\
= \; & \overline{x_i} \cdot f_{|x_i=c}.
\end{aligned}$$

Also wird wie behauptet insgesamt die Funktion $\overline{x_i} \cdot f_{|x_i=c}$ dargestellt.

Wir schätzen die Laufzeit und den Platzbedarf des Algorithmus ab; dabei messen wir nur in der Größe der Eingabe. Der erste Schritt kann die Größe von G offenbar höchstens verdoppeln: selbst wenn G aus einem vollständigen binären Baum der Tiefe i besteht (was nicht vorkommen kann, weil G reduziert ist), können nicht mehr als 2^{i-1} Einfügungen erforderlich sein, G enthält aber $2 + \sum_{0 \leq j < i-1} 2^j = 2^{i-1} + 1$ Knoten. Das Einfügen kann leicht auf linearem Platz in linearer Zeit durchgeführt werden. Der zweite Schritt entfernt alle x_i-Knoten, insbesondere werden alle neu eingefügten Knoten wieder entfernt. Der dritte Schritt schließlich kann in Linearzeit auf linearem Platz ausgeführt werden, das Ergebnis ist insgesamt höchstens so groß wie G. Algorithmus 3.17 kann also insgesamt in Zeit $O(|G|)$ auf Platz $O(|G|)$ ausgeführt werden.

Satz 3.18: Die Transformation eines π'ZBDD G in ein πZBDD H kann auf Platz $O(|G| + |H|)$ in Zeit $O(|G|\,|H| \log |H|)$ durchgeführt werden, wenn sowohl G als auch H mindestens cn Knoten für eine Konstante c enthalten.

Beweis: Wir übertragen für den Beweis Algorithmus 3.4 auf ZBDDs und benutzen Algorithmus 3.17 als Subroutine. Wir beginnen damit, eine Neuformulierung von Algorithmus 3.4 anzugeben, in der die Gemeinsamkeiten stärker als die Unterschiede herausgearbeitet werden.

Algorithmus 3.19 (π'ZBDD$\rightarrow$ πZBDD):

```
/* Eingabe: π'ZBDD G für f */
/* Ausgabe: πZBDD H für f (o.B.d.A. π = id) */
1.   for i:=1 to n do L_i:=nil;
2.   B:=nil;
3.   Erzeuge neuen x_1-Knoten v als Quelle von H;
4.   Erzeuge zwei verschiedene Senken für H;
5.   Insert(v, L_1);
6.   for i:=1 to n do
7.        for v ∈ L_i do
8.             Erzeuge reduziertes π'ZBDD G_v für die
               an v repräsentierte Funktion;
9.             w:=Lookup(G_v,B);
10.            if w ist Knoten aus H
11.            then verschmilz v und w;
12.            else Insert((v, G_v),B);
13.       for v ∈ L_i do
14.            for c=0 to 1 do
15.                 Erzeuge reduziertes π'ZBDD G_v für die
                    an v repräsentierte Funktion;
16.                 if G_v ist c-Senke then w:=c-Senke;
17.                 else
18.                      Erzeuge neuen Knoten w mit kleinster
                         Markierung x_j, die in G_v vorkommt;
19.                      Insert(x_j,L_j);
20.                 Mache w zum c-Nachfolger von v;
```

Wenn in Algorithmus 3.4 ein Knoten betrachtet wird, so kann dafür die Berechnung eines OBDD aus dem Eingabe-OBDD mit höchstens n Konstantsetzungen erforderlich sein. Bei OBDDs kann man in *einem* Durchgang *alle* Konstantsetzungen durchführen, so daß man insgesamt bei linearer Rechenzeit bleibt. Das ist bei ZBDDs anders. Wir haben uns schon oben ausführlich überlegt, daß in ZBDDs andere Funktionen repräsentiert werden müssen. Wenden wir für die Transformation mit Algorithmus 3.19 als Subroutine Algorithmus 3.17 bis zu n-mal an, so kann sich eine Gesamtrechenzeit von $O(n\,|G|)$ ergeben, der Platzbedarf bleibt bei $O(|G|)$. Wenn man alle zu behandelnden Variablen auf einmal betrachtet, kommt man bei naiver Durchführung auch nicht weiter. Führt man erst alle Einfügungen durch und entfernt anschließend die Knoten, erhält man im schlimmsten Fall das quasi-reduzierte ZBDD für f, so daß sich $O(n\,|G|)$ in Zeit *und* Platz ergeben. Aber man sollte sorgfältiger vorgehen: es ist nicht notwendig, *erst* alle Einfügungen und *danach* alle Löschungen durchzuführen. Man kann beides so integrieren, daß in O-Notation kein zusätzlicher Speicherplatz benötigt wird.

Wir beschreiben jetzt auf recht hohem Abstraktionsniveau den Algorithmus, den wir als Subroutine einsetzen wollen. Dabei setzen wir voraus, daß G beide Senken enthält. Wenn das nicht der Fall ist, muß die fehlende Senke ergänzt werden. Außerdem fügen wir einen

neuen „virtuellen Knoten“ als neue Quelle mit Markierung x_0 ein, um vor der eigentlichen Quelle einzufügende Knoten nicht separat behandeln zu müssen. Danach werden wir uns Details, vor allem die benötigten Datenstrukturen, überlegen. Schließlich kümmern wir uns um Korrektheit, Laufzeit und Speicherplatz.

Für die Notation des Algorithmus werden wir eine Umbenennung vornehmen, um jede Situation, in der Algorithmus 3.19 von Algorithmus 3.20 Gebrauch macht, erfassen zu können, andererseits aber keine allzu komplizierten Schreibweisen benutzen zu müssen. Wir gehen insgesamt von der im folgenden beschriebenen Situation aus; wie eine geeignete Umbenenung vorgenommen werden kann (die in einer Implementierung natürlich unnötig ist und unterbleibt), ist offensichtlich.

Zu einer Funktion g über den Variablen $y_1, \ldots, y_m$ $(m \leq n)$, die durch das reduzierte π'ZBDD für g gegeben ist, einer Teilmenge $C = \{y_{i_1}, y_{i_2}, \ldots, y_{i_r}\} \subseteq \{y_1, \ldots, y_m\}$ mit $r \leq m$ und $c_{i_j} \in \{0, 1\}$ für $1 \leq j \leq r$ soll das reduzierte π'ZBDD für die Funktion

$$\overline{y_{i_1}} \cdot \overline{y_{i_2}} \cdot \ldots \cdot \overline{y_{i_r}} g_{|y_{i_1}=c_{i_1},\ldots,y_{i_r}=c_{i_r}}$$

berechnet werden.

Algorithmus 3.20: Durchlaufe alle Knoten v von G (außer den Senken) in umgekehrt topologischer Reihenfolge (also etwa levelweise bottom-up). Sei y_k die Markierung von v. Für den aktuellen Knoten v führe folgende Schritte aus:

1. Falls es einen Index j zwischen 1 und r gibt, so daß $y_{i_j} = y_k$ gilt, ersetze alle Zeiger auf v durch Zeiger auf den c-Nachfolger von v.

2. Falls es keinen solchen Index j wie im ersten Fall gibt, es aber einen Index j' zwischen 1 und r gibt, so daß die Markierung des c-Nachfolgers von v gemäß der Variablenordnung π' hinter $y_{i_{j'}}$ steht und $c_{i_{j'}} = 1$ gilt, so mache die 0-Senke zum neuen c-Nachfolger von v.

Reduziere zum Schluß das Ergebnis.

Um den Algorithmus effizient (in unserem Sinne) durchführen zu können, müssen beide Schritte für jeden beliebigen Knoten in Zeit $O(z)$ durchführbar sein, wenn z die Anzahl explizit betrachteter Zeiger angibt; damit sind die Zeiger auf den c-Nachfolger und die Zeiger auf den betrachteten Knoten gemeint. Dafür müssen insbesondere die Tests in konstanter Zeit durchführbar sein. Das ist unter Zuhilfenahme zweier Hilfsarrays der Größe $O(m)$ möglich; damit wird der von Algorithmus 3.20 verwendete zusätzliche Speicherplatz nicht zu groß. In einem Array halten wir für jede Variable fest, durch welche Konstante sie zu ersetzen ist; wenn keine Ersetzung vorgesehen ist, vermerken wir hier den Wert 2. Mit Hilfe dieses Arrays kann der Test des ersten Schrittes leicht in konstanter Zeit durchgeführt werden. In einem zweiten Array halten wir für jede Variable fest, für die im ersten Array der Wert 2 vermerkt ist, welches die gemäß Variablenordnung π' maximale Variable ist, die durch die Konstante 1 ersetzt werden soll und nicht hinter dieser Variablen liegt; für Variablen, denen im ersten Array ein Wert ungleich 2 zugewiesen ist, kann hier

ein beliebiger Eintrag stehen. Damit ist klar, daß auch der Test des zweiten Schrittes in konstanter Zeit durchgeführt werden kann.

Um die Zeiger auf einen Knoten v effizient ändern zu können, müssen zu jedem Knoten v aus G seine Vorgänger effizient auffindbar sein. Dafür halten wir zu jedem Knoten v eine Liste seiner Vorgänger bereit. Die Initialisierung der Listen kann in Linearzeit durchgeführt werden. Weil der Algorithmus immer auf das Eingabe-ZBDD für den Transformationsalgorithmus angewendet wird, brauchen diese Listen im Rahmen der Transformation nur ein einziges Mal berechnet zu werden.

Die Initialisierung des zweiten Arrays ist nicht ganz unproblematisch in Zeit $O(m)$ durchzuführen. Im Kern genügt es allerdings, die Knotenmenge zu sortieren, wobei als Sortierkriterium im wesentlichen nur die Variablenmarkierungen verwendet werden. Darum kann man durch Verwenden einer Bucket-Sort-Technik das Sortieren durchaus in Linearzeit erledigen; es ist aber auch möglich, einen etwas direkteren und im Detail etwas weniger aufwendigeren Weg zu beschreiten. Wir gehen dafür davon aus, daß die konstantzusetzenden Variablen gemäß der Permutation π' sortiert vorliegen. Diese Voraussetzung ist nicht unrealistisch: die konstantzusetzenden Variablen werden ermittelt, indem zu einem Knoten ein Pfad von der Quelle aus gesucht wird. Damit liegen die Variablen sortiert gemäß der Permutation π vor. Weil π und π' beide bekannt sind, kann die Ordnung leicht in Zeit $O(m)$ angepaßt werden. Durch die gegebene Ordnung ist es im Rahmen eines Durchlaufs durch die Variablen leicht, das Array komplett zu füllen. Da dabei jeder Arrayeintrag nur genau einmal geschrieben und nicht gelesen wird, kann die Initialisierung in Zeit $O(m)$ durchgeführt werden.

Jetzt ist insgesamt gewährleistet, daß wir tatsächlich nicht mehr Zeit $O(|H| + n)$ für die Durchführung von Algorithmus 3.20 brauchen. Wir werden den Summanden n in der Analyse von hier ab nicht mehr explizit erwähnen und merken uns, daß die Ergebnisse nur stimmen, wenn Eingabe und Ausgabe der Transformation jeweils mindestens Größe cn für eine Konstante $c > 0$ haben. In den anderen Fällen sind die Graphen aber auch so klein, daß die größere Laufzeit zu verschmerzen ist. Außerdem muß dann in der Eingabe die Länge der Permutation mitgezählt werden, so daß zumindest die Länge der Eingabe immer mindestens n beträgt. Es kann also mit voller Berechtigung gesagt werden, daß der Algorithmus lineare Laufzeit hat.

Wir kommen jetzt zum Nachweis der Korrektheit. Wir betrachten den aktuellen Knoten v mit Markierung x_k. Die Senken brauchen nicht betrachtet zu werden und werden also korrekt behandelt. Wir können jetzt per Induktionsannahme annehmen, daß unterhalb von v alle nötigen Schritte korrekt durchgeführt wurden. Das Einfügen von Knoten kann nur erforderlich sein, wenn die entsprechenden Tests im zweiten Schritt positiv ausfallen wegen der gegebenen Variablenordnung. Ein Knoten, der als neuer c-Nachfolger von v eingefügt wird, erhält als 1-Nachfolger die 0-Senke und als 0-Nachfolger den alten c-Nachfolger von v. Trägt der neue Knoten die Marke x_l und ist $c_l = 0$, so erhält v als neuen 0-Nachfolger den 0-Nachfolger des neuen Knotens, das ist genau sein alter 0-Nachfolger. Der Schritt kann also ganz ausgelassen werden. Ist hingegen $c_l = 1$, so wird neuer c-Nachfolger von v der 1-Nachfolger des neuen Knotens also die 0-Senke. Damit ist klar,

daß der zweite Schritt korrekt ist. Der erste Schritt entspricht genau dem zweiten Schritt von Algorithmus 3.17. Zusammen mit der abschließenden Reduktion ist so die Korrektheit von Algorithmus 3.20 gesichert. Damit ist klar, daß Algorithmus 3.19 korrekt ist und die behaupteten Eigenschaften hat. □

Wir werden den so konstruierten Transformationsalgorithmus in Abschnitt 3.4.5 noch einmal betrachten, wenn wir einen effizienten Algorithmus für die Transformation eines π'OFDDs in ein πOFDD suchen.

Abschließend wollen wir noch erwähnen, daß die in Satz 3.18 geforderte Mindestanzahl von cn Knoten in Ein- und Ausgabe für die Abschätzung des Ressourcenbedarfs nötig ist, weil der Algorithmus in jedem Fall Aktionen für alle n Variablen durchführt. Wir sind damit wieder in dem Grenzbereich sehr kleiner Graphen, den wir schon im Anschluß an Algorithmus 2.17 besprochen haben. Wir wollen hier noch einmal festhalten, daß die Laufzeit von Algorithmus 3.19 in jedem Fall linear genannt werden kann, weil die Eingabe immer Länge $O(n + |G|)$ hat, da die Zielpermutation jedenfalls angegeben sein muß.

3.3 Transformationen mit graphgesteuerten BDDs

Im Zusammenhang mit graphgesteuerten BDDs wollen wir uns fünf verschiedenen Gruppen von Transformationsproblemen zuwenden. Dabei werden wir zum Teil mit Negativresultaten konfrontiert, zum Teil werden wir Ergebnisse von Sieling (1995) vorstellen, andererseits können wir hier auch neue Transformationsalgorithmen kennenlernen, wobei wir die Transformationsalgorithmen von Savický und Wegener (1994) als wirkungsvolles Werkzeug einsetzen werden.

Wir wollen im wesentlichen die verschiedenen Transformationen zwischen OBDDs, LBDDs und WBDDs vorstellen. Wegen ihrer engen Verwandtschaft nehmen wir zusätzlich FBDDs hinzu: OBDDs fordern eine strenge Ordnung der Variablen; graphgesteuerte BDDs erlauben, diese Ordnung zu lockern, indem für verschiedene Variablenbelegungen verschiedene Ordnungen zugelassen werden. FBDDs schließlich stellen überhaupt keine Anforderung an die Ordnung der Variablen. Graphgesteuerte BDDs stellen also in diesem Sinne die Mitte zwischen der extremen Ordnung der OBDDs und der fehlenden Ordnung der FBDDs dar. Weil graphgesteuerte BDDs schon sehr große Freiheiten bei der Aufstellung von Variablenordnungen lassen, macht es Sinn, graphgesteuerte BDDs zu betrachten, die von eingeschränkten Formen von Orakelgraphen gesteuert werden.

3.3.1 Transformation eines FBDD in ein graphgesteuertes BDD

Es soll ein FBDD in ein graphgesteuertes BDD transformiert werden. Der zu beachtende Orakelgraph G^O ist dabei gegeben[25].

[25] Wenn keine Ordnung gegeben ist, kann zu einem FBDD leicht ein passender Orakelgraph konstruiert werden. Ein entsprechender Algorithmus ist bei Sieling (1995) zu finden. Diese Operation ist aber keine Transformation im Sinne dieser Arbeit.

Sieling (1995) zeigt, daß es für dieses Problem genau dann einen deterministischen Polynomialzeitalgorithmus gibt, wenn es einen solchen Algorithmus für den Äquivalenztest von FBDDs gibt. Zum Beweis modifiziert er zum einen Algorithmus 3.4 von Savický und Wegener so, daß er statt auf OBDDs auf FBDDs operiert. Mit Hilfe eines deterministischen Polynomialzeitäquivalenztests für FBDDs erhält er so einen deterministischen Polynomialzeitalgorithmus für die Transformation eines FBDD in ein graphgesteuertes BDD. Zum anderen kann man leicht einen solchen Transformationsalgorithmus für einen Äquivalenztest verwenden. Sollen zwei FBDDs auf Äquivalenz getestet werden, wendet man zunächst auf beide FBDDs die LBDD-Reduktionsregeln an. Dann berechnet man zu einem der beiden FBDDs in polynomieller Zeit einen passenden Orakelgraphen. Anschließend benutzt man den polynomiellen Transformationsalgorithmus; man bricht die Transformation ab, wenn das Ergebnis größer wird als das erste LBDD, da dann klar ist, daß keine Äquivalenz besteht.

Das Verfahren erfordert exponentielle Laufzeiten, weil Algorithmus 3.4 mit seiner Laufzeit wesentlich davon abhängt, daß zwei reduzierte πOBDDs effizient daraufhin getestet werden können, ob sie die gleiche Funktion repräsentieren. Das ist bei FBDDs nicht der Fall, so daß man exponentielle Laufzeiten in Kauf nehmen muß, solange man einen deterministischen Algorithmus fordert und es keinen deterministischen Polynomialzeitalgorithmus für den FBDD-Äquivalenztest gibt[26].

Wir werden hier auf den Beweis nicht im Detail eingehen, da der Transformationsalgorithmus für die Transformation eines OBDD in ein graphgesteuertes BDD ähnlich strukturiert ist (Abschnitt 3.3.4) und ein positives Resultat darstellt.

3.3.2 Transformationen zwischen LBDDs und WBDDs

Der Vollständigkeit halber wollen wir hier kurz erwähnen, wie man ein gegebenes LBDD in ein WBDD bzw. umgekehrt transformiert, wobei jeweils der zugehörige Orakelgraph beibehalten wird. Die Ergebnisse können ausführlich bei Sieling (1995) nachgelesen werden.

Man transformiert ein G^O-gesteuertes WBDD in ein G^O-gesteuertes LBDD, indem man auf das gegebene WBDD den LBDD-Reduktionsalgorithmus anwendet. Die Transformation läßt sich also auf linearem Platz in linearer Zeit bewerkstelligen.

Für die umgekehrte Transformationsrichtung genügt es, wie bei der LBDD-Synthese einen Produktgraphen aus dem gegebenen LBDD, dem Orakelgraphen und der 0-Senke zu konstruieren. Wendet man auf diesen Graphen den WBDD-Reduktionsalgorithmus an, so erhält man das reduzierte WBDD. Ist G das G^O-gesteuerte LBDD, so sind Rechenzeit und Speicherplatz in $O\left(|G|\left|G^O\right|\right)$.

[26] Es sei hier nochmals an den probabilistischen Algorithmus von Blum, Chandra und Wegmann (1980) erinnert, der lineare Laufzeit hat.

3.3.3 Transformation eines graphgesteuerten BDD in ein πOBDD

Die Transformation graphgesteuerter BDDs in OBDDs haben wir schon in Abschnitt 3.1 ausführlich besprochen. Offensichtlich ist jedes graphgesteuerte BDD auch ein FBDD, wobei man den Orakelgraphen ignoriert. Damit kann der FBDD→OBDD-Transformationsalgorithmus angewendet werden. Eine Ausnutzung der im Orakelgraphen steckenden Information zur Verbesserung der Laufzeit darf dabei im allgemeinen nicht erwartet werden, da es leicht in Linearzeit möglich ist, zu einem beliebigen FBDD einen geeigneten Orakelgraphen zu konstruieren. Man kann allerdings bessere Algorithmen erwarten und auch erhalten, wenn der Orakelgraph eine besondere Gestalt hat. Einen solchen verbesserten Algorithmus wollen wir jetzt angeben.

Wir suchen Transformationsalgorithmen für die Transformation eines graphgesteuerten BDD in ein OBDD, die effizienter sind als die allgemeine Transformation eines FBDD in ein OBDD. Dazu muß der Orakelgraph eine besondere Gestalt haben, die gewinnbringend ausgenutzt werden kann. Wir definieren zunächst, was man sich darunter konkret vorzustellen hat.

Definition 3.21: Ein G^O-gesteuertes BDD heißt *listengesteuert*, wenn der Orakelgraph nach Entfernen der Senke und Ersetzen von Mehrfachkanten durch Einfachkanten eine Liste ist. G^O heißt dann *Listenorakel*.
Ein G^O-gesteuertes BDD heißt *baumgesteuert*, wenn der Orakelgraph nach Entfernen der Senke und Ersetzen von Mehrfachkanten durch Einfachkanten ein Baum ist. G^O heißt dann *Baumorakel*.

Offenbar sind nach Definition 3.21 OBDDs listengesteuerte BDDs.

Für baumgesteuerte BDDs kann eine platzoptimale Transformation in ein OBDD mit polynomieller Rechenzeit gefunden werden; das ist gegenüber der allgemeinen Transformation eines FBDD in ein OBDD eine erhebliche Verbesserung, da in praktischen Anwendungen häufig Platz eine knappe Ressource darstellt.

Der folgende Algorithmus ist eine beinahe wörtliche Wiedergabe des Algorithmus 3.4 für das Umordnen von OBDDs von Savický und Wegener. Die im Algorithmus nicht ausgeführten Details unterscheiden sich allerdings erheblich. Der Algorithmus wird sich bei näherer Betrachtung als Verallgemeinerung von Algorithmus 3.4 entpuppen.

Wir gehen davon aus, daß alle Kanten auch in umgekehrter Richtung abgespeichert sind. Diese Voraussetzung, die leicht im Rahmen eines Graphdurchlaufs hergestellt werden kann, stellt sicher, daß von jedem Knoten aus in Zeit $O(n)$ ein Pfad zurück zur Quelle konstruiert werden kann.

Algorithmus 3.22 (baumgesteuertes BDD→ πOBDD):

```
/* Eingabe: baumgesteuertes BDD G für f */
/* Ausgabe: πOBDD H für f (o.B.d.A. π = id) */
1.   For i:=1 to n do L_i:=nil;
2.   j:=min{1 ≤ i ≤ n | x_i nicht redundant};
3.   L_j:={x_j};
```

```
4.   For i:=j to n do
5.        For x ∈ L_i do
6.             y:=Nachfolger von x in L_i
7.             If x und y äquivalent then
8.                  Entferne y aus L_i;
9.                  Ersetze alle Kanten auf y durch Kanten auf x;
10.       For x∈ L_i do
11.            For c:=0 to 1 do
12.                 Bestimme y=c-Nachfolger von x mit Label x_k;
13.                 If y ist Senke
14.                 Then füge entsprechende Kante ein;
15.                 Else füge y in L_k ein; Füge Kante von x nach y ein;
```

Die Übereinstimmung mit Algorithmus 3.4 ist so groß, daß es ausreicht, die Schritte 2, 7 und 12 näher zu erläutern. Wir tragen zunächst zusammen, *was* in den drei Schritten zu tun ist und klären dann anschließend, *wie* genau das platzoptimal getan werden kann.

In Schritt 2 muß offenbar (höchstens n-mal) ein Redundanztest für ein baumgesteuertes BDD durchgeführt werden. Um diesen Test zu implementieren, genügt es, Ersetzung von Variablen durch Konstante und einen Äquivalenztest für baumgesteuerte BDDs geeignet zu kombinieren. Ist ein baumgesteuertes BDD für die Funktion f daraufhin zu testen, ob die Variable x_i redundant ist, berechnet man zunächst mittels Konstantsetzung baumgesteuerte BDDs für $f_{|x_i=0}$ und $f_{|x_i=1}$. Das können wir hier — im Gegensatz zum allgemeinen Fall — analog zu FBDDs und OBDDs in Linearzeit mit linearem Platzaufwand erledigen. Dazu erlauben wir, bei einer Konstantsetzung den zugehörigen Orakelgraphen ebenfalls zu modifizieren. Die Konstantsetzungen werden nicht nur im BDD selbst sondern simultan auch im Orakelgraphen durchgeführt. Man erhält also im allgemeinen baumgesteuerte BDDs zu verschiedenen Baumorakeln. Die so berechneten Ergebnis-BDDs sind dann einem Äquivalenztest zu unterziehen. Genau dann, wenn beide BDDs äquivalent sind, ist die Variable x_i redundant. Wir haben also das im allgemeinen für graphgesteuerte BDDs schwierige Problem der Ersetzung von Variablen durch Konstante „gelöst"; dafür müssen wir jetzt zwei baumgesteuerte BDDs, die von verschiedenen Baumorakeln gesteuert sein können, effizient auf Äquivalenz testen. Dem Problem wenden wir uns erst etwas später zu.

In Schritt 7 soll für zwei Knoten x und y entschieden werden, ob sie die gleiche Funktion repräsentieren. Dazu berechnen wir aus dem baumgesteuerten BDD G durch Konstantsetzung einiger Variablen zwei baumgesteuerte BDDs G' und G''. Wir erlauben dazu wie oben, im Orakelgraphen ebenfalls Konstantsetzungen durchzuführen. Dabei wählen wir die Konstantsetzungen jeweils so aus, daß sich in G Pfade zu x bzw. y ergeben. Die Berechnung der Pfade und der zugehörigen Konstantsetzungen ist effizient möglich, weil wir Kanten zu den Vorgängern zur Verfügung haben. Die Konstantsetzungen können dann durchgeführt werden, wie wir das schon im Zuge der Erläuterung des zweiten Schritts getan haben. Auch hier ist anschließend ein Äquivalenztest für zwei baumgesteuerte BDDs durchzuführen, die im allgemeinen von verschiedenen Baumorakeln gesteuert werden.

In Schritt 12 schließlich soll der c-Nachfolger (für $c \in \{0,1\}$) eines Knotens x berechnet werden. Es soll also entschieden werden, ob der c-Nachfolger eine Senke ist oder welche Variablenmarkierung er tragen muß. Dazu konstruieren wir wiederum ein baumgesteuertes BDD G' aus G durch Konstantsetzung einiger Variablen — auch hier simultan im BDD und im Orakelgraphen durchgeführt —, wobei die Konstantsetzungen so gewählt werden, daß sich in G ein Pfad zu x ergibt. Zusätzlich wird $x_i := c$ gesetzt. Für das so berechnete BDD kann man leicht prüfen, ob es eine Konstante repräsentiert. Wenn das nicht der Fall ist, kann wie in Schritt 2 der minimale Index $k > i$ gefunden werden, für den die zu repräsentierende Funktion essentiell von x_k abhängt.

In allen drei erläuterten Schritten wird also ein Äquivalenztest für zwei baumgesteuerte BDDs benötigt. In Sieling (1995) findet man einen Äquivalenztest für ein baumgesteuertes BDD G mit Orakel T und ein FBDD F, der in Zeit $O(|T|\,|G|\,|F|)$ mit Platz $O(|G|\,|F|)$ auskommt. Weil wir einen platzoptimalen Algorithmus angeben wollen, ist dieser Äquivalenztest hier unbrauchbar. Es ist jedoch nicht schwierig, für zwei baumgesteuerte BDDs einen besseren Algorithmus zu konstruieren. Für die Analyse benötigen wir den Begriff der Breite eines BDDs, den wir kurz formal festlegen wollen.

Definition 3.23: Zu einem BDD G definieren wir die *Breite* b wie folgt: Es bezeichne c_i die Anzahl aller x_i-Knoten in G mit $1 \leq i \leq n$. Es ist b das Maximum aller c_i.

Die Definition 3.23 läßt sich auf alle eingeführten BDD-Varianten anwenden. Insbesondere ist das auch bei Orakelgraphen möglich.

Eingabe sei ein baumgesteuertes BDD G mit Baumorakel T und ein baumgesteuertes BDD G' mit Baumorakel T'. Wir interpretieren im Baumorakel T Mehrfachkanten als Einfachkanten. Dann gibt es höchstens b Pfade von der Quelle zur Senke in T, wenn b die Breite von T bezeichnet. Für jeden dieser Pfade P berechnen wir eine minimale Konstantsetzung von Variablen, so daß gerade P durchlaufen wird. Nach dieser Konstantsetzung ist T eine Liste und G ein OBDD. Für das Baumorakel T' und das T'-gesteuerte BDD G' gehen wir ganz analog vor. Wir erhalten dann schließlich zwei OBDDs, die auf Äquivalenz zu testen sind. Verwendet man dafür den Transformationsalgorithmus 3.4 von Savický und Wegener (1994), wobei man die Größe des kleineren OBDDs als Größenschranke verwendet, ergibt sich für jeden Äquivalenztest (einschließlich der zugehörigen Graphisomorphietests) ein Zeitbedarf von $O(|G|\,|G'|\log|G'|)$ und ein Platzbedarf von $O(|G| + |G'|)$. Also ergibt sich für alle $b \cdot b'$ Äquivalenztests von OBDDs, die für den Äquivalenztest der beiden baumgesteuerten BDDs höchstens nötig sein können, eine Gesamtlaufzeit von $O(b\,|T| + bb'\,|T'| + bb'\,|G|\,|G'|\log|G'|)$ und ein Gesamtplatzbedarf von $O(|T| + |G| + |T'| + |G'|)$. Der Algorithmus ist also langsamer als der allgemeinere Algorithmus von Sieling, dafür aber platzoptimal. Wir halten das Ergebnis in einem Satz fest.

Satz 3.24: Man kann zu einem baumgesteuerten BDD G zu einer Booleschen Funktion f und einer Permutation der Eingangsvariablen π das reduzierte πOBDD für f in polynomieller Zeit platzoptimal berechnen.

□

Wir betrachten noch kurz, was passiert, wenn das baumgesteuerte BDD G als Orakel eine Liste hat, also auch als OBDD interpretiert werden kann. In diesem Fall erhält der angegebene Äquivalenztest für baumgesteuerte BDDs stets BDDs als Eingaben, deren Baumorakel Breite 1 haben. Wenn $b = b' = 1$ gilt, wenn also beide Baumorakel auch Listenorakel sind, sind zwei OBDDs mit verschiedenen Variablenordnungen zu transformieren. Dann ergibt sich als Laufzeit $O(|G|\,|G'|\log|G'|)$ und Platzbedarf $O(|G| + |G'|)$, was mit Algorithmus 3.4 übereinstimmt. Algorithmus 3.22 kann also berechtigt als Verallgemeinerung des πOBDD$\rightarrow$ π'OBDD-Transformationsalgorithmus von Savický und Wegener betrachtet werden.

3.3.4 Transformation eines πOBDD in ein WBDD

Wir wollen hier ein OBDD in ein WBDD transformieren, wobei natürlich der Orakelgraph des Zielmodells vorgegeben ist. Oberflächlich betrachtet erinnert die Aufgabe an die Transformation eines FBDD in ein graphgesteuertes BDD; wir wissen allerdings, daß es nur dann einen deterministischen Polynomialzeitalgorithmus hierfür gibt, wenn es einen deterministischen Polynomialzeitäquivalenztest für FBDDs gibt (vgl. den entsprechenden Abschnitt oben). Der Eindruck der Ähnlichkeit wird subjektiv durch folgende *falsche* Argumentation verstärkt.

Angenommen es gibt einen deterministischen Polynomialzeitalgorithmus für die Transformation eines OBDD in ein graphgesteuertes BDD. Dann transformiert man ein FBDD mit dem Algorithmus von Savický und Wegener in ein OBDD (deterministisch in Polynomialzeit). Anschließend transformiert man das OBDD in ein graphgesteuertes BDD. Anscheinend ergibt sich dann insgesamt ein deterministischer Polynomialzeitalgorithmus.

Tatsächlich ist das nicht so. Es gibt Funktionen, die sowohl eine Darstellung als FBDD als auch eine Darstellung als graphgesteuertes BDD polynomieller Größe haben, die aber nur mit OBDDs exponentieller Größe dargestellt werden können. Ein Beispiel für eine solche Funktion ist die Funktion Hidden Weighted Bit (Bryant (1991)), die wir in Abschnitt 3.4.3 (Definition 3.34) definieren werden. Dann wird durch die erste Transformation ein exponentiell großes Zwischenergebnis erzeugt, obwohl Ein- und Ausgabe nur polynomielle Größe haben. Damit ist der Algorithmus nicht mehr polynomiell.

Wir sind auch tatsächlich in der Lage, ein OBDD mit Hilfe eines deterministischen Polynomialzeitalgorithmus in ein äquivalentes graphgesteuertes BDD zu transformieren. Dabei ist es durchaus von Bedeutung, ob ein WBDD oder ein LBDD erzeugt werden soll. Aus diesem Grund stellen wir hier beide Varianten getrennt vor und beschäftigen uns in diesem Abschnitt mit der Transformation eines πOBDD in ein G^O-gesteuertes WBDD.

Wir benötigen einige technische Begriffe für WBDDs und deren Orakelgraphen. Dazu erinnern wir insbesondere an die Definition eines Schnürknotens $S(v)$ in Definition 2.9 und der zugehörigen Variablenmenge $Var(v)$.

Zu einem reduzierten πOBDD für die Funktion f und einem reduzierten Orakelgraphen G^O soll das G^O-gesteuerte WBDD für f berechnet werden. Der Algorithmus besteht im Kern aus einer rekursiven Funktion `Transform`(G,G^O), die zu einem πOBDD G

ein G^O-gesteuertes WBDD berechnet. Wir bezeichnen zu einem Graphen H den linken Subgraphen von H (der den 0-Nachfolger der Quelle von H als Quelle hat) mit H_0 und den rechten Subgraphen von H (dessen Quelle der 1-Nachfolger der Quelle von H ist) mit H_1. Zu einem Graphen H und einem Knoten v aus H nennen wir $H(v)$ den Subgraphen von H mit Quelle v.

Der Algorithmus verwendet eine Dictionary-Datenstruktur `D`, in der Verweise auf Knoten gespeichert werden. Es wird insgesamt nur ein WBDD W berechnet: in `D` befinden sich Verweise auf Knoten, die als Quellen von Sub-WBDDs fungieren. Als Suchschlüssel wird ein Paar (G', G'^O) aus πOBDD G' und Orakelgraphen G'^O verwendet. Alle im Algorithmus verwendeten OBDDs sind reduziert. Entstehen OBDDs durch Konstantsetzung von Variablen, so ist als Ergebnis stets das reduzierte OBDD zu verwenden.

Algorithmus 3.25 (πOBDD→WBDD):

```
Transform(G,G^O);
/* Eingabe: πOBDD G für f (o.B.d.A. π = id), */
/* und Orakelgraph G^O */
/* Ausgabe: G^O-gesteuertes WBDD für f */
     /* Rekursionsabbruch */
     1.   if G ist Senke, then return G;
     /* Vermeide Erzeugung zu löschender Knoten */
     /* v ist Quelle von G^O */
     2.   if f hängt von keiner Variablen aus Var(v) essentiell ab
          then Ersetze Variable aus Var(v) in G durch Konstante;
               Ersetze G^O durch G^O(S(v));
               Gehe zu 1.;
     /* Vermeide Erzeugung von zu verschmelzenden Knoten */
     3.   if D enthält WBDD W für (G,G^O)
          then return W;
     /* Erzeuge benötigtes G^O-WBDD W */
     4.   Erzeuge Knoten v mit gleicher Markierung x_i wie die
          Quelle von G^O;
     5.   v_0:=Transform(G_|x_i=0,G_0^O);
     6.   v_1:=Transform(G_|x_i=1,G_1^O);
     7.   W:=WBDD mit Quelle v, linken Subgraph v_0, rechtem Subgraph v_1;
     8.   Trage W mit Schlüssel (G,G^O) in D ein;
     9.   return W;
```

Wir formulieren die Eigenschaften des Algorithmus als Ergebnis dieses Abschnitts.

Satz 3.26: Zu einem πOBDD G für eine Boolesche Funktion f und einem Orakelgraphen G^O kann das reduzierte G^O-gesteuerte WBDD W in Zeit $O\left(|W|\left(\left|G^O\right|+|G|\right)\right)$ auf Platz $O\left(|W|\left(\left|G^O\right|+|G|\right)\right)$ berechnet werden.

Beweis: Wir zeigen, daß Algorithmus 3.25 dieses leistet. Wir betrachten zunächst die Rechenzeit. Offenbar wird die Prozedur `Transform` für jede Kante des WBDD W höchstens

einmal aufgerufen, also $O(|W|)$-mal. Der erste Schritt benötigt konstante Zeit. Die Schritte 4, 7 und 9 benötigen konstante Zeit. Die Schritte 5 und 6 benötigen (bis auf die Rekursion) Zeit $O(|G|)$. In den Schritten 3 und 8 ist je eine Dictionary-Operation auszuführen. Es kann höchstens soviele Einträge geben, wie es Knoten in W gibt, also nicht mehr als $O(|W|)$-viele. Jeder Zugriff auf `D` kostet folglich höchstens Zeit $O(\log |W|)$.

Jetzt fehlt nur noch die Abschätzung von Schritt 2. Es ist für eine durch ein OBDD G gegebene Funktion f zu entscheiden, ob f von mindestens einer Variablen aus $Var(v)$ essentiell abhängt. Offenbar kann $Var(v)$ leicht in Zeit $O\left(\left|G^O\right|\right)$ berechnet werden (im Rahmen einer Tiefensuche bis zum Schnürknoten $S(v)$, der selbst ebenfalls leicht in Zeit $O\left(\left|G^O\right|\right)$ berechnet werden kann). Das OBDD G ist reduziert oder kann doch leicht in Linearzeit $O(|G|)$ reduziert werden. Folglich kann in Zeit $O(|G|)$ ein Array über den Variablen berechnet werden, mit dem dann in konstanter Zeit abgelesen werden kann, ob f von der Variablen x_i essentiell abhängt. Nach Berechnung dieses Arrays genügt es, einmal die Menge $Var(v)$ zu durchlaufen und für jede Variable aus dieser Menge in konstanter Zeit zu testen, ob f von dieser Variablen essentiell abhängt. Somit kann insgesamt in Zeit $O\left(\left|G^O\right| + |G|\right)$ der Schritt 2 des Algorithmus durchgeführt werden. Der für das Array zusätzlich benötigte Speicherplatz beträgt nur $O(n)$ und fällt nicht ins Gewicht.

Die Ausführung der Prozedur `Transform` kostet also bis auf die rekursiven Aufrufe höchstens Zeit $O\left(\left|G^O\right| + |G|\right)$. Somit ergibt sich für den Algorithmus insgesamt eine Zeitbeschränkung von $O\left(|W|\left(\left|G^O\right| + |G|\right)\right)$. Zur Abschätzung des Speicherplatzes genügt es zu sehen, daß pro Aufruf von `Transform` höchstens ein OBDD und ein Orakelgraph im Speicher gehalten werden müssen. In der Datenstruktur `D` werden ebenfalls (global für alle Aufrufe) höchstens soviele Zeiger verwaltet, wie es Knoten im WBDD W gibt. Also genügt insgesamt Speicherplatz $O\left(|W|\left(|G| + \left|G^O\right|\right)\right)$. Der Algorithmus ist damit polynomiell in Platz und Zeit.

Kommen wir jetzt zur Korrektheit. Offensichtlich arbeitet der Algorithmus für konstante Funktionen korrekt. Die Verkleinerung der Graphen im zweiten Schritt ändert offenbar die darzustellende Funktion nicht. Der Orakelgraph darf verkleinert werden, da nur nicht benötigte Variablen ausgeschlossen werden. Der dritte Schritt ist zunächst einmal überhaupt möglich, weil reduzierte πOBDDs eine eindeutige Darstellung für Boolesche Funktionen darstellen. Da der Algorithmus stets mit reduzierten πOBDDs arbeitet und die Orakelgraphen ebenfalls minimal gehalten werden, kann es keine zwei verschiedenen Suchschlüssel geben, die auf den gleichen Knoten verweisen. Wird im vierten Schritt ein Knoten erzeugt, so ist dieser notwendig und kann weder mit einem anderen bereits erzeugten Knoten verschmolzen noch gelöscht werden. Der Knoten gehorcht offenbar der durch den Orakelgraphen gegebenen Variablenordnung. Die als Nachfolger berechneten Sub-WBDDs werden rekursiv berechnet und sind somit per Induktionsannahme korrekt. □

Man beobachtet, daß der Algorithmus nicht explizit Bezug nimmt auf die dem OBDD zugrundeliegende Ordnung π; trotzdem kann das πOBDD nicht einfach durch ein FBDD ersetzt werden. Es ist für den Algorithmus (konkret für den Zugriff auf `D`) wesentlich,

daß reduzierte πOBDDs eine eindeutige Darstellung für Boolesche Funktionen sind. Der Algorithmus kann also nicht direkt für FBDDs verwendet werden, was mit den Vorüberlegungen übereinstimmt. Will man FBDDs verwenden, so wäre eine naheliegende Modifikation, zur Verwendung als Suchschlüssel die FBDDs in OBDDs zu transformieren. An dieser Stelle kann sich dann ein exponentieller blow-up ergeben, der dafür sorgt, daß man keinen Polynomialzeitalgorithmus mehr hat. Der beschriebene Algorithmus ist also so nur für OBDDs brauchbar.

3.3.5 Transformation eines πOBDD in ein LBDD

Soll anstelle des WBDD ein äquivalentes LBDD für das OBDD berechnet werden, so kann man natürlich erst Algorithmus 3.25 anwenden und anschließend das Ergebnis mit dem LBDD-Reduktionsalgorithmus in ein LBDD transformieren. Weil ein LBDD aber um den Faktor $\left|G^O\right|$ kleiner sein kann, wird dabei möglicherweise zu viel Platz gebraucht. In praktischen Anwendungen ist beim Umgang mit BDDs häufig Platz eine knappe Ressource, so daß es sich lohnt, über eine platzeffizientere Version des Algorithmus nachzudenken. Tatsächlich ist es ohne größere Umstände möglich, den Speicherplatzbedarf zu senken.

Satz 3.27: Zu einem πOBDD G für eine Boolesche Funktion f und einem Orakelgraph G^O kann das reduzierte G^O-gesteuerte LBDD L in Zeit $O\left(|L|\left|G^O\right|\left(\left|G^O\right|+|G|\right)\right)$ auf Platz $O\left(|L|\left(\left|G^O\right|+|G|\right)\right)$ berechnet werden.

Beweis: Für den Beweis verwenden wir im wesentlichen den Transformationsalgorithmus 3.25 oben; natürlich ist die Bezeichnung „WBDD“ durch „LBDD“ zu ersetzen. *Nach* der vollständigen Erzeugung eines neuen Knotens (also *nachdem* die beiden rekursiven Aufrufe beendet sind) *bevor* der neue Knoten in die Dictionary-Datenstruktur eingetragen wird, überprüfen wir, ob der Knoten gemäß der Verschmelzungsregel mit einem bereits existierenden Knoten verschmolzen werden kann. Dazu modifizieren wir die Dictionary-Datenstruktur so, daß mit einem Tripel (Knotenmarkierung, linker Sohn, rechter Sohn) als Suchschlüssel in derselben Menge von Sub-LBDDs gesucht werden kann. Der Speicherplatzbedarf erhöht sich dabei lediglich um den Platz für die abzulegenden Schlüssel, ist also proportional zur Anzahl der erzeugten Knoten. Wenn ein äquivalenter Knoten gefunden wird, muß der neu erzeugte Knoten wieder entfernt werden. Ergebnis ist dann das gefundene LBDD. Ansonsten wird wie in Algorithmus 3.25 verfahren.

Offenbar wird die gleiche Knotenmenge wie oben betrachtet, der Zeitbedarf sinkt also nicht. Es werden aber höchstens n Knoten mehr im Speicher gehalten, als das LBDD schließlich enthält; pro Ebene gibt es höchstens einen zusätzlichen (nicht bestätigten) Knoten. Damit ergibt sich dann ein Gesamtzeitbedarf von

$$O\left(|L|\left|G^O\right|\left(\left|G^O\right|+|G|\right)\right)$$

und ein Gesamtspeicherplatzbedarf von

$$O\left(|L|\left(\left|G^O\right|+|G|\right)\right).$$

□

3.4 Transformationen mit OFDDs

Nach den bislang fast durchgängig effizienten Transformationsalgorithmen kommen wir nun zu einem Problemfeld, wo wir nicht immer Polynomialzeitalgorithmen präsentieren können. OFDDs sind eine BDD-Variante, die ähnlich wie OBDDs definiert werden kann; wählt man dort die Shannon-Zerlegung als Basis der Funktionsrepräsentation dient bei OFDDs die RME als Schlüssel zur effizienten Darstellung.

Wir haben schon in Abschnitt 2.4 gesehen, daß die Synthese für OFDDs im allgemeinen nicht effizient durchgeführt werden kann. Lediglich für die $\oplus$-Synthese haben wir einen Algorithmus skizziert, der die Synthese in polynomieller Zeit auf polynomiellem Platz durchführt. Bevor wir uns der Aufgabe zuwenden, ein πOBDD in ein äquivalentes πOFDD zu transformieren, wollen wir noch einige strukturelle Besonderheiten von OFDDs betrachten, die das Problem der Transformation so schwierig machen.

Wir haben in Abschnitt 2.4 die Ersetzung von Variablen durch Konstante nicht besprochen; das wollen wir jetzt hier nachholen. Gegeben ist ein πOFDD für die Funktion f, die über den Variablen $x_1, \ldots, x_n$ definiert ist. Wir wollen daraus ein πOFDD für $f_{|x_i=c}$ (mit $c \in \{0,1\}$) berechnen.

Nehmen wir zunächst an, daß $c = 0$ ist, es soll also eine 0-Setzung durchgeführt werden. Dann müssen wir an jedem x_i-Knoten, an dem eine Funktion g berechnet wird, stattdessen die Funktion $g_{|x_i=0}$ darstellen. Das ist leicht möglich, weil der 0-Nachfolger des x_i-Knotens diese Funktion darstellt. Es genügt also — ganz genau wie bei FBDDs und OBDDs —, alle Zeiger auf x_i-Knoten durch Zeiger auf die 0-Nachfolger dieser Knoten zu ersetzen.

Jetzt nehmen wir an, daß $c = 1$ ist, also eine 1-Setzung durchzuführen ist. Dann müssen wir an jedem x_i-Knoten, an dem eine Funktion g berechnet wird, stattdessen die Funktion $g_{|x_i=1}$ darstellen. Das ist allerdings nicht ohne Aufwand trivial realisierbar, weil diese Funktion im allgemeinen an keinem der Knoten des OFDD berechnet wird. Es muß also ein geeignetes πOFDD für die Funktion $g_{|x_i=1}$ berechnet werden. Eine geeignete Lösung ist weder schwer zu erkennen noch aufwendig durchzuführen: der 0-Nachfolger repräsentiert die Funktion $g_{|x_i=0}$, der 1-Nachfolger die Funktion $g_{|x_i=0} \oplus g_{|x_i=1}$. Eine einfache $\oplus$-Synthese der beiden Sub-πOFDDs genügt also, um das gewünschte πOFDD zu erhalten. Dabei kann sich aber die Größe des Ergebnisses ergeben als das Produkt der Größen der beiden Eingabe-OFDDs. Also kann sich die Größe eines OFDD durch die Ersetzung einer Variablen durch 1 quadrieren.

Dieser unangenehme Umstand kann durch fortgesetzte 1-Setzung von Variablen verschlimmert werden. Bollig, Löbbing, Sauerhoff und Wegener (1995) haben gezeigt, daß durch eine logarithmische Anzahl von Ersetzungen von Variablen durch die Konstante 1 die Größe eines OFDD exponentiell zunehmen kann. Weil das Ergebnis für diesen Abschnitt wesentlich ist, werden wir den entsprechenden Satz mit Beweis hier wiedergeben. Vorher müssen wir kurz zwei Funktionen definieren, die für den Beweis von Bedeutung sind. Beide Funktionen sind über $\binom{n}{2}$ Variablen $x_{i,j}$, $1 \leq i < j \leq n$, definiert. Diese x-Variablen beschreiben einen ungerichteten Graphen über n Knoten $1, \ldots, n$. Zwei Knoten i und j sind genau dann mit einer Kante verbunden, wenn $x_{i,j} = 1$ gilt.

Definition 3.28: Die Funktion $1-cl_{n,3}$ nimmt genau dann den Funktionswert 1 an, wenn der durch eine Belegung der x-Variablen beschriebene Graph aus genau einem Dreieck und $n-3$ isolierten Knoten besteht.

Definition 3.29: Die Funktion $\oplus - cl_{n,3}$ nimmt genau dann den Funktionswert 1 an, wenn der durch eine Belegung der x-Variablen beschriebene Graph eine ungerade Anzahl Dreiecke enthält.

Man sieht leicht ein, daß $\oplus - cl_{n,3} = \tau(1 - cl_{n,3})$ gilt; wegen der Eigenschaften von τ gilt somit dann auch $1 - cl_{n,3} = \tau(\oplus - cl_{n,3})$. Ajtai, Babai, Hajnal, Komlós, Pudlák, Rödl, Szemerédi und Turán (1986) haben gezeigt, daß FBDDs für $\oplus - cl_{n,3}$ exponentielle Größe haben; das gilt dann erst recht für OBDDs. Damit ist klar, daß OFDDs für $1-cl_{n,3}$ notwendig exponentielle Größe haben. Andererseits kann die jeweils andere Funktion leicht durch OBDDs/OFDDs polynomieller Größe dargestellt werden. Von dem Umstand, daß die Funktion $1 - cl_{n,3}$ nur OFDD-Repräsentationen exponentieller Größe hat, macht der folgende Satz unmittelbar Gebrauch.

Satz 3.30: Wendet man auf ein πOFDD polynomieller Größe die Operation „Ersetzung von Variablen durch Konstante“ logarithmisch oft an, kann ein πOFDD für eine Funktion entstehen, die nur exponentiell große Darstellungen als OFDD hat.

Beweis: Sei f eine Boolesche Funktion auf $N := \binom{n}{2} + \left\lceil \log \binom{n}{3} \right\rceil$ Variablen. Wir definieren $l := \left\lceil \log \binom{n}{3} \right\rceil$; f ist definiert über den Variablen $y_1, \ldots, y_l, x_{rs}$ für $1 \leq r < s \leq n$. Wir beschreiben jetzt f durch Angabe eines OFDD G, das f berechnet. G beginnt mit einem vollständigen binären Baum mit Tiefe l über den y-Variablen. Dieser Baum hat 2^l Blätter; $\binom{n}{3}$ dieser Blätter führen zu OFDDs für die Funktionen $\Delta_{i,j,k}$ für $1 \leq i < j < k \leq n$, die von den Variablen x_{rs} abhängen. $\Delta_{i,j,k}$ ist genau dann 1, wenn $x_{ij} = x_{jk} = x_{ik} = 1$ gilt und alle anderen x_{rs} mit 0 belegt sind; das kann man so interpretieren, daß $\Delta_{i,j,k}$ genau dann den Funktionswert 1 berechnet, wenn der durch die x-Variablen beschriebene Graph über n Knoten genau das Dreieck i, j, k und $n-3$ isolierte Knoten enthält. Die restlichen Blätter des Baumes über den y-Variablen (falls es welche gibt) zeigen direkt auf die 0-Senke.

Offenbar kann jede Funktion $\Delta_{i,j,k}$ durch ein OFDD der Größe $O(n^2)$ dargestellt werden. Somit hat das OFDD G insgesamt $O(n^5)$ Knoten und damit polynomielle Größe in N.

Die Funktion $f_{|y_1=1,\ldots,y_l=1}$ entsteht aus f durch l-viele Ersetzungen einer Variablen durch die Konstante 1. Wenn man alle Teil-OFDDs, die an den Blättern des vollständigen Baumes über den y-Variablen stehen, mit fortgesetzter $\oplus$-Synthese verbindet, erhält man ein OFDD für $f_{|y_1=1,\ldots,y_l=1}$. Folglich gilt $f_{|y_1=1,\ldots,y_l=1} = \bigoplus_{1 \leq i<j<k \leq n} \Delta_{i,j,k}$. Weil gemäß Definition der Funktionen $\Delta_{i,j,k}$ stets höchstens eine Funktion $\Delta_{i,j,k}$ den Funktionswert 1 annimmt, kann $\oplus$ durch $\vee$ ersetzt werden. Also gilt auch $f_{|y_1=1,\ldots,y_l=1} = \bigvee_{1 \leq i<j<k \leq n} \Delta_{i,j,j}$. Das ist aber gerade die Funktion $1-cl_{n,3}$, die nur OFDD-Repräsentationen exponentieller Größe hat. $\square$

3.4.1 Transformation eines πOBDD in ein πOFDD

Wir wollen πOBDDs in πOFDDs transformieren. Weil OBDDs und OFDDs aufgrund ihrer unterschiedlichen Funktionszerlegungen an den Knoten kaum strukturelle Gemeinsamkeiten aufweisen, werden wir Transformationsalgorithmen auf ziemlich hohem Abstraktionsniveau präsentieren. Wir machen Gebrauch von Operationen auf OBDDs und OFDDs und kümmern uns kaum noch um einzelne Kanten und Knoten, wie wir das vor allem bei der Transformation von OBDDs und ZBDDs in Abschnitt 3.2.1 intensiv getan haben.

Bevor wir konkrete Transformationsalgorithmen vorstellen und diskutieren, wollen wir kurz einige allgemeine Überlegungen darstellen; wir führen das exemplarisch an der Aufgabe der Transformation eines πOBDD in ein πOFDD vor. Für die umgekehrte Transformationsrichtung kann man analoge Aussagen machen.

Naheliegende Transformationsalgorithmen für die Transformation eines πOBDD in ein πOFDD gehen in gewisser Weise rekursiv vor. Es werden rekursiv πOFDDs für geeignete Subfunktionen berechnet, die dann zum gewünschten Gesamt-OFDD zusammengesetzt werden können. Wir betrachten ein πOBDD für eine Boolesche Funktion f, an dessen Quelle ein x_1-Knoten steht. Wir nehmen o. B. d. A. an, daß das OBDD reduziert vorliegt. Offenbar müssen und werden im OBDD dann Knoten bereitstehen, an denen die Funktionen $f_{|x_1=0}$ und $f_{|x_1=1}$ repräsentiert werden. In einem πOFDD für die gleiche Funktion steht ebenfalls ein x_1-Knoten an der Quelle. Dann müssen und werden im OFDD aber Knoten bereitstehen, an denen die Funktionen $f_{|x_1=0}$ und $f_{|x_1=0} \oplus f_{|x_1=1}$ repräsentiert werden. Es werden also im OBDD und im OFDD verschiedene Funktionen repräsentiert. Diese fundamentale Differenz zwischen den beiden Repräsentationsformen muß an irgendeiner Stelle überbrückt werden. Es gibt dafür zwei ganz verschiedene Möglichkeiten.

Man kann einerseits rekursiv OFDDs für $f_{|x_1=0}$ und $f_{|x_1=1}$ berechnen, um dann das OFDD für $f_{|x_1=0} \oplus f_{|x_1=1}$ mit einer $\oplus$-Synthese zu berechnen. Das OFDD für $f_{|x_1=1}$ kann aber quadratisch in der Größe des OFDD für $f_{|x_1=0}$ und f sein, so daß hier (in der Rekursion) ein exponentieller blow-up ermöglicht wird.

Andererseits kann man zunächst ein OBDD für $f_{|x_1=0} \oplus f_{|x_1=1}$ berechnen, um dann rekursiv OFDDs für $f_{|x_1=0}$ und $f_{|x_1=0} \oplus f_{|x_1=1}$ aus den vorliegenden OBDDs zu berechnen. In diesem Fall kann die Größe des OBDD für $f_{|x_1=0} \oplus f_{|x_1=1}$ quadratisch in der Größe des Eingabe-OBDD sein; das Ergebnis-OFDD kann bekanntlich erheblich kleiner sein. Es droht hierbei also (in der Rekursion) OBDD-seitig ein exponentieller blow-up.

Das Problem ist wie gesehen für rekursiv vorgehende Transformationsalgorithmen in der unterschiedlichen Funktionsdarstellung der Modelle begründet, der gestellten Aufgabe also inhärent und somit unvermeidlich. Damit ist natürlich noch nichts gesagt über allgemeine Algorithmen für die Transformation eines πOBDD in ein πOFDD, die eine Transformation auf eine Art und Weise durchführen, die den Rückgriff auf Subfunktionen vermeidet. Daß solche Transformationen im Prinzip denkbar sind, hat exemplarisch der Transformationsalgorithmus 3.15 gezeigt, der nicht rekursiv formuliert ist und keinen Bezug auf Subfunktionen nimmt.

Man sollte nun aber auch nicht umgekehrt annehmen, daß ein bottom-up vorgehender Algorithmus automatisch bessere Ergebnisse bringt. Wenn ein Transformationsalgorithmus bottom-up vorgeht und dabei für jeden OBDD-Knoten ein äquivalentes πOFDD konstruiert, so werden OFDDs für exakt die gleichen Subfunktionen von f berechnet, wie das auch ein rekursiv vorgehender Algorithmus tut, der mittels table-look-up vermeidet, Zwischenergebnisse mehrfach zu berechnen. Die Reihenfolge, in der diese OFDDs berechnet werden, ist abhängig davon, ob man die Berechnung top-down (rekursiv) oder bottom-up vornimmt. Die Anzahl und Größe berechneter Zwischenergebnisse ist in beiden Fällen identisch. Es ist auch in beiden Ansätzen nicht schwierig, nicht mehr benötigte Zwischenergebnisse zu erkennen und so früh wie möglich aus dem Speicher zu entfernen. Die beiden Ansätze unterscheiden sich also bei geeigneter Implementierung weder im Speicherplatz noch in der Laufzeit.

Es ist offensichtlich, daß bei so einem Vorgehen zwei verschiedene Klassen von OFDDs berechnet werden. Zum einen werden OFDDs berechnet, die später als Teile des Gesamt-OFDD wieder auftauchen. Das betrifft etwa das OFDD für $f_{|x_1=0}$ (und in Folge alle ebenfalls ausschließlich durch 0-Setzungen entstehenden Zwischen-OFDDs). Zum zweiten werden OFDDs berechnet, die später im Gesamt-OFDD nicht wieder auftauchen (jedenfalls nicht a priori). Der Transformationsalgorithmus ist genau dann besonders schlecht, wenn das Eingabe-OBDD und das Ergebnis-OFDD polynomielle Größe haben und „auf dem Weg“ ein Zwischen-OFDD berechnet wird, das später nicht mehr im Ergebnis auftaucht und exponentielle Größe hat. Diese Aussage wollen wir konkretisieren.

Satz 3.31: Jeder Algorithmus für die Transformation eines πOBDD in ein πOFDD, der zu jedem Knoten v des πOBDD und der an v repräsentierten Funktion ein äquivalentes πOFDD berechnet, benötigt im worst case exponentielle Laufzeit und exponentiellen Speicherplatz gemessen in der Größe von Ein- *und* Ausgabe.

Vor dem Beweis wollen wir noch einmal die Aussage klarstellen und ihre Konsequenzen explizit formulieren. Die Schwierigkeit bei der πOBDD$\rightarrow$ πOFDD-Transformation besteht darin, daß im OBDD Funktionen als Subfunktionen repräsentiert werden können, die exponentielle Darstellungsgröße als πOFDD haben und im Ergebnis-OFDD gar nicht auftauchen. Alle Transformationsalgorithmen, die sich an der Struktur des πOBDD orientieren und die zwischenzeitlich die gleiche Menge von Booleschen Funktionen intern repräsentieren, müssen exponentiell in Laufzeit und Platz sein.

Beweis: Wir betrachten noch einmal die Funktion f aus Satz 3.30. Wir haben gesehen, daß $f_{|y_1=1,\ldots,y_l=1}$ exponentielle OFDD-Größe hat.

Hier wird jetzt gezeigt, daß f polynomielle OBDD-Größe hat und in einem OBDD für f die Funktion $f_{|y_1=1,\ldots,y_l=1}$ dargestellt wird. Damit stellt f ein worst case Beispiel für die πOBDD$\rightarrow$ πOFDD-Transformation dar, wie sie hier beschrieben wird.

Für einen Vektor $a = (a_1, \ldots, a_l)$ benutzen wir die Schreibweise f_a abkürzend für $f_{|y_1=a_1,\ldots,y_l=a_l}$. Ab hier verzichten wir auf die Betrachtung der Blätter des y-Variablenbaumes im OFDD, die direkt zur 0-Senke führen. Das kann man o. B. d. A. tun, da wir im

folgenden ausschließlich $\oplus$-Summen betrachten, so daß die konstante Nullfunktion keine Rolle spielt.

An den Blättern des y-Variablenbaumes im OFDD werden die Funktionen $\bigoplus_{c_i \leq a_i} f_c$ für alle $a \in \{0,1\}^l$ dargestellt. Wir wollen f in einem OBDD darstellen, das ebenfalls mit einem Baum der Tiefe l über den y-Variablen beginnen soll. An den Blättern dieses Baumes müssen die Funktionen f_a für alle $a \in \{0,1\}^l$ dargestellt werden. Um nachzuweisen, daß f polynomielle OBDD-Größe hat, genügt es zu zeigen, daß alle f_a polynomielle OBDD-Größe haben. Offensichtlich lassen sich alle Funktionen f_a als $\bigoplus$-Summen von $\Delta_{i,j,k}$-Funktionen darstellen: im OFDD werden nämlich gerade $\bigoplus$-Summen von f_a-Funktionen dargestellt, die $\Delta_{i,j,k}$-Funktionen sind gemäß Definition von f.

Alle diese Funktionen f_a sind der Funktion $1-cl_{n,3}$ ähnlich: eine Funktion wird genau dann 1, wenn es genau ein Dreieck auf einer bestimmten Menge von Knoten gibt. Die restlichen Knoten der Knotenmenge müssen alle isolierte Knoten sein. Wegen der Definition der $\Delta_{i,j,k}$ auf verschiedenen Tripeln kann es keinesfalls mehr als eine 1 geben. Alle diese Funktionen f_a können im wesentlichen mit Hilfe eines OBDD repräsentiert werden, wie das bei $1-cl_{n,3}$ geschieht, es gibt nur weniger Knoten, auf denen Dreiecke gebildet werden können. Dadurch wächst das darstellende OBDD aber nicht wesentlich in der Größe.

Also ist f eine Funktion mit polynomieller OBDD-Darstellung, polynomieller OFDD-Darstellung, das OBDD für f enthält einen Knoten für die Funktion $f_{|y_1=1,\ldots,y_l=1}$ und eben diese Funktion hat nur exponentiell große OFDD-Darstellungen. □

Wir stellen hier zwei verschiedene Algorithmen für die Transformation eines πOBDD in ein äquivalentes πOFDD vor. Beide Algorithmen sind auf hohem Abstraktionsniveau und rekursiv formuliert. Die Vorbemerkungen in Abschnitt 3.4 und Satz 3.31 haben verdeutlicht, daß beide Algorithmen im worst case exponentielle Laufzeit und exponentiellen Platzbedarf sowohl in der Länge der Eingabe wie auch in der Länge der Ausgabe haben.

Der erste Algorithmus führt die Transformation so weit wie möglich OFDD-seitig, der zweite möglichst OBDD-seitig aus. Konkret lassen sich beide Algorithmen wie folgt beschreiben. Es soll ein πOBDD für die Funktion f in ein äquivalentes πOFDD transformiert werden. O. B. d. A. ist π die Identität; außerdem hängt f o. B. d. A. von x_1 essentiell ab.

	Algorithmus 1	Algorithmus 2
1. Berechne rekursiv	πOFDD für $f_{\|x_1=0}$;	πOFDD für $f_{\|x_1=0}$;
2. Berechne	rekursiv πOFDD für $f_{\|x_1=1}$;	πOBDD für $f_{\|x_1=0} \oplus f_{\|x_1=1}$;
3. Berechne	πOFDD für $f_{\|x_1=0} \oplus f_{\|x_1=1}$;	rek. πOFDD für $f_{\|x_1=0} \oplus f_{\|x_1=1}$;
4. Setze	Teilergebnisse zusammen;	Teilergebnisse zusammen;

Man beobachtet direkt, daß bei beiden Algorithmen in jedem rekursiven Aufruf die Zahl der Eingangsvariablen um eins abnimmt; damit ist garantiert, daß beide Algorithmen terminieren. Man beobachtet aber ebenfalls, daß beim ersten Algorithmus in jedem rekursiven Aufruf die Größe des Eingabe-OBDD garantiert abnimmt; beim zweiten Algorithmus hingegen kann die Größe des Eingabe-OBDD sogar wachsen.

Es ist selbstverständlich, daß beide Algorithmen nicht naiv rekursiv implementiert werden dürfen. Wie bei allen rekursiv formulierten Algorithmen auf BDD-artigen Datenstrukturen

sind geeignete Datenstrukturen einzusetzen, um die Mehrfachberechnungen von Zwischenergebnissen so weit wie möglich zu reduzieren. Als geeignete Dictionary-Datenstrukturen bieten sich an:

Arrays: Die Zugriffszeit ist garantiert konstant, dafür ist der Platzbedarf sehr groß. Arrays sind darum in der Praxis meist nicht brauchbar.

AVL-Bäume: Die Zugriffszeit ist logarithmisch in der Zahl der abgespeicherten Zwischenergebnisse. Dafür ist der Platzbedarf nur linear.

Hashing: Die Zugriffszeit ist im Durchschnitt praktisch konstant. Im worst case kann die Laufzeit allerdings erheblich größer werden. Hashing hat sich in der Praxis bewährt.

Meiner Meinung nach bestehen wesentliche Unterschiede zwischen den beiden Algorithmen in folgendem: Der erste Algorithmus berechnet OFDDs für Funktionen, die nicht benötigt werden; nicht benötigte OBDDs werden dafür nicht erzeugt. Das bedeutet, daß alle rekursiven Aufrufe in ein und demselben OBDD stattfinden. Damit erhöht sich die Wahrscheinlichkeit, in der Datenstruktur bereits berechnete Zwischenergebnisse wiederzufinden. Außerdem besteht die Hoffnung, daß, wenn die Ergebnisdarstellung im OFDD eine Größe hat, die kleiner als die Größe des Eingangs-OBDD ist, dies auch für viele Zwischenergebnisse gelten kann. Außerdem nimmt in der Rekursion die Größe des zu transformierenden OBDD stets ab, was auf kürzere Rechenzeiten und geringeren Platzbedarf hoffen läßt. Diese Hoffnungen können im worst case selbstverständlich alle unerfüllt bleiben.

3.4.2 Transformation eines πOFDD in ein πOBDD

Wir werden hier in aller Kürze analog zu Abschnitt 3.4.1 zwei Algorithmen für die Transformation eines πOFDD in ein πOBDD angeben. Beide Algorithmen entsprechen ganz den Algorithmen des genannten Abschnitts; wie weit diese Übereinstimmung genau geht, wollen wir uns anschließend kurz überlegen. Wir nehmen wieder o. B. d. A. an, daß π die Identität ist; außerdem hängt f o. B. d. A. von x_1 essentiell ab.

	Algorithmus 1	Algorithmus 2
1. Berechne rekursiv	πOBDD für $f_{\mid x_1=0}$;	πOBDD für $f_{\mid x_1=0}$;
2. Berechne	πOFDD für $f_{\mid x=1=1}$;	rek. πOBDD für $f_{\mid x_1=0} \oplus f_{\mid x_1=1}$;
3. Berechne	rekursiv πOBDD für $f_{\mid x_1=1}$;	πOBDD für $f_{\mid x_1=1}$;
4. Setze	Teilergebnisse zusammen;	Teilergebnisse zusammen;

Die Korrektheit der Algorithmen ist offensichtlich. Über die Laufzeit ist nichts zu sagen, was nicht schon über die Algorithmen für die Gegenrichtung gesagt worden ist. Wir fragen uns jetzt noch, wie eng diese Algorithmen und die zwei Algorithmen aus Abschnitt 3.4.1 zusammenhängen. Dazu betrachten wir die πOBDDs und πOFDDs für andere Funktionen außer der Funktion f, die berechnet werden. Jeder Algorithmus muß πOBDDs für die Funktionen $f_{\mid x_1=0}$ und $f_{\mid x_1=1}$ und πOFDDs für die Funktionen $f_{\mid x_1=0}$ und $f_{\mid x_1=0} \oplus f_{\mid x_1=1}$ berechnen. Die beiden Algorithmen, die wir als Algorithmus 1 bezeichnet haben, berechnen

beide zusätzlich ein πOFDD für $f_{|x_1=1}$. Die beiden Algorithmen, die wir als Algorithmus 2 bezeichnet haben, berechnen *beide* zusätzlich ein πOBDD für $f_{|x_1=0} \oplus f_{|x_1=1}$. Die zwei Algorithmen 1 berechnen ebenso wie die zwei Algorithmen 2 jeweils die gleiche Menge von Zwischenergebnissen. Darum müssen Laufzeit- und Speicherplatzverhalten der beiden Algorithmen auch jeweils gleich sein. Diesen Umstand nutzen wir in Abschnitt 3.4.3 aus: wir werden lediglich die Transformation von πOBDDs in πOFDDs praktisch untersuchen.

3.4.3 Empirische Aussagen über die Transformationen zwischen πOBDDs und πOFDDs

Die in den Abschnitten 3.4.1 und 3.4.2 entworfenen Transformationsalgorithmen zeigen alle im worst case exponentielles Laufzeit- und Platzverhalten — das gilt für die Transformation eines πOBDD in ein äquivalentes πOFDD ebenso wie für die Transformation in umgekehrter Richtung. Das ist auch nicht unbedingt verwunderlich: viele Operationen, die im Kontext von OBDDs effizient realisiert werden können, weisen bei OFDDs einen exponentiellen worst case auf. Es wird aber von guten Praxiseigenschaften von OFDDs berichtet (etwa von Kebschull, Schubert und Rosenstiel (1992), Kebschull und Rosenstiel (1993)). Darum ist es natürlich sinnvoll, die betrachteten Transformationsalgorithmen auch einer empirischen Untersuchung zu unterziehen; dann kann auch möglicherweise abgeschätzt werden, ob trotz des exponentiellen worst case in praktischen Fällen ein Arbeiten mit den Algorithmen möglich ist. Wie wir uns schon in Abschnitt 3.4.2 überlegt haben, genügt es, nur die zwei Transformationsalgorithmen für die Transformation eines πOBDD in ein πOFDD zu untersuchen.

Die Implementierung der Algorithmen fußt auf einem „fertigen“ BDD-Paket, konkret wurde das BDD-Paket von David E. Long verwendet. Für die Verwendung hier waren verschiedene Modifikationen nötig, die in Anhang A ausführlich erläutert werden. Alle Modifikationen wurden so durchgeführt, daß die Effizienz des Long-Pakets nicht beeinträchtigt wurde.

Wir werden im wesentlichen zwei verschiedene Gruppen von Funktionen verwenden, um die praktische Effizienz der Transformationen zu untersuchen. Zum einen werden das einige mehr oder minder „theoretische“ uniforme Funktionen sein, die häufig herangezogen werden, wenn diskutiert werden soll, wie groß die Darstellung typischer Funktionen in bestimmten BDD-Modellen ist. Zum anderen werden wir eine Reihe von Benchmark-Schaltkreisen untersuchen.

Wir definieren zunächst die verwendeten uniformen Funktionen. Die Definitionen der Funktionen hier sind alle Wegener (1994/95) entnommen.

Definition 3.32: Die Funktion *direkte Adressierung* $DA(a_{k-1}, \ldots, a_0, x_0, \ldots, x_{2^k-1})$ ist auf $k + 2^k$ Variablen definiert. Die k Adressvariablen $a_{k-1}, \ldots, a_0$ definieren eine Adresse $a := \sum_{0 \leq i < k} 2^i a_i$. Der Funktionswert ist die durch a bezeichnete Stelle in den x-Variablen x_a.

Definition 3.33: Die Funktion *indirekte Adressierung* $IDA(a_{k-1}, \ldots, a_0, x_0, \ldots, x_{2^k-1})$ ist auf $k+2^k$ Variablen definiert. Die k Adressvariablen $a_{k-1}, \ldots, a_0$ definieren eine Adresse $a := \sum_{0 \leq i < k} 2^i a_i$. Die Variablen x_a, ..., x_{a+k-1} (die Indizes sind $\bmod 2^k$ zu rechnen) bezeichnen ihrerseits eine Adresse b. Der Funktionswert ist die durch b bezeichnete Stelle in den x-Variablen x_b.

Definition 3.34: Die Funktion *Hidden Weighted Bit*[27] $HWB(a_1, \ldots, a_n)$ ist auf n Variablen definiert. Wir bezeichnen die Summe der a_i als $s := \sum_{1 \leq i \leq n} a_i$. Der Funktionswert ist a_s für $s > 0$ und 0 sonst.

Die direkte und die indirekte Adressierung sind Funktionen, die in realen Anwendungen auf natürliche Weise entstehen. Sie sind geeignet, die Eigenschaften von OBDDs zu erhellen: wir bemerken ohne Beweis, daß die direkte Adressierung leicht mit OBDDs polynomieller Größe dargestellt werden kann, die indirekte Adressierung hingegen erfordert exponentiell große OBDDs[28].

Die Funktion Hidden Weighted Bit ist offensichtlich eine etwas künstliche Funktion, die von Bryant (1991) eingeführt wurde. Es handelt sich um eine in gewisser Weise „einfachste“ Funktion, die exponentiell große OBDDs erfordert. Wesentliche Erweiterungen des Modells (Lockerung der Variablenordnung, Aufgabe der read-once Eigenschaft) führen sofort zu polynomiell großen Darstellungen.

Neben den erwähnten uniformen Funktionen haben wir die zwei Transformationsalgorithmen auch bei einigen Benchmark-Schaltkreisen eingesetzt. Wir verwenden dazu Schaltkreise aus der MCNC Benchmark-Suite (Yang (1991)). Konkret verwenden wir zunächst die in Tabelle 2 angegebenen Benchmark-Schaltkreise.

Name	Funktion	Gatter	Eingänge	Ausgänge
C17	Logic	6	5	2
C432	Priority Decoder	160	36	7
alu2	ALU	335	10	6
count	Counter	143	35	16
decod	Decoder	22	5	16
z4ml	2-bit Add	20	7	4

Tabelle 2: Verwendete Benchmark-Schaltkreise

Die Anzahl der Gatter sagt nur sehr begrenzt etwas über die Schwierigkeit des Schaltkreises aus, zumal wir nicht die Umwandlung des Schaltkreises in OBDDs sondern die Transformation der OBDDs in OFDDs betrachten. Wir geben darum in einer zweiten Tabelle (Tabelle 3, S. 80) die Größe aller OBDDs und OFDDs wieder, die sich im Rahmen der Testreihe ergeben. Als Ordnung wählen wir jeweils die Reihenfolge, in der die Inputs in der Schaltkreisbeschreibung auftauchen.

[27] Diese Funktion wurde von Bryant (1991) eingeführt.
[28] Beweise hierzu findet man bei Wegener (1994/95).

Die uniformen Funktionen DA und IDA sind im Test auf 20 Eingabevariablen, HWB auf 16 Eingabevariablen definiert. Es hat sich gezeigt, daß die Verwendung anderer Anzahlen von Eingabevariablen die Ergebnisse nicht wesentlich ändern: die Charakteristika der beiden Algorithmen bleiben im Verhältnis gleich, zumindest solange nicht garbage collections durchgeführt werden.

Name	|OBDDs|	Summe	|OFDDs|	Summe
C17	7/7	14	9/8	17
C432	19/73/73/215/96/177/239	892	28/189/189/401/155/304/388	1665
alu2	38/128/3/3/78/9	259	39/65/3/3/105/7	222
count	9/10/11/12/.../23/24	264	8/11/13/15/.../37/39	383
decod	6/6/.../6	96	6/6/.../6	96
z4ml	27/18/9/4	58	23/13/7/4	47
DA	32	32	97	97
IDA	6884	6884	1884	1884
HWB	743	743	1598	1598

Tabelle 3: Größen der verwendeten Testfunktionen

Alle Testläufe sind auf drei verschiedenen Rechnern durchgeführt worden, zum einen unter Linux 1.1.59 auf einem Pentium 90 PC mit 8 MB internem Speicher und einem Intel 486SX 33 PC mit 4 MB internem Speicher, zum anderen unter SunOS 4.1.4 auf einer Sparc Classic Workstation mit 16 MB internem Speicher. Die Verwendung von drei verschiedenen Rechnern (und zwei verschiedenen Betriebssystemen) soll die Wahrscheinlichkeit reduzieren, daß Hardwarefehler oder fehlerhafte Betriebssysteme die Ergebnisse beeinflussen[29]. Die Größe der Testfunktionen ist so gewählt, daß selbst der kleinste Rechner die Transformationen im internen Speicher durchführen kann. Dadurch sollen die Rechenzeiten auf den drei Architekturen jeweils untereinander vergleichbar bleiben[30]. Ein zweites Ziel ist die Vermeidung einer garbage collection, die eine sehr rechenzeitaufwendige paketinterne Funktion darstellt; kommen garbage collections vor, läßt sich nur noch schwer sagen, zu welchen Teilen die Rechenzeit durch die Transformation an sich bestimmt ist. Die anderen Maße, die wir verwenden, sind aber von durchgeführten garbage collections unabhängig. Wenn man bereit ist, die Rechenzeit nicht mehr als Effizienzmaß zu verwenden und sich auf die Betrachtung des Rechenaufwands der Transformationsalgorithmen allein beschränkt, kann man also auch größere Schaltkreise zum Vergleich heranziehen.

Wir stellen die Ergebnisse der Durchläufe in Tabelle 4 (S. 82) zusammen und machen uns dann an den Versuch einer Interpretation der Ergebnisse. Weil auf allen drei Architekturen (natürlich) die gleichen Ausgaben erzeugt wurden, wird lediglich die Rechenzeit auf den verschiedenen Rechnern gegenübergestellt. Die Rechenzeit wurde ermittelt als

[29]Vor allem der Pentium wurde mißtrauisch untersucht; Fließkommaarithmetik wird allerdings nicht verwendet.

[30]Einzig die Funktion IDA läßt sich nicht im mit 4 MB sehr kleinem Speicher des 486SX transformieren.

Differenz zwischen zwei mit `clock()` ermittelten Werten, die direkt vor Beginn und direkt nach Ende der Transformationen berechnet wurden. Die dabei vom Betriebssystem zur Verfügung gestellten Timer können ungenau sein, es ist nicht wirklich ein Vergleich der Rechenzeiten zwischen den drei Rechnern möglich. Es genügt uns aber auch, die Rechenzeiten auf den verschiedenen Rechnern getrennt zu betrachten und zu ermitteln, ob die Ergebnisse insofern konsistent sind, als sich die ermittelten Werte einigermaßen proportional verhalten. Die Verwendung von Beispielen, die so groß sind, daß die Rechenzeit hinreichend zuverlässig manuell gestoppt werden kann, ist völlig ausgeschlossen: die unterschiedliche Hauptspeichergröße wird bei Funktionen dieser Größe zum einzig ausschlaggebenden Faktor. Weil auf der Workstation die Zeit in Mikrosekunden, auf den Intel PCs in hundertstel Sekunden gemesen wird, werden die auf der Sparc Classic ermittelten Werte mit 10^{-4} multipliziert und dann auf den nächsten ganzen Zahlenwert gerundet. Neben der Rechenzeit betrachten wir zwei weitere Effizienzmaße. Zum einen betrachten wir die Anzahl von Cache-Lese-Zugriffen (Cache look-up); weil von diesen Zugriffen vor allem die fehlgeschlagenen zeitintensive Rechnungen verursachen können, wird die Anzahl der fehlgeschlagenen Lese-Zugriffe (Cache miss) gesondert aufgeführt. Andere Effizienzmaße wie etwa die Anzahl aller erzeugter Knoten ergeben kein anderes Bild. Beide Transformationsalgorithmen haben sich in der Analyse als exponentiell erwiesen, weil viele Knoten temporär erzeugt werden können, die weder Teil der Eingabe noch Teil der Ausgabe sind. Beim ersten Algorithmus sind das die Knoten in der OFDD-Darstellung der Funktion $f_{|x_i=1}$, beim zweiten Algorithmus handelt es sich um die Knoten in der OBDD-Darstellung von $f_{|x_i=0} \oplus f_{|x_i=1}$. Wir stellen die Anzahl tatsächlich erzeugter temporärer Knoten ebenfalls in der Tabelle dar, um zu sehen, wie viele Knoten konkret „nur für Zwischenrechnungen“ erzeugt werden. Wir halten dabei nur die für die Ausgabe erzeugten temporären Knoten fest, zählen also die in rekursiven Aufrufen für temporäre Zwischenergebnisse entstehenden temporären Knoten nicht mit. Diese Zählweise impliziert, daß man bei gleichzeitigem Zählen der Knoten in den OFDD-Darstellungen für $f_{|x_i=0} \oplus f_{|x_i=1}$ als Vergleichswert hier für beide Verfahren den gleichen Wert erhält. Um die Zahl temporärer Knoten besser beurteilen zu können, fügen wir die Zahl von OFDD-Knoten in Darstellungen von $f_{|x_i=0} \oplus f_{|x_i=1}$ in Klammern hinzu. Man mache sich klar, daß alle Maße so gewählt sind, daß kleinere Werte besser als größere Werte sind. Jeder Tabelleneintrag enthält zwei Zeilen: die obere Zeile gibt die Werte des ersten Algorithmus, die untere Zeile die Werte des zweiten Algorithmus wieder.

Bevor wir die beiden verschiedenen Transformationsalgorithmen miteinander vergleichen, wollen wir die ermittelten Werte für sich allein betrachten; dabei wollen wir zum einen die gemessenen Laufzeiten im Vergleich beurteilen, zum anderen interessieren wir uns für die Ergebnisse der Cachezugriffe.

Man kann direkt sehen, daß es keinen wesentlichen Unterschied macht, ob man die Anzahl der lesenden oder die Anzahl der fehlgegangenen Cachezugriffe zählt: vergleicht man die Verhältnisse der Ergebnisse für den ersten und zweiten Transformationsalgorithmus, ergeben sich (außer bei der Funktion count) keine nennenswerten Unterschiede. Vergleicht man auf entsprechende Art die auf den drei Systemen ermittelten Laufzeiten, so zeigt sich, daß zumindest in den Fällen, in denen im Rahmen der Meßgenauigkeit eine nennenswerte

Laufzeit festzustellen ist, die Laufzeitverhältnisse das gleiche Bild widerspiegeln. Es ist also nicht entscheidend, welches Kriterium man für die Messung der Effizienz zugrunde legt, es ergeben sich stets in etwa die gleichen Verhältnisse. Dieser Umstand macht die Interpretation der Zahlenwerte natürlich erheblich leichter.

Name	P 90	486SX 33	Sparc	#look-up	#miss	#tmp. Knoten
C17	0	0	0	28	23	15 (16)
	0	0	0	35	29	16 (16)
C432	7	34	27	7.334	5.210	4.815 (4.751)
	11	36	43	11.387	7.301	3.451 (4.751)
alu2	1	4	5	1.821	1.044	383 (280)
	1	4	5	1.295	837	261 (280)
count	1	4	5	1.408	1.010	808 (368)
	1	4	5	1.430	780	248 (368)
decod	0	0	2	106	65	55 (80)
	0	1	2	117	65	80 (80)
z4ml	0	0	0	225	142	87 (56)
	0	0	2	151	114	59 (56)
DA	0	1	0	146	146	56 (96)
	1	2	2	433	359	46 (96)
IDA[31]	27	89	98	38.565	19.854	982 (1.484)
	260	1388	808	244.963	145.370	15.159 (1.484)
HWB	17	57	63	25.334	12.707	2.510 (2.524)
	18	55	63	21.804	10.970	1.852 (2.524)

Tabelle 4: Maßzahlen der Transformationen

Ein interessanter „Ausreißer“ ist die Funktion HWB: die Anzahlen der Cachezugriffe legen nahe, daß der zweite Algorithmus etwas effizienter ist als der erste (die Unterschiede liegen bei etwa 16%). Die Laufzeiten auf dem 486SX 33 sind auch entsprechend; auf der Sparc brauchen beide Algorithmen gleich lange, auf dem Pentium hingegen sind die Laufzeiten genau umgekehrt. Das muß allerdings gar nicht so irritieren, wie man auf den ersten Blick glauben möchte: das Beispiel ist relativ klein, die Prozessorarchitekturen sind erheblich verschieden, minimale Laufzeitunterschiede wie hier können durchaus auftreten. Dieses Beispiel legt aber doch nahe, die ermittelten Laufzeiten nicht als zentrales Effizienzkriterium zu benutzen, um sich nicht in zu große Abhängigkeit von einem konkreten Rechner zu begeben.

Eine zweite Auffälligkeit beim Vergleich der Anzahl der Cachezugriffe und der Laufzeiten betrifft die Unterschiede zwischen OBDD- und OFDD-Operationen. Scheinbar laufen OBDD-Operationen ein wenig schneller ab als OFDD-Operationen. Das liegt im wesentlichen daran, daß die im Long-Paket durchgeführten Änderungen zum wesentlichen Teil aus

[31] Auf dem 486SX wurde der Swapspeicher benutzt.

dem Hinzufügen von OFDD-Operationen bestehen, die mit etwas mehr Sicherheitsabfragen versehen sind, als die entsprechenden Operationen im OBDD-Teil[32]. Dieses Ungleichgewicht könnte natürlich durch eine geeignete Änderung der Implementation beseitigt werden; weil aber Rechenzeiten hier nicht als Effizienzmaß verwendet werden, ist das unnötig.

Beim Vergleich der Transformationsalgorithmen beschränken wir uns also ausschließlich auf die Zahlen der lesenden und fehlgegangenen Cachezugriffe und lassen die Laufzeiten außen vor. Das macht unsere Betrachtungen wie gesagt von konkret verwendeten Rechnern unabhängig, auch wenn die Vergleiche zwischen den drei verwendeten Computern gezeigt haben, daß sich die Verhältnisse der Laufzeiten meist nicht wesentlich ändern.

Ein Vergleich der beiden Transformationsalgorithmen anhand der empirisch gewonnenen Werte läßt sich nicht leicht durchführen, da es keinen eindeutig besseren Algorithmus gibt. Es gibt vier Beispielfunktionen, bei denen Algorithmus 1 recht eindeutig besser ist als Algorithmus 2 (C17, C432, DA und IDA); weiter gibt es drei Beispielfunktionen, bei denen Algorithmus 2 der etwas bessere Algorithmus ist (alu2, z4ml, HWB). Bei den zwei Funktionen count und decod ist die Situation in etwa bei beiden Algorithmen gleich.

Die Größe der Unterschiede zwischen den beiden Algorithmen fällt bei fast allen Beispielen moderat aus, die Ergebnisse unterscheiden sich höchstens um den Faktor drei; in vielen Fällen ist der Faktor Sogar deutlich kleiner als zwei. Es ist in diesen Fällen nicht wirklich entscheidend, welcher Transformationsalgorithmus verwendet wird. Bei der indirekten Adressierung hingegen beträgt der Quotient zwischen größeren und kleineren Werten schon etwa sieben, die Unterschiede treten also ganz deutlich hervor, wie auch die Laufzeiten zeigen. Diese Ausnahmesituation von IDA rechtfertigt aber sicher nicht die Bevorzugung des ersten Algorithmus; ein einzelnes Beispiel hat dafür gewiß nicht ausreichend Aussagekraft.

Wir wollen versuchen zu erkennen, in welchen Fällen sich welcher Transformationsalgorithmus eher bewährt. Wir wollen diese Aussage empirisch gewinnen und werden darum eine Art induktiv gewonnene Hypothese formulieren, die wir argumentativ zu stützen versuchen, ohne einen formalen Beweis zu führen (oder auch nur führen zu können). Wir ziehen dafür zusätzlich zu den Benchmarkergebnissen aus Tabelle 4 (S. 82) die Größen der OBDDs und OFDDs hinzu, wie sie aus Tabelle 3 (S. 80) hervorgehen. Wir interessieren uns vor allem dafür, welches BDD-Modell die kompaktere Darstellung der betrachteten Funktion bietet. Man erkennt direkt, daß für C17, C432, count, DA und HWB eine Darstellung in OBDDs weniger Knoten erfordert als die äquivalente OFDD-Darstellung. Die Funktion decod hat gleich große OBDD- und OFDD-Darstellungen, bei den anderen Funktionen sind OFDDs kleiner als äquivalente OBDDs. Der Vergleich mit den Ergebnissen aus Tabelle 4 (S. 82) führt uns zu folgender Hypothese.

> Der erste Transformationsalgorithmus ist vorzuziehen, wenn die Darstellung der Funktionen als OBDD kleiner ist als die Darstellung als OFDD.

[32] Diese zusätzlichen Vorsichtsmaßnahmen sollen ausgleichen, daß der OBDD-Teil wesentlich länger und gründlicher verwendet und somit getestet ist als der neu hinzugefügte OFDD-Teil. Der so asymmetrisch entstehende Rechenaufwand wird dabei bewußt in Kauf genommen.

Wir bemerken zunächst, daß die Funktionen HWB und IDA Gegenbeispiele darstellen; die Aussage ist also sicher nicht allgemeingültig. Wir werden trotzdem an dieser Aussage festhalten; HWB und IDA ist gemeinsam, daß sie nur OBDD-Darstellungen exponentieller Größe haben. Von HWB ist außerdem bekannt, daß es auch nur durch exponentiell große OFDDs repräsentiert werden kann[33]. Sie sind insofern extreme und für praktische Anwendungen nicht unbedingt typische Funktionen; nur Funktionen mit polynomieller Darstellungsgröße bleiben bei relevanten Anzahlen von Eingabevariablen praktisch handhabbar.

Wir wollen zunächst versuchen zu verstehen, was die Hypothese nahelegt. Offenbar ist es günstiger, ein OFDD für die Funktion $f_{|x_i=1}$ als ein OBDD für die Funktion $f_{|x_i=0} \oplus f_{|x_i=1}$ zu berechnen, wenn die OBDD-Darstellung insgesamt kleiner ist als die OFDD-Darstellung. Ist die OFDD-Darstellung insgesamt kleiner, scheint die Lage gerade umgekehrt zu sein. Bei der Funktion IDA stimmt das offenbar nicht; das liegt möglicherweise daran, daß die OBDD-Darstellung dort *wesentlich* größer als die OFDD-Darstellung der Funktion ist, so daß es sich lohnt, so wenig OBDDs als irgend möglich zu berechnen; Algorithmus 1 ist offenbar günstiger, weil bei diesem Verfahren die Transformation so weit als möglich OFDD-seitig ausgeführt wird. Für die Funktion HWB könnten analoge Argumente greifen; allerdings sind die Größenunterschiede zwischen OBDD- und OFDD-Darstellung hier nicht so deutlich wie bei der indirekten Adressierung. Die Unterschiede im Laufzeitverhalten zwischen den beiden Transformationsalgorithmen sind aber auch bei weitem kleiner als bei IDA.

Um unsere Hypothese, die wir mehr als Daumenregel denn als Gesetzmäßigkeit verstehen wollen, zu erhärten, werden wir die beiden Algorithmen auf 20 weiteren Schaltkreisen aus der MCNC Benchmark-Suite testen. Wir geben die verwendeten Schaltkreise zusammen mit ihren Größen in Tabelle 5 (S. 85), die Größe der Darstellungen der Funktionen als πOBDD und πOFDD in Tabelle 6 (S. 86) wieder.

Bei der Wiedergabe der Darstellungsgrößen geben wir diesmal nicht mehr die Größen aller zur Ausgabe gehörenden Funktionen separat an; wir beschränken uns sowohl bei OBDD- als auch bei OFDD-Knoten auf die Angabe der Summe der Knoten im jeweiligen Modell. Man kann erkennen, daß die Funktionen cc und pcler8 unsere Hypothese nicht berühren, weil die Darstellung als OBDD und OFDD genau gleich viele Knoten braucht.

In Tabelle 7 (S. 87) findet man die Benchmarkergebnisse für die Funktionen aus Tabelle 6 (S. 86). Von den 18 Funktionen, die bezüglich unserer Hypothese eine Aussage erlauben, sprechen lediglich die Funktionen c8 und unreg gegen unsere Behauptung, die anderen 16 Funktionen hingegen bestätigen unsere Aussage. Man darf also berechtigt sagen, daß wir ein recht gutes heuristisches a-posteriori-Kriterium gefunden haben, das in vielen Fällen zutreffend beschreibt, welcher Transformationsalgorithmus der effizientere ist.

Wir nennen das Kriterium ein a-posteriori-Kriterium, weil seine Anwendung verlangt, daß sowohl die Darstellungsgröße als OBDD als auch die Darstellungsgröße als OFDD bekannt ist; das ist aber tatsächlich erst *nach* Durchführung der Transformation gegeben.

[33]Einen Beweis hierzu findet man zum Beispiel bei Becker, Drechsler und Werchner (1995).

Name	Funktion	Gatter	Eingänge	Ausgänge
apex 6	Logic	452	135	99
apex 7	Logic	167	49	37
b9	Logic	125	41	21
c8	Logic	164	28	18
cc	Logic	47	21	20
cht	Logic	229	47	36
cm151a	Logic	33	12	2
example2	Logic	277	85	66
frg1	Logic	105	28	3
frg2	Logic	1.004	143	139
pcler8	Logic	84	27	17
sct	Logic	91	19	15
term1	Logic	358	34	10
ttt2	Logic	200	24	21
unreg	Logic	97	36	16
vda	Logic	585	17	39
x1	Logic	285	51	35
x2	Logic	42	10	7
x3	Logic	715	135	99
x4	Logic	369	94	71

Tabelle 5: Zusätzliche Benchmark-Schaltkreise

Wir haben also zwar ein einigermaßen zuverlässiges Kriterium für die Auswahl des passenden Algorithmus in der Hand, können dieses Kriterium allerdings in der Praxis nicht anwenden. Dieser unbefriedigende Zustand führt unmittelbar zu einer Idee für eine leichte Verbesserung der Lage. Die Wahl des „richtigen“ Transformationsalgorithmus ist offenbar dann besonders wichtig, wenn die Effizienz der beiden Verfahren sich extrem voneinander unterscheidet. In Fällen, in denen die Algorithmen ähnliches Verhalten zeigen, ist es hingegen unwesentlich, welcher Algorithmus verwendet wird. Man kann als Konsequenz formulieren, daß es hinnehmbar ist, die Transformation etwas weniger effizient durchzuführen, wenn dafür die Gefahr des starken Abweichens vom besseren Laufzeitverhalten gemindert wird. Dieser Idee folgend werden wir „hybride“ Algorithmen konstruieren, die beide Transformationsverfahren in sich vereinen. Ein solcher Hybridalgorithmus beginnt die Transformation wie einer der zwei Transformationsalgorithmen, darf das verwendete Transformationsverfahren aber zwischendurch wechseln. Je nach Startverfahren und Kriterium für einen Wechsel erhalten wir unterschiedliche hybride Algorithmen.

Wir wollen dabei zweierlei erreichen: zum einen wollen wir Algorithmen finden, die robust gegenüber der ursprünglichen Wahl des Transformationsverfahrens sind; unabhängig davon, ob man eingangs Algorithmus 1 oder Algorithmus 2 zur Transformation verwendet, soll am Ende stets ein dicht beieinander liegender Ressourcenbedarf stehen. Wir wollen diese Robustheit aber natürlich nicht dadurch bezahlen, daß wir an Effizienz verlieren: die

Hybridalgorithmen sollen nicht wesentlich langsamer sein als der jeweils bessere „reine“ Transformationsalgorithmus.

Name	Summe |OBDDs|	Summe |OFDDs|
apex 6	3.887	1.885
apex 7	1.906	746
b9	288	383
c8	170	191
cc	140	140
cht	239	267
cm151a	1.022	48
example2	874	805
frg1	206	283
frg2	7.256	4.680
pcler8	191	191
sct	188	199
term1	592	606
ttt2	315	318
unreg	177	190
vda	5.281	1.934
x1	1.663	2.267
x2	90	75
x3	3.887	1.885
x4	1.186	1.310

Tabelle 6: Größe der zusätzlichen Benchmarkfunktionen

Unserer Hypothese folgend, vergleichen wir die Anzahlen von Knoten in der OBDD- und OFDD-Repräsentation. Weil wir *während* der laufenden Transformation das Verfahren wechseln wollen, werten wir die Eingabe- und Ausgabegrößen der rekursiven Transformationsfunktion (für Einzelheiten vgl. Anhang A) aus. Man mache sich direkt hier klar, daß wir damit den durch unsere Hypothese gesteckten Rahmen schon wieder verlassen: über die Größen der Teil-OBDDs und Teil-OFDDs, die im Laufe der Transformation betrachtet werden, haben wir bislang gar keine Aussage gemacht. Wir erweitern unsere Hypothese hier (etwas waghalsig) auf diese Zwischenresultate und hoffen, diese Erweiterung empirisch gerechtfertigt zu sehen.

Name	P 90	486SX 33	Sparc	#look-up	#miss	#tmp. Knoten
apex 6	15	51	58	18.227	11.172	15 (16)
	10	31	42	10.467	7.151	16 (16)
apex 7	6	17	18	7.202	3.999	4.815 (4.751)
	3	10	12	3.779	869	3.451 (4.751)
b9	0	2	2	596	410	314 (407)
	1	3	3	925	611	345 (407)
c8	0	1	2	587	405	372 (172)
	0	1	2	459	267	143 (172)
cc	0	0	0	314	226	160 (124)
	0	1	0	251	194	124 (124)
cht	0	2	3	641	468	230 (254)
	0	3	2	726	496	255 (254)
cm151a	1	5	5	2.189	1.154	119 (40)
	0	2	2	789	562	40 (40)
example2	1	5	7	1.677	1.134	949 (1.696)
	2	5	7	1.543	1.029	1.789 (1.696)
frg1	2	4	5	1.317	959	1.084 (1.099)
	3	11	15	3.153	2.259	587 (1.099)
frg2	42	140	140	44.065	24.517	8.778 (6.072)
	30	86	98	23.645	16.001	7.879 (6.072)
pcler8	0	2	0	484	333	329 (307)
	0	1	2	300	228	307 (307)
sct	0	2	2	585	371	207 (199)
	1	2	3	661	461	241 (199)
term1	3	8	10	2.932	1.891	663 (610)
	3	12	13	3.612	2.508	592 (610)
ttt2	1	2	5	1.095	639	456 (491)
	1	2	5	890	602	443 (491)
unreg	1	2	2	679	465	305 (209)
	1	2	2	457	329	209 (209)
vda	30	93	117	38.902	20.534	8.224 (2.780)
	8	28	38	10.802	7.159	2.701 (2.780)
x1	10	33	40	11.188	6.948	2.927 (3.502)
	15	44	53	13.063	8.965	3.023 (3.502)
x2	0	1	2	283	174	158 (100)
	0	1	0	206	142	100 (100)
x3	15	49	58	18.239	11.170	3.126 (3.312)
	10	32	40	10.413	7.183	3.149 (3.312)
x4	4	13	15	4.515	2.929	3.676 (2.916)
	4	17	20	6.902	3.298	2.946 (2.916)

Tabelle 7: Maßzahlen der zusätzlichen Transformationen

Wir werden drei verschiedene Kriterien variabel halten, so daß wir hier genau acht verschiedene Hybridalgorithmen erhalten, die wir miteinander und mit den „reinen“ Transformationsalgorithmen vergleichen wollen. Zum ersten wird das zu Beginn gewählte Verfahren jeweils für einen Algorithmus festgelegt. Zum zweiten legen wir fest, ob das Transformationsverfahren nur höchstens einmal oder beliebig oft gewechselt werden darf. Drittens werden wir einmal nur fordern, daß Ein- und Ausgabe sich in der Größe überhaupt unterscheiden, um einen Wechsel des Transformationsverfahrens herbeizuführen; andererseits werden wir fordern, daß die Größen der Darstellungen in den beiden Modellen sich wenigstens um einen Faktor 1,5 unterscheiden. Wir führen die so definierten Hybridalgorithmen zur besseren Übersicht in Tabelle 8 auf.

Algorithmus	Startverfahren	Wechsel	Abstandsfaktor
Hyb-1-b-0	Algorithmus 1	beliebig oft	1.0
Hyb-2-b-0	Algorithmus 2	beliebig oft	1.0
Hyb-1-1-0	Algorithmus 1	einmal	1.0
Hyb-2-1-0	Algorithmus 2	einmal	1.0
Hyb-1-b-5	Algorithmus 1	beliebig oft	1.5
Hyb-2-b-5	Algorithmus 2	beliebig oft	1.5
Hyb-1-1-5	Algorithmus 1	einmal	1.5
Hyb-2-1-5	Algorithmus 2	einmal	1.5

Tabelle 8: Hybridalgorithmen

Bevor wir uns empirischen Ergebnissen zuwenden, wollen wir noch kurz zusammenfassen, was wir uns von den Hybridalgorithmen erhoffen. Als erstes wünschen wir für alle Hybridverfahren, daß die Algorithmen, die sich lediglich durch den Startalgorithmus unterscheiden, sehr ähnliche Ergebnisse zeigen. Außerdem sollen alle Hybridalgorithmen im Verhalten nur unwesentlich von dem jeweils besseren „reinen“ Transformationsverfahren abweichen; wir werden diese zwei Verfahren kurz als rein-1 und rein-2 bezeichnen.

Um die Anzahl zu interpretierender Daten überschaubar zu halten, wählen wir aus den 29 benutzten Benchmarkschaltkreisen 17 größere Schaltkreise aus, bei denen wir aufgrund ihrer Größe deutlich erkennbare Differenzen erwarten. Es sei nochmals betont, daß nicht unmittelbar klar ist, daß die Hybridalgorithmen überhaupt eine Verbesserung darstellen. Wir stützen hier wieder alle Aussagen auf konkrete Laufzeitergebnisse, werden also den durch Empirie gesteckten Rahmen in diesem Abschnitt nicht verlassen.

Wir benutzen als einziges Effizienzmaß die Anzahl der look-up Operationen im Cache; diese reduzierte Sicht gibt das Laufzeitverhalten der Algorithmen hinreichend akkurat wieder, wie die oben betrachteten Ergebnisse unmittelbar zeigen. Durch die Beschränkung auf einen einzelnen Wert können die notwendigen Vergleiche der Verfahren deutlich leichter durchgeführt werden. Die konkreten Ergebnisse können am Ende dieses Abschnitts in den Tabellen 11–13 (S. 91–93) nachgeschlagen werden; in diesen Tabellen findet man gleichzeitig die relative Abweichung von der Anzahl der look-ups des besseren „reinen“ Verfahrens.

Wir beobachten als erstes, daß das Testfeld insofern gutmütig genannt werden kann, als 13 der insgesamt 17 Funktionen von allen Verfahren mit ähnlichem Aufwand bearbeitet werden konnten. Nur vier der Schaltkreise (cm151a, frg1, IDA, vda) lassen relative Unterschiede aufkommen, die nicht durch den Faktor 2 begrenzt sind. Unsere erste Zielvorstellung war, daß wir größere Robustheit gegenüber der Wahl des Startverfahrens erreichen. Wir stellen zum Vergleich der Verfahren für alle Algorithmen die minimale, maximale und mittlere relative Abweichung in der Tabelle 9 gegenüber.

Algorithmus	min. rel. Abw.	mittl. rel. Abw.	max. rel. Abw.
rein-1 vs. rein-2	1,02	1,96	6,35
hyb-1-b-0 vs. hyb-2-b-0	1,00	1,01	1,05
hyb-1-1-0 vs. hyb-2-1-0	1,00	1,19	2,77
hyb-1-b-5 vs. hyb-2-b-5	1,00	1,28	3,02
hyb-1-1-5 vs. hyb-2-1-5	1,00	1,25	2,76

Tabelle 9: Vergleich der Robustheit

Man erkennt ohne Schwierigkeiten, daß alle hybriden Verfahren deutlich robuster gegenüber der Wahl des Startverfahrens sind als die reinen Transformationsalgorithmen. Die Verfahren, bei denen die Differenz der Knotenanzahlen nicht erst den Faktor 1,5 überschreiten muß, um einen Wechsel des Transformationsverfahrens herbeizuführen, sind ihrerseits wieder robuster als die Verfahren, die eine größere Differenz erfordern. Das ist nicht verwunderlich: die Wahl des Startverfahrens spielt eine geringere Rolle, wenn es besonders einfach ist, das aktuell benutzte Transformationsverfahren zu wechseln.

Robustheit gegenüber dem Startverfahren ist eine wünschenswerte Eigenschaft, weil dadurch die Gefahr verringert wird, daß man durch eine ungünstige (anfängliche) Wahl des Transformationsverfahrens wesentlich mehr Zeit für die Transformation aufwendet als bei günstigerer Wahl möglich gewesen wäre. Robustheit ist aber nur dann ein sinnvoller Vorzug, wenn dadurch nicht die Effizienz der Transformation insgesamt leidet. Wir betrachten darum jetzt die Effizienz aller Verfahren im Vergleich. Wir geben dabei genau wie in Tabelle 9 die minimale, mittlere und maximale relative Abweichung wieder.

Algorithmus	min. rel. Abw.	mittl. rel. Abw.	max. rel. Abw.
rein-1	1,00	1,50	3,60
rein-2	1,00	1,46	6,35
hyb-1-b-0	0,89	1,42	6,21
hyb-2-b-0	0,89	1,42	6,21
hyb-1-1-0	1,00	1,49	3,24
hyb-2-1-0	0,99	1,40	3,72
hyb-1-b-5	0,91	1,43	5,89
hyb-2-b-5	0,96	1,48	5,89
hyb-1-1-5	1,00	1,26	2,23
hyb-2-1-5	0,96	1,30	2,76

Tabelle 10: Vergleich der relativen Abweichungen

Offenbar sind alle Hybridverfahren im Mittel sogar etwas effizienter als die „reinen“ Verfahren; somit sind die hybriden Algorithmen praktisch immer vorzuziehen. Im Einzelfall kann es erhebliche Effizienzunterschiede geben, wie die maximale relative Abweichung deutlich macht. Hier zeigt sich, daß die Hybridverfahren, die höchstens einmal das Transformationsverfahren wechseln dürfen, etwas besser sind als die anderen Verfahren. Das ist durchaus einzusehen: wenn relativ viele Wechsel der Transformationsalgorithmen durchgeführt werden, sinkt die Wahrscheinlichkeit, gleiche Zwischenergebnisse im Cache wiederzufinden, so daß mehr Teil-OBDDs bis zum Ende durchtransformiert werden müssen.

Auf der gewählten Testmenge ist das Verfahren, das eine größere Differenz der Knotenanzahlen zum Wechsel des Transformationsverfahrens erfordert und höchstens einen Wechsel zuläßt, das Beste (hyb-1-1-5 bzw. hyb-2-1-5). Auch das ist nicht überraschend: es ist wie gesehen günstiger, das Transformationsverfahren höchstens einmal zu wechseln. Dann sollte dieser Wechsel aber doch „wohlüberlegt“, also nicht bei der ersten auftretenden Anzahldifferenz durchgeführt werden.

Wir sehen also insgesamt, daß es erhebliche Möglichkeiten gibt, die Transformationsverfahren durch Kombination und geschickt gewählte Parametrisierung noch etwas besser zu machen. Eine genaue Betrachtung solcher Heuristiken ist hier sicher nur sehr begrenzt sinnvoll und findet deshalb nicht statt. Eine Optimierung der Parametrisierung (etwa das Zulassen von höchstens c Wechseln der Transformationsverfahren für $c \in \mathbb{N}$ oder die Wahl von Differenzfaktoren $d \in \mathbb{R}^+$) erlaubt sicher, das beste Verfahren für die ausgewählte Testmenge zu finden. Damit ist nicht viel darüber ausgesagt, welches Verfahren „überhaupt das Beste“ ist: solche Aussagen sind bei empirischen Untersuchungen von heuristischen Verfahren auch nicht zu erwarten. In der Praxis macht die Erprobung verschiedener Parameter sicherlich Sinn, um für den vorliegenden (im allgemeinen sehr großen) Schaltkreis, ein möglichst gut „passendes“ Transformationsverfahren zu finden.

Schaltkreis	rein-1	rein-2	rel. Abw.
C432	7.334 1,00	11.387 1,55	1,55
alu2	1.821 1,41	1.295 1,00	1,41
apex6	18.227 1,74	10.467 1,00	1,74
apex7	7.202 1,91	3.779 1,00	1,91
cm151a	2.189 2,77	789 1,00	2,77
count	1.408 1,00	1.430 1,02	1,02
example2	1.677 1,09	1.543 1,00	1,09
frg1	1.317 1,00	3.153 2,39	2,39
frg2	44.065 1,86	23.645 1,00	1,86
HWB	25.334 1,16	21.804 1,00	1,16
IDA	38.565 1,00	245.005 6,35	6,35
term1	2.932 1,00	3.612 1,23	1,23
ttt2	1.095 1,23	890 1,00	1,23
vda	38.902 3,60	10.802 1,00	3,60
x1	11.188 1,17	13.063 1,00	1,17
x3	18.239 1,77	10.313 1,00	1,77
x4	4.515 1,00	4.719 1,05	1,05

Tabelle 11: Anzahl der look-ups der „reinen“ Verfahren

Schaltkreis	Hyb-1-b-0	Hyb-2-b-0	rel. Abw.	Hyb-1-1-0	Hyb-2-1-0	rel. Abw.
C432	7.625 1,04	7.625 1,04	1,00	7.334 1,00	7.396 1,01	1,01
alu2	1.489 1,15	1.492 1,15	1,00	1.811 1,40	1.905 1,47	1,05
apex6	9.600 0,92	9.659 0,92	1,01	18.960 1,81	19.206 1,83	1,01
apex7	4.299 1,14	4.280 1,13	1,00	7.202 1,91	7.334 1,94	1,02
cm151a	789 1,00	789 1,00	1,00	2.189 2,77	789 1,00	2,77
count	1.388 0,99	1.388 0,99	1,00	1.408 1,00	1.393 0,99	1,01
example2	1.517 0,98	1.572 1,02	1,04	1.703 1,10	1.522 1,01	1,10
frg1	2.214 1,68	2.214 1,68	1,00	1.317 1,00	2.263 1,72	1,72
frg2	30.046 1,27	30.101 1,27	1,00	44.065 1,86	31.622 1,34	1,39
HWB	26.671 1,22	26.671 1,22	1,00	25.334 1,16	25.335 1,16	1,00
IDA	239.374 6,21	239.374 6,21	1,00	38.565 1,00	38.561 1,00	1,00
term1	2.601 0,89	2.601 0,89	1,00	2.932 1,00	2.999 1,02	1,02
ttt2	1.053 1,18	1.001 1,12	1,05	1.114 1,25	1.009 1,13	1,10
vda	16.202 1,50	16.247 1,50	1,00	35.052 3,24	35.317 3,27	1,01
x1	11.336 1,01	11.338 1,01	1,00	11.188 1,00	11.275 1,01	1,01
x3	9.405 0,91	9.484 0,92	1,01	18.800 1,82	18.980 1,84	1,01
x4	5.031 1,11	5.054 1,12	1,00	4.515 1,00	4.641 1,03	1,03

Tabelle 12: Anzahl der look-ups der Hybridverfahren mit Faktor 1,0

Schaltkreis	Hyb-1-b-5	Hyb-2-b-5	rel. Abw.	Hyb-1-1-5	Hyb-2-1-5	rel. Abw.
C432	7.334 1,00	8.786 1,20	1,20	7.334 1,00	8.786 1,20	1,20
alu2	1.821 1,41	1.295 1,00	1,41	1.821 1,41	1.295 1,00	1,41
apex6	9.876 0,94	10.408 0,99	1,05	18.227 1,74	19.279 1,84	1,06
apex7	4.729 1,25	3.779 1,00	1,25	4.729 1,25	3.779 1,00	1,25
cm151a	807 1,02	789 1,00	1,02	807 1,02	789 1,00	1,02
count	1.408 1,00	1.593 1,13	1,13	1.408 1,00	1.593 1,13	1,13
example2	1.678 1,09	1.543 1,00	1,09	1.678 1,09	1.543 1,00	1,09
frg1	1.317 1,00	3.982 3,02	3,02	1.317 1,00	3.629 2,76	2,76
frg2	35.169 1,49	30.756 1,30	1,14	36.922 1,56	31.496 1,33	1,17
HWB	25.334 1,16	25.337 1,16	1,00	25.334 1,16	25.337 1,16	1,00
IDA	227.024 5,89	227.122 5,89	1,00	38.565 1,00	38.594 1,00	1,00
term1	2.655 0,91	3.720 1,27	1,40	2.963 1,01	3.914 1,33	1,32
ttt2	1.095 1,23	960 1,08	1,14	1.095 1,23	960 1,08	1,14
vda	21.671 2,01	11.936 1,10	1,82	24.094 2,23	15.290 1,42	1,58
x1	11.188 1,00	10.742 0,96	1,04	11.188 1,00	10.742 0,96	1,04
x3	9.687 0,94	10.279 1,00	1,06	17.490 1,70	19.165 1,86	1,10
x4	4.515 1,00	4.792 1,06	1,06	4.515 1,00	4.792 1,06	1,06

Tabelle 13: Anzahl der look-ups der Hybridverfahren mit Faktor 1,5

3.4.4 Weiterführende Gedanken zur Transformation eines πOBDD in ein πOFDD

Wir haben in Abschnitt 3.4.1 zwei verschiedene Algorithmen vorgestellt, die beide geeignet sind, ein πOBDD für eine Funktion f in ein πOFDD für diese Funktion f zu transformieren. Beide Algorithmen können im worst case (gemessen in der Größe von Eingabe *und* Ausgabe) exponentielle Laufzeit und exponentiellen Speicherplatz haben. Zwar haben einige Erläuterungen zu erklären versucht, warum dieses Transformationsproblem schwieriger ist als die anderen betrachteten Transformationsprobleme, einen schlüssigen Beweis für die besondere Schwierigkeit konnten wir aber nicht präsentieren. Auch in diesem Abschnitt werden wir einen solchen Nachweis nicht liefern; allerdings können wir konstruktiv zeigen, daß sich das Problem auch nicht in Zeit und Platz als so schwierig erweist. Konkret wollen wir hier einen Algorithmus erarbeiten, der das Transformationsproblem platzeffizient löst. Der Algorithmus wird allerdings immer exponentielle Laufzeit (in der Zahl der Variablen) haben, sein praktischer Einsatz verbietet sich somit von selbst.

Paradigmatisch ist hier der Transformationsalgorithmus 3.4 für die OBDD-Umordnung, der Platzoptimalität mit hohem (dort aber überschaubarem) Rechenaufwand erkauft. Die OBDD-Umordnung wird ebenenweise top-down durchgeführt: es werden in einer ersten Phase alle möglichen Verschmelzungen in der aktuellen Ebene durchgeführt. In einer zweiten Phase werden dann für alle übriggebliebenen Knoten beide Nachfolger erzeugt, dabei wird darauf geachtet, daß keine eliminierbaren Knoten erzeugt werden.

Um den Algorithmus implementieren zu können, muß zum einen für zwei Knoten des Zielmodells entscheidbar sein, ob sie die gleiche Subfunktion repräsentieren. Zum anderen muß zu einer Funktion der kleinste Index j (bezüglich der Variablenordnung π des Zielmodells) gefunden werden, für den diese Funktion essentiell von x_j abhängt.

Bei der OBDD-Umordnung ist die Eingabe ein π'OBDD, Ausgabe ist ein πOBDD. Zu zwei Knoten des πOBDD kann man durch Folgen von Konstantsetzungen im π'OBDD zwei äquivalente π'OBDDs bestimmen, so daß der Gleichheitstest effizient durchgeführt werden kann. Ganz analog kann auch der minimale Index j im wesentlichen durch eine Folge von Konstantsetzungen im π'OBDD bestimmt werden. Bei der Transformation eines πOBDD in ein πOFDD ist die Lage insofern ähnlich, als auch hier Eingabe ein πOBDD ist. Allerdings wird die Sachlage erheblich verkompliziert, weil im Zielmodell eine andere Funktionszerlegung vorgenommen wird. Die Funktionszerlegung ist allerdings bekannt, so daß prinzipiell klar ist, wie die dargestellten Funktionen berechnet werden können. Es sind jeweils $\oplus$-Summen von durch Konstantsetzung entstehenden πOBDDs zu bilden. Wir wollen platzoptimal vorgehen und werden darum zwar Konstantsetzungen tatsächlich durchführen, auf die $\oplus$-Synthese von OBDDs allerdings verzichten. Stattdessen berechnen wir die $\oplus$-Summen der entsprechenden Funktionswerte. Weil es zu einer Booleschen Funktion auf n Variablen 2^n verschiedene Eingabebelegungen gibt, braucht jeder wesentliche Schritt des Algorithmus zumindest 2^n Schritte. Wir benötigen aber nur wenig zusätzlichen Speicherplatz. Zur Verdeutlichung notieren wir zunächst den Algorithmus in etwas formalerer Weise, wobei wir auf die Wiedergabe aller Details verzichten werden. Wir setzen dabei o. B. d. A. voraus, daß das Eingabe-OBDD keine konstante Funktion repräsentiert.

Algorithmus 3.35 (πOBDD$\rightarrow$ πOFDD):

```
/* Eingabe: πOBDD G für f */
/* Ausgabe: reduziertes πOFDD H für f (o.B.d.A. π = id) */
1.  Reduziere G;
2.  j:=kleinster Index, so daß x_j-Knoten in G vorkommt;
3.  for i:=j to n do L_i:=nil;
4.  Erzeuge zwei verschiedene Senken für H;
5.  Erzeuge neuen x_j-Knoten v als Quelle von H;
6.  Insert(v,L_j);
7.  for i:=j to n do
8.      Führe alle möglichen Verschmelzungen zwischen Knoten
        aus der aktuellen Liste L_i durch;
9.      Erzeuge für alle übriggebliebenen Knoten aus L_i
        beide Nachfolger und füge sie in die entsprechenden
        Listen ein, wenn sie nicht konstante Funktionen repräsentieren;
```

Wir wollen Algorithmus 3.35, der im Grundgerüst mit den Algorithmen 3.4 und 3.19 übereinstimmt, jetzt so verfeinern, daß deutlich wird, wie eine platzeffiziente Durchführung möglich ist.

Als erstes ist festzuhalten, daß wir aus Platzgründen auf ein Wörterbuch D für die Knoten verzichten; wir nehmen dafür quadratische Laufzeiten in Kauf, um im achten Schritt alle möglichen Verschmelzungen finden und durchführen zu können.

Mit jedem Knoten des schrittweise erzeugten OFDDs speichern wir ein Array über den Variablen der Größe n, aus dem wir ablesen können, welche Funktion an diesem Knoten repräsentiert wird. Jedes Arrayfeld kann die Werte 0, 1 oder 2 annehmen. Eine 2 bedeutet, daß die entsprechende Variable noch nicht getestet wurde. Eine 0 bedeutet, daß eine Subfunktion der Ausgangsfunktion f zu bilden ist, bei der die entsprechende Variable mit dem Wert 0 belegt wird. Eine 1 zeigt an, daß zwei Subfunktionen zu bilden sind: bei der einen ist die entsprechende Variable mit 0, bei der anderen diese Variable mit 1 zu belegen. Alle so gebildeten Subfunktionen sind in einer $\oplus$-Summe zu addieren.

Wir initialisieren die Quelle mit einem Array, in dem alle Felder mit 2 belegt sind. Damit weisen wir der Quelle wie gewünscht die Funktion f zu. Nehmen wir jetzt an, daß wir im Algorithmus fortgeschritten sind. Es genügt zu beschreiben, wie für zwei Knoten entschieden werden kann, ob sie zu verschmelzen sind, und wie für einen Knoten seine beiden Nachfolger bestimmt werden können.

Seien u und v zwei Knoten der aktuellen Knotenliste L_i, die mit den Arrays F_u und F_v assoziiert sind. Wir durchlaufen in einer Schleife alle 2^n möglichen Eingaben für die Ausgangsfunktion f. Für jede Belegung der Variablen führen wir folgende Schritte aus: Wir berechnen den an u bzw. v berechneten Funktionswert wie folgt: Für u und v sind jeweils alle durch $\oplus$-Summen zu verknüpfenden Subfunktionen von f implizit durch F_u und F_v gegeben. Man führt nacheinander die nötigen Konstantsetzungen im Eingabe-OBDD durch, berechnet für jedes so erhaltene OBDD den Funktionswert für die aktuelle Variablenbelegung und verknüpft alle so erhaltenen Funktionswerte in einer $\oplus$-Summe.

Auf diese Art müssen niemals mehr als zwei OBDDs im Speicher gehalten werden, wobei keines größer als das Eingabe-OBDD werden kann. Auf diese Art erhält man schließlich zwei Funktionswerte. Sind diese Funktionswerte verschieden, werden unterschiedliche Funktionen repräsentiert, die Knoten u und v können nicht verschmolzen werden. Sind hingegen alle 2^n auf diese Art erhaltenen Funktionswerte gleich, so wird an beiden Knoten die gleiche Funktion repräsentiert, die Knoten können also verschmolzen werden. Der Test benötigt folglich für zwei Knoten jeweils Zeit $O(2^{2n}\,|G|)$ und Platz $O(|G|)$.

Sei nun v ein mit x_j markierter Knoten, für den die beiden Nachfolger zu bestimmen sind. Im mit v assoziierten Array F_v ist die j-te Stelle bisher mit 2 belegt; die read-once Eigenschaft erzwingt, daß bislang kein Test von x_j vorgekommen sein kann. Für den 0-Nachfolger wird diese Stelle in F_v nun mit 0, für den 1-Nachfolger mit 1 belegt. Für die so erzeugten neuen Arrays werden wie beim Äquivalenztest zweier Knoten in einer Schleife sämtliche Funktionswerte berechnet. Dabei wird bestimmt, ob die repräsentierte Funktion konstant ist; wenn das der Fall ist, wird die entsprechende Senke der Nachfolger. Sonst wird der kleinste Index $k > j$ bestimmt, für den die Funktion essentiell von x_k abhängt. Dieser Schritt kann insgesamt in Zeit $O(2^{2n}\,|G|)$ und Platz $O(|G|)$ durchgeführt werden.

Damit braucht Algorithmus 3.35 insgesamt Zeit $O(2^{2n}\,|G|\,|F|^2)$. Wir brauchen in keinem Schritt mehr Platz als $O(|G|)$, zusätzlich wird selbstverständlich Platz für die Knoten des OFDD H benötigt. Weil wir mit jedem Knoten ein Array der Größe n abspeichern, müssen wir insgesamt Platz $O(|G| + n\,|H|)$ belegen. Damit ist der Algorithmus nicht platzoptimal — er ist allerdings auch nicht sehr weit von Platzoptimalität entfernt. Wir müssen lediglich den Speicherplatz für die mit den Knoten assoziierten Arrays einsparen. Das ist auch gar nicht weiter schwierig: für jeden Knoten u hängt die Belegung des Arrays nur vom bei seiner Erzeugung gewählten Weg von der Quelle zu ihm ab. Welche Felder des Arrays nicht mit 2 belegt sind, hängt davon ab, welche Variablen bis zu diesem Knoten getestet wurden. Die Belegung dieser Felder hängt davon ab, über welche Kante der Weg führt. Wenn mit jedem Knoten ein Hinweis auf den Knoten abgespeichert wird, von dem aus der Knoten erzeugt wurde, kann das benötigte Array leicht durch Zurücklaufen bis zur Quelle erzeugt werden. Mit dieser Modifikation ändert sich die Laufzeit des beschriebenen Transformationsalgorithmus nicht, der Platzbedarf beträgt dann $O(|G| + |H|)$; der Algorithmus ist also platzoptimal.

Man kann mit etwas Mühe die Laufzeit noch leicht verbessern. Es ist durchaus möglich, ähnlich wie bei der OBDD-Umordnung ein Wörterbuch über den Knoten des OFDD zu bilden. Dabei muß allerdings ein veränderter Suchschlüssel benutzt werden, da ein äquivalentes OBDD ja aus Platzgründen nicht berechnet werden darf. Man kann aber als Suchschlüssel die Folge von OBDDs benutzen, die zur Bestimmung der Funktionswerte herangezogen wird. Die Schlüssel werden komponentenweise sortiert angeordnet; dann können die Verschmelzungen ein wenig effizienter durchgeführt werden. Als neue Laufzeit ergibt sich dann Zeit $O(2^{2n}\,|G|\,|F|\log|F|)$.

Algorithmus 3.35 unterscheidet sich grundlegend von allen bis jetzt betrachteten Transformationsalgorithmen: die Struktur der Eingabe wird vollkommen ignoriert; konkret wird die Eingabe lediglich für die effiziente Berechnung der Funktionswerte für konkrete Va-

riablenbelegungen benutzt. Der Algorithmus orientiert sich ausschließlich an der Struktur der Ausgabe; es wird konkret schrittweise berechnet, welche Funktionen repräsentiert werden. Damit ist klar, daß man mit einem geeignet modifizierten Ansatz noch andere Transformationen platzoptimal durchführen kann.

Satz 3.36: Man kann Algorithmus 3.35 so implementieren, daß ausgehend von einer Booleschen Funktion $f : \{0,1\}^n \to \{0,1\}$, die in einem Format gegeben ist, das die Berechnung des Funktionswertes $f(a)$ zu einem beliebigen Eingabevektor $a \in \{0,1\}^n$ auf konstantem Platz erlaubt, platzoptimal das minimale πOBDD, πZBDD oder πOFDD zu einer beliebigen Variablenordnung π berechnet wird.

Beweis: Man beobachtet direkt, daß auf die Konstantsetzungen im OBDD, die oben beschrieben wurden, verzichtet werden kann. Für Konstantsetzungen genügt es offenbar, nur die Funktionswerte für die geeigneten Variablenbelegungen zu berechnen. Dadurch steigt der Platzbedarf sicher nicht. Folglich ist die einzige Anforderung an die Eingabedatenstruktur, daß die Auswertung auf Platz $O(1)$ durchgeführt werden kann. Für die Konstruktion der Ausgabe ist entscheidend, daß zu jedem Knoten im aktuell berechneten Teilergebnis auf Platz $O(1)$ berechnet werden kann, welche Funktion an diesem Knoten dargestellt werden muß, wenn der Weg von der Quelle zu diesem Knoten bekannt ist. Außerdem müssen alle Funktionswerte für diese Funktion auf konstantem Platz berechenbar sein. Für OFDDs haben wir das oben ausführlich gezeigt. Für OBDDs und ZBDDs ist das ebenfalls gegeben: die Funktionszerlegung an den Knoten ist bekannt, die Berechnung der darzustellenden Funktionen und deren Funktionswerte ist sowohl bei OBDDs als auch bei ZBDDs offensichtlich nicht schwieriger als bei OFDDs. Damit ist klar, daß Algorithmus 3.35 geeignet so modifiziert und implementiert werden kann, daß er insgesamt nur Platz $O(|G| + |H|)$ braucht, wenn $|G|$ die Größe der Eingabe und $|H|$ die Größe der Ausgabe bezeichnet. □

3.4.5 Transformation eines π'OFDD in ein πOFDD

Nach den nicht sehr befriedigenden Ergebnissen aus den beiden vorangegangenen Abschnitten 3.4.1 und 3.4.2 kann hier für die Transformation eines π'OFDD für die Funktion f in ein πOFDD für die Funktion f ein effizienter Algorithmus angegeben werden, der sogar platzoptimal ist.

Satz 3.37: Die Transformation eines π'OFDD G in das reduzierte πOFDD H kann in Zeit $O(|G|\,|H| \log |H|)$ auf Platz $O(|G| + |H|)$ berechnet werden, wenn G und H beide mindestens cn Knoten für eine Konstante $c > 0$ enthalten.

Sowohl Laufzeit als auch Platzbedarf stimmen überein mit den π'OBDD$\to$ πOBDD- und π'ZBDD$\to$ πZBDD-Transformationsalgorithmen. Das ist (natürlich) kein Zufall. Bevor wir Satz 3.37 beweisen, wollen wir eine Hilfsaussage machen, die den wesentlichen Teil des Beweises enthält und eine möglicherweise überraschende Tatsache zum Gegenstand hat.

Lemma 3.38: Der Algorithmus für die Transformation eines π'ZBDD in ein πZBDD aus Abschnitt 3.2.3 kann unverändert verwendet werden, um ein π'OFDD in ein πOFDD zu transformieren.

Beweis: Wir werden beim Beweis zwei Transformationsalgorithmen präsentieren, die beide eine OFDD-Umordnung realisieren. Beide Verfahren sind im Laufzeit- und Platzverhalten merklich schlechter als die in Satz 3.37 angekündigte Transformation; wir benötigen die Verfahren aber hier im Beweisgang.

Grundlage des Beweises ist die strukturelle Verwandtschaft zwischen quasi-reduzierten OBDDs und OFDDs, wie wir sie in Satz 2.22 herausgestellt haben[34]. Das quasi-reduzierte πOFDD für eine Funktion f ist identisch mit dem quasi-reduzierten πOBDD für die Funktion $\tau(f)$. Damit kann man einen Transformationsalgorithmus für den Wechsel der Variablenordnung bei OFDDs konstruieren, der wie folgt aussieht: Eingabe ist das reduzierte π'OFDD G für f, Ausgabe ist das reduzierte πOFDD H für f.

1. Berechne zu G das quasi-reduzierte π'OFDD G_1 für f.
2. Berechne zu G_1 das reduzierte π'OBDD G_2 für $\tau(f)$.
3. Berechne zu G_2 das reduzierte πOBDD G_3 für $\tau(f)$ mit Algorithmus 3.4.
4. Berechne zu G_3 das quasi-reduzierte πOBDD G_4 für $\tau(f)$.
5. Berechne zu G_4 das reduzierte πOFDD H für f.

Wir überlegen uns zunächst, daß die Korrektheit der Umordnung gewährleistet ist und alle Schritte leicht durchgeführt werden können. Für die Konstruktion des quasi-reduzierten π'OFDD G_1 für f genügt analog zur Konstruktion des quasi-reduzierten ZBDD aus dem reduzierten ZBDD in Abschnitt 3.2.2 die geeignete Anwendung einer inversen Eliminationsregel für OFDDs. Wir haben schon gesehen, daß G_1 gleichzeitig als π'OBDD für $\tau(f)$ interpretiert werden kann. Damit genügt für den zweiten Schritt des Algorithmus eine OBDD-Reduktion. Im dritten Schritt wird der OBDD-Umordnungsalgorithmus 3.4 angewendet. Das so erhaltene reduzierte πOBDD G_3 für $\tau(f)$ kann mit Algorithmus 3.13 zum quasi-reduzierten πOBDD G_4 für $\tau(f)$ transformiert werden. Genau wie beim Übergang vom ersten zum zweiten Schritt kann hier, beim Übergang vom vierten zum fünften Schritt, G_4 auch als πOFDD für $\tau(\tau(f))$ interpretiert werden. Gemäß unserer Überlegungen aus Abschnitt 2.4 ist $\tau(\tau(f)) = f$, also liegt mit G_4 ein πOFDD für f vor, das im fünften Schritt nur noch reduziert werden muß.

Wir betrachten den Algorithmus noch etwas genauer und stellen fest, daß eine Verbesserung möglich ist. Die ersten zwei Schritte des Algorithmus erfordern eine Anwendung von Reduktionsregeln: zunächst ist eine inverse Eliminationsregel anzuwenden, danach ist eine Reduktion durchzuführen. Genau in dieser Reihenfolge gingen wir bei der Transformation

[34] Ausgangspunkt des Verfahrens ist ein polynomieller Äquivalenztest für OFDDs zu verschiedenen Ordnungen, der von Beate Bollig entwickelt wurde.

eines πOBDD in ein πZBDD vor; wir haben dort gesehen, daß eine Integration der beiden Schritte eine Verbesserung bedeutet. Für die konkrete Situation hier haben wir sogar schon einen geeigneten Algorithmus entworfen, es handelt sich um den Algorithmus für die Transformation eines πZBDD in ein πOBDD: Der erste Schritt erfordert die Anwendung der inversen Eliminationsregel für OFDDs, die mit der inversen Eliminationsregel für ZBDDs identisch ist. Der zweite Schritt erfordert eine OBDD-Reduktion. Ganz analog kann man die Schritte 4 und 5 integrieren und in einem Schritt durch Algorithmus 3.15 durchführen lassen. Weil wir schon im Zusammenhang mit ZBDDs gesehen haben, daß dieses Vorgehen eine Verbesserung darstellt, machen wir uns hier gar keine Gedanken um Laufzeit und Platz; stattdessen wenden wir uns gleich dem so modifizierten Algorithmus zu, den wir zunächst einmal aufschreiben.

1. Wende auf G den πZBDD$\rightarrow$ πOBDD-Transformationsalgorithmus an. Das Ergebnis ist G'.

2. Wende auf G' den π'OBDD$\rightarrow$ πOBDD-Transformationsalgorithmus von Savický und Wegener an. Das Ergebnis ist G''.

3. Wende auf G'' den πOBDD$\rightarrow$ πZBDD-Transformationsalgorithmus an. Das Ergebnis ist H.

Dieser Algorithmus bewerkstelligt eine OFDD-Umordnung, ist allerdings nicht platzoptimal. Die Verwendung der Transformationen zwischen ZBDDs und OBDDs stellt immer noch eine Art Umweg dar; die Graphen G' bzw. G'' können jeweils um einen Faktor n größer sein als die Graphen G und H. Wir betrachten den darum nicht befriedigenden Algorithmus jetzt aber aus einer etwas anderen Perspektive. Man sieht leicht ein, daß dieser Algorithmus, den wir für die Transformation des reduzierten π'OFDD in das reduzierte πOFDD für eine Funktion f konstruiert haben, ohne Änderung geeignet ist, das reduzierte π'ZBDD für g in das reduzierte πZBDD für g zu transformieren. Weil reduzierte OFDDs und ZBDDs eine eindeutige Darstellung für Boolesche Funktionen sind, ist ganz offensichtlich, daß die Eigenschaft, Transformationsalgorithmus für beide Modelle zu sein, *jedem* Transformationsalgorithmus zukommt, der zu einem π'OFDD ein äquivalentes πOFDD (oder zu einem π'ZBDD ein äquivalentes πZBDD) berechnet. Das gilt also insbesondere auch für Algorithmus 3.19 aus Abschnitt 3.2.3. Der dort entwickelte Algorithmus kann folglich unverändert für die OFDD-Umordnung benutzt werden. □

Lemma 3.38 kann als ein Spezialfall einer allgemeinen Regel betrachtet werden, deren Gültigkeit offensichtlich ist. Alle Verfahren, die auf ZBDDs operieren und nur Bezug nehmen auf die ZBDD-Reduktionsregeln, können unverändert auf OFDDs übertragen werden. Das gilt, weil die ZBDD- und OFDD-Reduktionsregeln identisch sind.

Beweis von Satz 3.37: Nach Lemma 3.38 ist Algorithmus 3.19 ein geeigneter Algorithmus. Die Aussagen über die Laufzeit und den Speicherplatz ergeben sich dann als direkte Folgerung aus Satz 3.18. □

4 Abschließende Bemerkungen

Wir wollen in diesem Abschnitt noch einmal kurz zusammenfassen, welche Transformationsprobleme wir konkret betrachtet haben. Wir haben OBDDs als zentrales BDD-Modell benutzt; beinahe alle hier vorgestellten Algorithmen gehen von OBDDs aus oder enden bei OBDDs. Damit bleiben in dem Gebiet, das die eingeführten BDD-Modelle — OBDDs, ZBDDs, OFDDs und graphgesteuerte BDDs — abgesteckt haben, große Lücken. Wir wollen versuchen, diese Lücken zu schließen, ohne eigens neue Algorithmen konstruieren zu müssen. Wir wollen die fehlenden Transformationsalgorithmen durch geeignete Kombination uns bekannter Algorithmen erhalten. Das dient nicht nur der Vollständigkeit; zusätzlich hilft es zu sehen, wie weit die vorgestellten Ideen und die darauf basierenden Algorithmen tragen.

4.1 Transformationen mit OFDDs

Wir beginnen, indem wir die Grenzen dieses Ansatzes aufzeigen. Wir haben keinen Algorithmus angegeben, der einen Wechsel von oder zu OFDDs ermöglicht und der polynomielle Laufzeit im worst case garantiert. Der einzige Algorithmus, der polynomiellen Platzbedarf garantieren kann (Algorithmus 3.35) erkauft die Platzeffizienz mit unzumutbarer Laufzeit. Damit ist klar, daß sich hier auch durch Kombination nur exponentielle Algorithmen ergeben können.

4.2 Transformationen mit ZBDDs

Besser sieht die Situation bei ZBDDs aus. Wir können zeit- und platzoptimal ein πOBDD in ein πZBDD transformieren (und umgekehrt), so daß wir alle für OBDDs konstruierten Algorithmen verwenden können. Da das reduzierte πOBDD für f sich von dem reduzierten πZBDD für f in der Größe höchstens um einen Faktor n unterscheidet, werden die Algorithmen in Zeit- und Platzverhalten durch den „Umweg“ über OBDDs nicht unzumutbar schlechter.

Satz 4.1: Man kann zu einem πOBDD G für f ein π'ZBDD H für f berechnen in Zeit $O(n\,|G|\,|H|\log|H|)$ auf Platz $O(n\,|G|+|H|)$.

Beweis: In einem ersten Schritt transformieren wir das πOBDD G in das reduzierte πZBDD G' mit Algorithmus 3.15. Weil $|G'| = O(n\,|G|)$ gilt, kommen wir mit Platz $O(n\,|G|)$ und Zeit $O(n\,|G|)$ aus. Dann wechseln wir die Variablenordnung mit Algorithmus 3.19. Die Eingabe für den Algorithmus ist G' mit Größe $O(n\,|G|)$, Ausgabe ist H. Damit ergeben sich die behaupteten Zeit- und Platzschranken. □

Satz 4.2: Man kann zu einem πZBDD G für f ein π'OBDD H für f berechnen in Zeit $O(n\,|G|\,|H|\log|H|)$ auf Platz $O(n\,|G|+|H|)$.

Beweis: Wir gehen analog zum Beweis von Satz 4.1 vor. Weil für die Gegenrichtung Algorithmen mit gleichem Zeit- und Platzverhalten vorliegen, erhalten wir auch in der Kombination gleiches Zeit- und Platzverhalten. □

Um aus einem ZBDD ein graphgesteuertes BDD zu machen, verwenden wir die entsprechenden Transformationen, die wir für OBDDs erarbeitet haben.

Satz 4.3: Man kann zu einem πZBDD G für f ein G^O-gesteuertes WBDD W für f berechnen in Zeit $O\left(|W|\left(\left|G^O\right| + n\,|G|\right)\right)$ auf Platz $O\left(|W|\left(\left|G^O\right| + n\,|G|\right)\right)$.

Beweis: Wir gewinnen mit dem Algorithmus aus dem Beweis zu Satz 3.16 ein πOBDD für f, das Größe $O(n\,|G|)$ hat. Die Anwendung von Algorithmus 3.25 liefert dann ein G^O-gesteuertes WBDD W für f. Weil die Eingabe Größe $O(n\,|G|)$ hat, kommen wir wie behauptet mit Zeit und Platz $O(|W|\,(\left|G^O\right| + n\,|G|))$ aus. □

Satz 4.4: Man kann zu einem πZBDD G für f ein G^O-gesteuertes LBDD L für f berechnen in Zeit $O\left(|L|\left|G^O\right|\left(\left|G^O\right| + n\,|G|\right)\right)$ auf Platz $O\left(|L|\left(\left|G^O\right| + n\,|G|\right)\right)$.

Beweis: Genau wie beim Beweis von Satz 4.3 führen wir nach der Transformation in ein πOBDD mit dem Algorithmus aus Abschnitt 3.2.2 den entsprechenden OBDD-Transformationsalgorithmus aus dem Beweis von Satz 3.27 aus. Damit erhalten wir insgesamt die behaupteten Zeit- und Platzschranken. □

Satz 4.5: Man kann zu einem graphgesteuerten BDD G für f ein ZBDD H für f berechnen in Zeit $O(n\,|G|\,|H|\log n\,|H|)$ auf Platz $O(n\,|G|\,|H|)$.

Beweis: Man berechnet mit dem Transformationsalgorithmus von Savický und Wegener (1994) (Abschnitt 3.1.1) aus dem graphgesteuerten BDD ein äquivalentes OBDD G'. G' hat dabei Größe $O(n\,|H|)$. Bei der anschließenden Transformation in ein äquivalentes ZBDD mit Algorithmus 3.15 kann die Größe nicht weiter zunehmen. Wir erhalten also die behaupteten Zeit- und Platzschranken. □

Weil wir für die Transformation eines baumgesteuerten BDD in ein OBDD einen besseren Algorithmus finden konnten, als die allgemeine Transformation eines FBDD in ein OBDD, können wir hier auch für ZBDDs ein etwas besseres Ergebnis präsentieren.

Satz 4.6: Man kann zu einem baumgesteuerten BDD G mit Baumorakel T für f ein ZBDD G für f in polynomieller Zeit auf Platz $O(|T| + |G| + n\,|H|)$ berechnen.

Beweis: Wir führen die Transformation eines baumgesteuerten BDD in ein πOBDD (Algorithmus 3.22) und die Transformation eines πOBDD in ein πZBDD (Algorithmus 3.15) hintereinander aus. Damit ergeben sich die behaupteten Eigenschaften. □

4.3 Transformationen zwischen graphgesteuerten BDDs

Die einzige Frage, die wir in Zusammenhang mit Transformationen zwischen graphgesteuerten BDDs noch nicht in Betracht gezogen haben, ist, ob wir ein baumgesteuertes BDD effizienter in ein LBDD oder WBDD transformieren können, als das bei graphgesteuerten BDDs zu beliebigen Orakelgraphen der Fall ist. Vor allem die platzoptimale Transformation eines baumgesteuerten BDD in ein OBDD könnte eine solche Vermutung nahe legen; wir können hier aber durch Kombination von Algorithmen nichts gewinnen. Weil OBDDs und graphgesteuerte BDDs sich exponentiell in der Größe unterscheiden können, ist mit einem Umweg über OBDDs sicher nichts gewonnen. Wir können also nur die allgemeinen Algorithmen für die Aufgabe benutzen, die selber nicht effizient waren.

4.4 Tabellarische Gesamtübersicht

Wir wollen in einer Tabelle noch einmal übersichtlich zusammenfassen, welche Ergebnisse wir insgesamt präsentieren konnten. Wir beschränken uns dabei auf die Betrachtung der Laufzeit; platzeffiziente Algorithmen lassen sich leicht konstruieren, wenn man exponentielle Laufzeit in Kauf zu nehmen bereit ist, wie wir in Satz 3.36 gezeigt haben. Wir vermerken in einem Tabellenfeld jeweils die Stelle, wo das entsprechende Ergebnis notiert ist. Für Transformationsprobleme, die wir nicht betrachtet haben, verwenden wir die Kürzel „expo“ und „RP“ als Tabelleneintrag. Der Eintrag „expo.“ soll anzeigen, daß mit den hier diskutierten Transformationen nur exponentielle Algorithmen für dieses Problem aufgezeigt werden könnten. Das betrifft ausschließlich Transformationen, an denen OFDDs beteiligt sind. Der Eintrag „RP“ soll anzeigen, daß probabilistische Polynomialzeitalgorithmen konstruiert werden können, die als Subroutine den FBDD-Äquivalenztest von Blum, Chandra und Wegmann (1980) benutzen. Es ist anzumerken, daß wir für Transformationen mit baumgesteuerten BDDs oft keine eigenen Transformationsalgorithmen konstruiert haben. Wir verwenden dann die entsprechenden Algorithmen und Ergebnisse für allgemeine graphgesteuerte BDDs. Die Transformation eines LBDD in ein OBDD haben wir ebenso wie die Transformation eines WBDD in ein OBDD auf die Transformation eines FBDD in ein OBDD zurückgeführt. Die so erhaltenen Ergebnisse schlagen sich ähnlich wie bei baumgesteuerten BDDs in der Übersicht nieder.

	πOBDD	πZBDD	πOFDD	G^O-LBDD	G^O-WBDD	T-WBDD
πOBDD	—	3.15	3.4.1	3.27	3.25	3.25
π'OBDD	3.4	4.1	expo.	3.27	3.25	3.25
πZBDD	3.16	—	expo.	4.4	4.3	4.3
π'ZBDD	4.2	3.19	expo.	4.4	4.3	4.3
πOFDD	3.4.2	expo.	—	expo.	expo.	expo.
π'OFDD	expo.	expo.	3.19	expo.	expo.	expo.
G^O-LBDD	3.2	4.5	expo.	—	3.3.2	RP
G'^O-LBDD	3.2	4.5	expo.	RP	RP	RP
G^O-WBDD	3.2	4.5	expo.	3.3.2	—	RP
G'^O-WBDD	3.2	4.5	expo.	RP	RP	RP
T-WBDD	3.22	4.6	expo.	RP	RP	—

Tabelle 14: Übersicht aller Transformationen

A Integration von OFDDs in das BDD-Paket von Long

Grundlage der Implementierung der beiden verschiedenen Algorithmen für die Transformation eines πOBDD in ein πOFDD aus Abschnitt 3.4.1 ist eine Bibliothek zum Umgang mit OBDDs, die von David E. Long stammt[35] und vollständig in der Programmiersprache C implementiert ist. Diese Bibliothek, die wir im folgenden kurz als BDD-Paket bezeichnen werden, stellt alle wichtigen Funktionen zum Umgang mit OBDDs und MTBDDs (multiterminal binary decision diagrams; es handelt sich hierbei um eine OBDD-Variante, bei der an den Senken Zahlen notiert werden dürfen) in überaus effizienter Implementierung zur Verfügung. Die Funktionalität des BDD-Pakets umfaßt Standardfunktionen wie Synthese, Ersetzung von Variablen durch Konstante, Erfüllbarkeitstest, Ausgabe von OBDDs und Statistiken, Speichern und Laden von OBDDs. Zentrale Datenstruktur ist ein sogenannter `bdd_manager`; OBDDs sind immer einem `bdd_manager` zugeordnet, dabei können prinzipiell beliebig viele OBDDs, die *einer* Variablenordnung unterliegen, in einem `bdd_manager` untergebracht werden. Der Manager beinhaltet alle Datenstrukturen, die für alle OBDDs relevant sind, etwa die Variablenordnung und die zugrundeliegenden Variablen, außerdem verwaltet er benutzte Hashtabellen. Zum einen werden alle OBDD-Knoten in einer Hashtabelle gespeichert, die verschiedenen OBDDs haben dabei gemeinsame Knoten: im Paket werden negierte Kanten zugelassen; eine negierte Kante, die zu einem Knoten führen, der eine Funktion f repräsentiert entspricht semantisch einer Kante zu einem Knoten, der die Funktion $\overline{f}$ repräsentiert. So genügt es stets, nur eine Senke darzustellen. Zum anderen werden alle Operationen mit Unterstützung von Hashtabellen ausgeführt, so daß Mehrfachauswertungen bei gleichen Aufrufparametern vermieden werden.

Zentral für die Effizienz ist eine eigene Speicherverwaltung, die Long in einer eigenen Speicherbibliothek untergebracht hat. Der Benutzer der BDD-Bibliothek wird, auch bei

[35] Die Beschreibung der OBDD-Bibliothek von Long stützt sich auf die on-line verfügbaren Manualseiten und den Quelltext der Bibliotheksroutinen.

Hinzufügen eigener Funktionen, so weit wie möglich vom Umgang mit der Speicherverwaltung abgeschirmt. Zentral ist eine Prozedur zur Erzeugung eines neuen OBDD-Knotens, die zum einen sicherstellt, daß stets nur reduzierte OBDDs im Speicher gehalten werden, zum anderen alle erforderlichen Schritte (Speicheranforderung, Überprüfung der Einhaltung eines möglicherweise vorhandenen Limits für die Anzahl von Knoten, Manipulation der Hashtabelle) durchführt, um Inkonsistenzen im `bdd_manager` zu vermeiden.

Das beschriebene BDD-Paket ist in zweierlei Hinsicht modifiziert worden, bevor es hier für den Test der Algorithmen für die Transformation von πOBDDs in πOFDDs eingesetzt wurde. Die erste Modifikation fand am Lehrstuhl 2 des Fachbereichs Informatik der Universität Dortmund statt[36], dabei wurden zum einen verschiedene neue Funktionen implementiert, zum anderen wurde eine neue Datenstruktur eingeführt.

Das BDD-Paket von Long unterstützt wie oben erläutert zunächst nur πOBDDs; in praktischen Anwendungen ist es aber offensichtlich sinnvoll, OBDDs für verschiedene Funktionen nicht disjunkt darzustellen, wenn gleiche Subfunktionen darzustellen sind. Man läßt darum mehrere Quellen zu (für jede darzustellende Funktion eine Quelle); die so entstehende Variante nennt man SBDDs (shared binary decision diagrams), sie wurde von Minato, Ishiura und Yajima (1990) eingeführt. Tatsächlich werden im Original-Long-Paket schon SBDDs verwendet; die Knoten aller πOBDDs (die in einem BDD-Manager zusammengefaßt sind), werden in *einer* Hashtabelle gehalten, dabei kann ein Knoten durchaus zu verschiedenen OBDDs gehören. Die Einführung von SBDDs erfordert also keinerlei Modifikationen des Long-Pakets, man kann quasi als neue Oberfläche SBDDs als neue Datenstruktur dem Benutzer zur Verfügung stellen. Verwendet werden Arrays variabler Länge von OBDDs.

Im Gegensatz zur eher kosmetischen Erweiterung um SBDDs stellen die funktionalen Erweiterungen eine echte Bereicherung des Long-Pakets dar. Zum einen wurde der OBDD-Umordnungsalgorithmus 3.4 implementiert, der allerdings im Rahmen dieser Arbeit nicht praktisch eingesetzt wird. Aus Sicht dieser Diplomarbeit ist als wesentlich die Implementierung einer Funktion zu nennen, die das Einlesen von Schaltkreisen im Structure Logic Interchange Format (SLIF) ermöglicht; ein eingelesener Schaltkreis wird dabei in ein äquivalentes SBDD umgewandelt. Dabei kann entweder die durch die Reihenfolge der Eingabevariablen in der SLIF-Datei gegebene Ordnung oder eine explizit angegebene Ordnung für das SBDD verwendet werden. Die zwei hier vorgestellten Algorithmen setzen auf den so generierten OBDDs auf, um sie in äquivalente OFDDs zu transformieren.

Die Erweiterungen des BDD-Pakets von Long, die für die Implementierung der zwei Transformationsalgorithmen durchgeführt werden müssen und für die Ermittlung der empirischen Ergebnisse aus Abschnitt 3.4.3 im Rahmen dieser Diplomarbeit implementiert wurden, sind allerdings deutlich anderer Natur. OFDDs unterscheiden sich strukturell von OBDDs, so sind beispielsweise die Eliminationsregeln für beide Modelle verschieden (vgl. Abschnitt 2.4). Es muß also an verschiedenen Stellen unterschieden werden, ob OBDD- oder OFDD-Knoten verwaltet werden. Um die Struktur nicht unnötig zu verkomplizieren, habe ich entschieden, nur Knoten eines Typs in einem `bdd_manager` zuzulassen; es werden

[36] Die Modifikation wurde von Oliver Giel und Carsten Endrikat durchgeführt.

also stets entweder nur OBDD- oder nur OFDD-Knoten verwaltet. Funktionen, die eine Transformation durchführen sollen, müssen dann mit zwei verschiedenen Managern arbeiten. Das ist insofern etwas problematisch, als daß nicht ganz klar ist, welcher Hashtabelle ein Aufruf einer Transformationsfunktion zuzuordnen ist; wir ordnen jeden Aufruf der Hashtabelle des Managers zu, der das Ergebnis enthält oder enthalten wird.

Im wesentlichen genügt es, die Funktion zu modifizieren, die für die Erzeugung neuer Knoten zuständig ist. Bei OFDDs wurden keine negierten Kanten (außer Kanten zur Senke) zugelassen, Negationen von OFDDs werden durch $\oplus$-Synthese mit der konstanten Einsfunktion durchgeführt. Die $\oplus$-Synthese mußte für OFDDs neu implementiert werden, da sich eine OFDD-Synthese nicht ohne Schwierigkeiten mit den OBDD-Synthesealgorithmen durchführen läßt. Wir geben hier zunächst die Implementierung der $\oplus$-Synthese wieder, dabei beschränken wir uns auf die Wiedergabe der Kernfunktion `bdd_ofddxor_step`. Diese Funktion ist in der Implementierung in der für den Benutzer sichtbaren Funktionen `bdd_ofddxor` gekapselt; dadurch lassen sich Fehler, die durch Speichermangel oder ähnliche Ausnahmesituationen entstehen, elegant abfangen.

Zum Verständnis der Quelltexte sind folgende Bemerkungen zur Notation wahrscheinlich hilfreich. Die beiden Senken sind stets mittels `BDD_ZERO` und `BDD_ONE` zu erreichen. Dabei ist notwendig der verwendete `bdd_manager` mitanzugeben, da jeder Manager eigene Senken verwaltet. Zu einem OBDD `f` kann man mit `BDD_ELSE(f)` den 0-Nachfolger und mit `BDD_THEN(f)` den 1-Nachfolger ermitteln. Die Bezeichnung deutet die enge Verwandtschaft zu Branchingprogrammen an. Bei OFDDs sind diese Bezeichnungen offensichtlich irreführend; sie werden aber unverändert weiter benutzt, damit möglichst viele Funktionen ohne Quelltextänderungen OBDDs *und* OFDDs behandeln können. Die Markierung eines Knotens `f` kann man mit `BDD_INDEXINDEX(f)` ermitteln. Alle anderen verwendeten Funktionen und Makros verraten allein durch ihren Namen genug über ihre Aufgabe.

```
bdd bdd_fddxor_step(bdd_manager fddm,bdd f,bdd g)
{
        bdd_indexindex_type     top_indexindex;
        bdd                     f1,f2,g1,g2,temp1,temp2,result,tmpres;

        BDD_SETUP(f);
        BDD_SETUP(g);
        /* Zuerst werden einfache Sonderfaelle behandelt. */
        if (f == BDD_ZERO(fddm)) { BDD_TEMP_INCREFS(g); return g; }
        if (g == BDD_ZERO(fddm)) { BDD_TEMP_INCREFS(f); return f; }
        if (BDD_SAME_OR_NEGATIONS(f,g))
                if (f == g)
                        return BDD_ZERO(fddm);
                else
                        return BDD_ONE(fddm);
        if (BDD_OUT_OF_ORDER(f,g)) /* Stelle eindeutige Reihenfolge */
                BDD_SWAP(f,g);     /* fuer Hashoperationen her. */
```

```
        if (bdd_lookup_in_cache2(fddm,OP_FDDXOR,(INT_PTR)f,(INT_PTR)g,
                        (INT_PTR *)&result))
                return result;
        /* Das Ergebnis muss tatsaechlich berechnet werden. */
        /* top_indexindex wird der minimale Variablenindex der Quellen */
        /* von f und g zugeordnet. */
        BDD_TOP_VAR2(top_indexindex,fddm,f,g);
        /* Dem Index folgend werden die OFDDs fuer die Rekursion */
        /* ausgewaehlt. */
        if (top_indexindex==BDD_INDEXINDEX(f))
        { f1=BDD_THEN(f); f2=BDD_ELSE(f); }
        else
        { f2=f; f1=BDD_ZERO(fddm); }
        if (top_indexindex==BDD_INDEXINDEX(g))
        { g1=BDD_THEN(g); g2=BDD_ELSE(g); }
        else
        { g2=g; g1=BDD_ZERO(fddm); }
        temp1=bdd_fddxor_step(fddm,f1,g1); /* rekursiv 1-Nachfolger */
        temp2=bdd_fddxor_step(fddm,f2,g2); /* rekursiv 0-Nachfolger */
        /* bdd_find erzeugt noetigenfalls einen neuen Knoten. */
        result=bdd_find(fddm,top_indexindex,temp1,temp2);
        bdd_insert_in_cache2(fddm,OP_FDDXOR,(INT_PTR)f,(INT_PTR)g,
                (INT_PTR)result);
        return result;
}
```

Ganz analog zur Kapselung der OFDD-Synthese ist die OFDD-Transformation gekapselt, auch hier genügt die Darstellung der zentralen Funktion `bdd_bdd2fdd_step`, die beide Transformationsalgorithmen realisiert. Welche Transformationsmethode konkret zu benutzen ist, wird durch einen Parameter festgelegt.

```
bdd bdd_bdd2fdd_step(bdd_manager bddm,bdd f,bdd_manager fddm,int method)
{
        bdd                     bdd0,bdd1,fdd0,fdd01,result;
        bdd_indexindex_type     top_indexindex;
        int                     negated=0;

        BDD_SETUP(f);
        /* Triviale Faelle brechen die Rekursion ab. */
        if (BDD_IS_CONST(f))
        {
                if (f == BDD_ONE(bddm))
                        return BDD_ONE(fddm);
                else
```

```
                        return BDD_ZERO(fddm);
        }
        if (!BDD_IS_OUTPOS(f))/*keine negierte Kante zur Quelle erlauben*/
        { negated=1; f=BDD_NOT(f); }
        if (bdd_lookup_in_cache1(fddm,OP_BDDFDD,(INT_PTR)f,
            (INT_PTR *)&result))
        {
                if (negated)
                        result=bdd_fddxor_step(fddm,result,BDD_ONE(fddm));
                return result;
        }
        /* Ergebnis muss berechnet werden. */
        bdd0=BDD_ELSE(f);
        bdd1=BDD_THEN(f);
        /* OFDD fuer f0 rekursiv. */
        fdd0=bdd_bdd2fdd_step(bddm,bdd0,fddm,method);
        if (method==2)
        {/*2.Algorithmus: OBDD f0 xor f1, OFDD fuer f0 xor f1 rekursiv*/
                bdd     bdd01;
                bdd01=bdd_ite_step(bddm,bdd0,BDD_NOT(bdd1),bdd1);
                fdd01=bdd_bdd2fdd_step(bddm,bdd01,fddm,method);
        }
        else
        {/*1.Algorithmus: OFDD fuer f1 rekursiv, OFDD fur f0 xor f1*/
                bdd     fdd1;
                fdd1=bdd_bdd2fdd_step(bddm,bdd1,fddm,method);
                fdd01=bdd_fddxor_step(fddm,fdd0,fdd1);
        }
        top_indexindex=BDD_INDEXINDEX(f);
        /* bdd_find erzeugt noetigenfalls neuen OFDD-Knoten. */
        result=bdd_find(fddm,top_indexindex,fdd01,fdd0);
        bdd_insert_in_cache1(fddm,OP_BDDFDD,(INT_PTR)f,(INT_PTR)result);
        if (negated)
                result=bdd_fddxor_step(fddm,result,BDD_ONE(fddm));
        return result;
}
```

Beide Implementierungen folgen „dem Geist“ des Long-Pakets, sind im wesentlichen im gleichen Stil wie alle Funktionen des Pakets implementiert. Die Vermeidung von Kanteninvertern bei OFDDs liegt nicht ganz auf der Linie des Long-Pakets; weil aber beide Algorithmen auf OFDDs operieren müssen, spielt das beim praktischen Vergleich der beiden verschiedenen Transformationsalgorithmen keine Rolle.

Es ist ohne Aufwand einzusehen, daß beide Transformationsalgorithmen insofern ähnlich implementiert sind, als daß Implementationsdetails die Unterschiede zwischen den

Algorithmen nicht verfälschen. Unterschiede im Laufzeitverhalten sind also als algorithmenbedingt und nicht als implementationsbedingt anzusehen. Die Implementierung darf als geeignet für praktische Untersuchungen betrachtet werden.

B Bemerkungen zur Implementierung von Transformationsalgorithmen

Der Schwerpunkt dieser Diplomarbeit liegt im Entwurf und der Analyse verschiedener Transformationsalgorithmen. Transformationsalgorithmen lassen sich in der Praxis sinnvoll als Tools in verschiedenen Situationen einsetzen; das gilt sowohl für Umordnungsalgorithmen als auch für allgemeine Transformationen. Wir wollen zwei Beispiele zur Verdeutlichung anführen.

Nehmen wir zum einen an, daß für zwei verschiedene Schaltkreise entschieden werden soll, ob sie die gleichen Funktionen repräsentieren. Einer der beiden Schaltkreise gilt etwa als fehlerfrei, der zweite soll eine (möglicherweise kosteneffizientere) Realisierung darstellen. Zur Verifikation genügt es zu überprüfen, ob die berechneten Funktionen gleich sind. Viele Heuristiken zur Bestimmung guter Variablenordnungen lassen sich von der Struktur des Schaltkreises leiten oder nutzen zumindest eine so gefundene Ordnung als Ausgangspunkt für Verbesserungen (Panda und Somenzi (1995)). So entstehen auf natürliche Weise beispielsweise OBDDs zu verschiedenen Ordnungen. Zur Verifikation kann eine OBDD-Darstellung mit Hilfe eines Umordnungsalgorithmus transformiert werden, um den Gleichheitstest einfach zu machen.

Ein zweites Beispiel betrifft etwa die Verifikation eines Schaltkreises, der eine gegebene Spezifikation erfüllen soll. Möglicherweise ist der Entwurf des Schaltkreises stark an der Shannon-Zerlegung orientiert, so daß die Erstellung eines OBDD zum Schaltkreis nahe liegt und vermutlich günstig ist. Wenn die Spezifikation stärker algebraisch orientiert ist, liegt eine Verwendung von OFDDs nahe, um die in der Spezifikation beschriebenen Funktionen zu repräsentieren. Dann ist eine Transformation der OBDDs in OFDDs (oder umgekehrt) sinnvoll, um wiederum den Gleichheitstest effizient durchführen zu können.

Obwohl Transformationsalgorithmen wie gesehen von erheblicher *praktischer* Bedeutung sein können, haben wir uns hier fast ausschließlich auf ihre theoretische Untersuchung beschränkt. Vor der Verwendung der Algorithmen ist eine geeignete Implementierung durchzuführen; wir wollen hier in aller Kürze einige mehr oder minder allgemeine Überlegungen zur Implementierung von Transformationsalgorithmen zusammenfassen.

Transformationsalgorithmen werden sicher niemals um ihrer selbst willen implementiert, normalerweise werden sie als Tools oder Subroutinen größerer Programme eingesetzt. Damit ist offensichtlich, daß die Transformationsalgorithmen in aller Regel in bestehende BDD-Pakete eingepaßt werden müssen; die Erfahrungen, die hier konkret mit dem Long-Paket gesammelt wurden, dürfen vermutlich als durchaus typisch gelten.

Bei der Implementierung von Transformationsalgorithmen sind — wie bei vielen konkreten Implementierungen — verschiedene, widerstreitende Ziele miteinander in Einklang

zu bringen. Ein zentraler Konflikt besteht im Problem der Trennung der veschiedenen BDD-Modelle: zum einen soll eine möglichst saubere Trennung zwischen den verschiedenen Modellen erreicht werden, um die Algorithmen auf den einzelnen Modellen ohne viele Fallunterscheidungen möglichst effizient implementieren zu können. Andererseits ist es für Transformationen möglicherweise wünschenswert, Knoten nur durch Neuinterpretation zu „transformieren"; bei der Transformation zwischen πOBDDs und πZBDDs (Algorithmus 3.15) etwa werden viele Knoten des OBDD unverändert ins ZBDD übernommen. Das ist in einer Implementierung nur möglich, wenn die Trennung zwischen verschiedenen Modellen nicht zu strikt ist. Sonst müssen beizubehaltende Knoten kopiert werden; das schmälert ohne Zweifel die Effizienz.

Das Kopieren von BDDs und Teil-BDDs ist für einige Algorithmen durchaus von Bedeutung. Die Funktionalität von BDD-Paketen sollte um entsprechende effiziente Funktionen bereichert werden, zumindest im Rahmen solcher Kopierfunktionen sollten die Grenzen zwischen verschiedenen BDD-Modellen ohne weiteren Effizienzverlust überwunden werden können.

Ein zweites zentrales Problem sind benötigte Zusatzinformationen, die mit den Knoten assoziiert werden müssen. Wir haben schon oft erwähnt, daß beim Umgang mit BDDs gerade Platz meist eine kritische Ressource ist. Darum ist es natürlich ein wichtiges Ziel, für jeden BDD-Knoten nur möglichst wenig Speicherplatz zu benutzen; es soll also nur wenig oder kein Speicherplatz ungenutzt bleiben. Andererseits müssen für Transformationsalgorithmen häufig zusätzliche Informationen bereitgestellt werden: so kann zum Beispiel für jeden Knoten eine Liste seiner Vorgänger benötigt werden; möglicherweise muß auch nur ein bestimmter Vorgänger in konstanter Zeit zugreifbar sein. Algorithmus 3.15 wiederum erfordert, daß mit den beiden ausgehenden Kanten jedes Knotens zusätzliche Informationen verbunden werden. Es wird also deutlich, daß Transformationsalgorithmen zum Teil zusätzlichen Speicherplatz in erheblichem Umfang erfordern, auch wenn dieser erforderliche Speicherplatz in der Analyse nicht auftaucht. Multiplikative Erhöhung des Speicherplatzbedarfs ist beim Umgang mit BDDs ein ernstzunehmender Umstand, auch wenn die Faktoren relativ kleine Konstante sind (und im Rahmen der O-Notation keine Rolle spielen). Weil dieser nur im Rahmen der Transformation erforderliche Speicherplatz nicht ständig (ungenutzt) zur Verfügung stehen sollte, sind Wege zu finden, wie mit den Knoten dynamisch zusätzlicher Speicherplatz verbunden werden kann. Dabei muß der Zugriff vom Knoten auf den mit ihm assoziierten Speicherplatz (und auch die umgekehrte Zugriffsrichtung, von den Zusatzinformationen zum Knoten) effizient in konstanter Zeit möglich sein. Die Anforderung zusätzlichen Speicherplatzes sollte dabei genau wie die abschließende Wiederfreigabe nicht viel Zeit in Anspruch nehmen.

Als letztes nicht von der Hand zu weisendes Problem ist die Portabilität zu nennen. Selbstverständlich ist es bei der Implementierung von Programmen, die BDD-Strukturen verwenden, wie bei beinahe allen Programmen wünschenswert, die Programme ohne großen Aufwand auf andere Hardware- und Softwareplattformen portieren zu können. Im Idealfall sollte eine einfache Neucompilierung des Programms ausreichen. Diese Zielsetzung verlangt offensichtlich, bei der Implementierung keinen Gebrauch von spezifischen Eigenschaften einer bestimmten Rechnerumgebung zu machen. Andererseits läßt sich im Rah-

men einer auf Effizienz bedachten Implementierung gerade die Ausnutzung von Hardwareeigenschaften praktisch nicht vermeiden. So ist es sicher sinnvoll, die Größe von primitiven Dateneinheiten der gegebenen Prozessorarchitektur anzupassen und ihr entsprechend 16, 32 oder auch 64 Bit-Zahlen zur Grundlage der BDD-Strukturen zu machen. Die Verwendung von Kanteninvertern ist häufig sinnvoll, um Speicherplatz zu sparen. Aus Effizienzgründen kann es sinnvoll sein, das benötigte Bit direkt in einer unbenutzten Stelle der verwendeten Zeiger unterzubringen. Daß ein solches Verfahren in direktem Widerspruch zum Ziel der Portabilität steht, liegt auf der Hand. Ein letzter in diesem Rahmen zu nennender Punkt ist die Speicherverwaltung. Werden BDDs verwendet, müssen häufig neue Knoten angefordert und andere Knoten gelöscht werden; dabei handelt es sich jeweils nur um kleine Speichereinheiten. Bei vielen Betriebssystemen ist die Anforderung von Speicher eine zeitaufwendige Operation, das häufige Anfordern und Wiederfreigeben sehr kleiner Speichereinheiten fördert außerdem das Fragmentieren des Speichers, so daß hier zusätzliche Probleme zu erwarten sind. Ein üblicher Ausweg besteht hier in der Implementierung einer eigenen Speicherverwaltung, deren Ziel das Ersetzen der Standardspeicherverwaltung durch effizientere Routinen ist; hier liegt eine weitere Quelle erheblicher Schwierigkeiten bei der Portierung auf andere Betriebssysteme oder Hardwareplattformen.

Es gibt also eine ganze Reihe von Schwierigkeiten, die bei der Implementierung von BDD-Funktionalität im allgemeinen und bei der Implementierung von Transformationsalgorithmen besonders auftreten. Das Einfügen von Transformationsalgorithmen in bestehende BDD-Pakete ist eine im hohen Maße nicht-triviale Aufgabe; vor allem tiefgreifende Modifikationen der BDD-Pakete selbst sind erforderlich, um eine reibungslose Zusammenarbeit zwischen BDD-Routinen verschiedener BDD-Modelle einerseits und Transformationsalgorithmen andererseits zu ermöglichen. Es gibt in praktischer Hinsicht also noch viel Platz für Verbesserungen.

Literatur

M. Ajtai, L. Babai, P. Hajnal, J. Komlós, P. Pudlák, V. Rödl, E. Szemerédi und G. Turán (1986): Two lower bounds for branching programs. Proceedings of the Annual ACM Symposium on Theory of Computing, S. 30–38.

B. Becker und R. Drechsler (1993): On the computational power of functional decision diagrams. Technischer Bericht Universität Frankfurt.

B. Becker und R. Drechsler (1995): How many decomposition types do we need? European Design and Test Conference, S. 438–443.

B. Becker, R. Drechsler und R. Werchner (1995): On the Relation Between BDDs and FDDs. Information and Computation 123, S. 185–197.

M. Blum, A. K. Chandra und M. N. Wegmann (1980): Equivalence of free Boolean graphs can be decided probabilistically in polynomial time. Information Processing Letters 10(2), S. 80–82.

B. Bollig (1993): Eine exponentielle untere Schranke für BP1s. unveröffentlicht.

B. Bollig, M. Löbbing, M. Sauerhoff und I. Wegener (1995): Complexity Theoretical Aspects of OFDDs. Proceedings of the Workshop on Applications of the Reed-Muller Expansion in Circuit Design, IFIP WG 10.5, S. 198–205.

R. E. Bryant (1986): Graph-based algorithms for Boolean function manipulation. IEEE Transactions on Computers, C-35, S. 677–691.

R. E. Bryant (1991): On the complexity of VLSI implementations and graph representations of Boolean functions with applications to integer multiplication. IEEE Transactions on Computers, C-40(2), S. 205–213.

R. E. Bryant (1992): Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. ACM Computing Surveys, 24(3), S. 293–318.

R. E. Bryant (1995): Binary Decision Diagrams and Beyond: Enabling Technologies for Formal Verification. IEEE/ACM International Conference on Computer Aided Design.

R. Hüskes (1995): Intels Tauschaktion. c't, Magazin für Computertechnik, Februar 1995, S. 17.

U. Kebschull und W. Rosenstiel (1993): Efficient graph-based computation and manipulation of functional decision diagrams. Proceedings of the European Design Automation Conference, S. 278–282.

U. Kebschull, E. Schubert und W. Rosenstiel (1992): Multilevel logic synthesis based on functional decision diagrams. Proceedings of the European Design Automation Conference, S. 43–47.

C. Y. Lee (1959): Representation of switching circuits by binary-decision programs. Bell System Technical Journal, S. 985–999.

S. Minato (1993): Zero-suppressed BDDs for set manipulation in combinatorial problems. Proc. 30. ACM/IEEE Design Automation Conference, S. 272–277.

S. Minato (1994): Calculation of unate cube set algebra using zero-suppressed BDDs. Proc. 31. ACM/IEEE Design Automation Conference, S. 420-424.

S. Minato, N. Ishiura und S. Yajima (1990): Shared binary decision diagrams with attributed edges for efficient Boolean function manipulation. Proceedings of the 27th ACM/IEEE Design Automation Conference, S. 52–57.

S. Panda und F. Somenzi (1995): Who are the Variables in Your Neighborhood. International Workshop on Logic Synthesis IWLS '95, S. 5.11–5.20.

V. Pratt (1995): Anatomy of the Pentium Bug. TAPSOFT 6, S. 97–107.

P. Savický und I. Wegener (1994): Efficient Algorithms for the Transformation between Different Types of Binary Decision Diagrams. Proc. of 14th Conference on the Foundations of Software Technology and Theoretical Computer Science, Lecture Notes in Computer Science 880, S. 390–401.

G. Schnurer (1993): Pentium-Rätselraten. c't, Magazin für Computertechnik, April 1993, S. 128–130.

O. Schröer und I. Wegener (1994): The Theory of Zero-suppressed BDDs and the Number of Knights Tours, Forschungsbericht Informatik Nr. 552, Universität Dortmund.

D. Sieling (1995): Algorithmen und untere Schranken für verallgemeinerte OBDDs. Aachen.

D. Sieling und I. Wegener (1995): Graph driven BDD's — A new data structure for Boolean functions. Theoretical Computer Science 141, S. 283–310.

D. Sieling und I. Wegener (1993): Reduction of OBDDs in linear time. Information Processing Letters 48, S. 139–144.

I. Wegener (1993): Skript zur Vorlesung „Theorie des Logikentwurfs", Sommersemester 1993, Universität Dortmund.

I. Wegener (1994): Efficient Data Structures for Boolean Functions. Discrete Mathematics 136, S. 347–372.

I. Wegener (1994/95): Skript zur Vorlesung „Effiziente Datenstrukturen für Boolesche Funktionen", Sommersemester 1994 und Wintersemester 1994/95, Universität Dortmund.

S. Yang (1991): Logic Synthesis and Optimization Benchmarks User Guide Version 3.0. Technical Report, MCNC, International Workshop of Logic Synthesis.

Tabellenverzeichnis

1 Übersicht der BDD-Modelle 35
2 Verwendete Benchmark-Schaltkreise 79
3 Größen der verwendeten Testfunktionen 80
4 Maßzahlen der Transformationen 82
5 Zusätzliche Benchmark-Schaltkreise 85
6 Größe der zusätzlichen Benchmarkfunktionen 86
7 Maßzahlen der zusätzlichen Transformationen 87
8 Hybridalgorithmen 88
9 Vergleich der Robustheit 89
10 Vergleich der relativen Abweichungen 89
11 Anzahl der look-ups der „reinen“ Verfahren 91
12 Anzahl der look-ups der Hybridverfahren mit Faktor 1,0 92
13 Anzahl der look-ups der Hybridverfahren mit Faktor 1,5 93
14 Übersicht aller Transformationen 103

Abbildungsverzeichnis

1 Zwei verschiedene FBDDs für die Funktion $x_1 \wedge x_2$ 6
2 Ein FBDD für f 7
3 Funktionsrepräsentation im OBDD 8
4 Funktionsrepräsentation im ZBDD 22
5 Funktionsrepräsentation im OFDD 28
6 Inverse Eliminationsregel für OFDDs 32
7 Inverse Eliminationsregel für OBDDs 46
8 Reduziertes OBDD 48
9 OBDD nach vollständigem Durchlauf durch alle L_i 48
10 Quasi-reduziertes OBDD 49
11 Inverse Eliminationsregel für ZBDDs 56
12 Reduziertes ZBDD für $\overline{x_i}$ 58