

Begleitmaterial zur Vorlesung

Binary Decision Diagrams

Sommersemester 2003

Detlef Sieling

RWTH Aachen

FB Informatik

Ahornstr. 55

52074 Aachen

Von diesem Begleitmaterial dürfen einzelne Ausdrücke oder Kopien für private Zwecke hergestellt werden. Jede weitere Verwertung außerhalb der engen Grenzen des Urheberrechtsgesetzes ist ohne besondere Zustimmung des Autors unzulässig.

Inhaltsverzeichnis

1	Einleitung und Überblick	3
1.1	Motivation und Definitionen	3
1.2	Anforderungen an Datenstrukturen für boolesche Funktionen	6
1.3	Beziehungen zwischen Branchingprogrammen und anderen nichtuniformen Berechnungsmodellen	8
1.4	Literatur	9
2	Ordered Binary Decision Diagrams	10
2.1	Die Reduktion und die Eindeutigkeit von reduzierten OBDDs	11
2.2	OBDD-Darstellungen von ausgewählten Funktionen	16
2.3	Das Variablenordnungsproblem	21
2.4	Algorithmen auf OBDDs	24
2.5	Schlussbemerkungen	33
3	Methoden zum Beweis unterer Schranken für BDDs	35
3.1	Untere Schranken für BDDs ohne weitere Einschränkungen	35
3.2	Beweis unterer Schranken mit Kommunikationskomplexität	37
4	Anwendungen	46
4.1	Verifikation des Pentium-Dividierers	46
4.2	Verifikation von sequentiellen Schaltkreisen	50
4.2.1	Flip-Flops und das Huffman-Modell für sequentielle Schaltkreise	50
4.2.2	Die Erreichbarkeitsanalyse	54

1 Einleitung und Überblick

1.1 Motivation und Definitionen

Ein Binary Decision Diagram (BDD) oder Branchingprogramm (BP) ist ein Graph, der einen Algorithmus zur Berechnung einer booleschen Funktion beschreibt. Boolesche Funktionen und Darstellungen für boolesche Funktionen haben eine zentrale Rolle in der Informatik, weil viele Probleme in der Informatik durch boolesche Funktionen beschrieben werden können und natürlich auch die Hardware boolesche Funktionen realisiert. In der Komplexitätstheorie versucht man, den Aufwand für die Berechnung von booleschen Funktionen in verschiedenen Berechnungsmodellen möglichst genau anzugeben. Hierbei werden auch BDDs untersucht, da es Simulationen zwischen BDDs und allen anderen nichtuniformen sequentiellen Berechnungsmodellen gibt. Damit können Ergebnisse über BDDs auf alle anderen nichtuniformen sequentiellen Berechnungsmodelle übertragen werden.

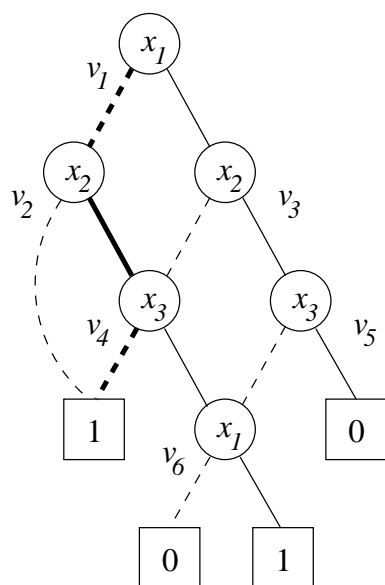
Darstellungen von booleschen Funktionen, wie auch von Mengen und Relationen werden auch in vielen Programmen benötigt. Mengen und Relationen können dabei durch ihre charakteristischen Funktionen dargestellt werden, die wiederum boolesche Funktionen sind. Varianten von BDDs werden insbesondere in Programmen für Hardwareentwurf und -verifikation, Logiksynthese, Analyse von endlichen Automaten oder Testmuster-generierung benutzt (siehe z.B. Bryant (1992) und Wegener (1993a)). Srinivasan, Kam, Malik und Brayton (1990) haben vorgeschlagen, Probleme wie die Kanalverdrahtung oder das Färben von Graphen durch boolesche Funktionen zu codieren und mit Hilfe von BDDs zu lösen. Bevor wir derartige Anwendungen genauer untersuchen und die Anforderungen dieser Anwendungen an Datenstrukturen für boolesche Funktionen beschreiben, wollen wir zunächst BDDs/Branchingprogramme definieren und einige Beispiele betrachten. Wir merken hierbei an, dass im Bereich der Anwendungen der Begriff BDD und im Bereich der Komplexitätstheorie der Begriff Branchingprogramm jeweils gängiger ist, obwohl beide Begriffe dieselbe Bedeutung haben. Zur Vereinheitlichung verwenden wir überwiegend den Begriff BDD.

BDDs wurden bereits von Lee (1959) über if-then-else Programme definiert. Wir benutzen die äquivalente, aber anschaulichere Definition über gerichtete Graphen. Ein BDD ist ein gerichteter azyklischer Graph mit einem Startknoten, auch Quelle genannt. Der Graph enthält nur innere Knoten mit Ausgangsgrad 2 und Senken mit Ausgangsgrad 0. Die inneren Knoten sind mit einer Variablen markiert, und die beiden ausgehenden Kanten sind mit 0 bzw. 1 markiert. Die Senken sind ebenfalls mit 0 oder 1 markiert.

Die Berechnung der dargestellten Funktion für eine gegebene Eingabe beginnt an der Quelle. Wenn ein erreichter Knoten mit der Variablen x_i markiert ist, wird entsprechend dem Wert von x_i über die mit 0 oder die mit 1 markierte Kante ein neuer Knoten erreicht. Dieses wird iteriert, bis man eine Senke erreicht. Die Markierung dieser Senke ist der gesuchte Funktionswert.

Abb. 1 zeigt ein Beispiel für ein BDD und das äquivalente if-then-else Programm. Dieses BDD stellt die Funktion $f(x_1, x_2, x_3) = \bar{x}_2 \vee \bar{x}_3$ dar. Bei der graphischen Darstellung benutzen wir die Vereinbarung, dass Kanten stets nach unten gerichtet sind und dass die gestrichelte Kante, die einen Knoten verlässt, mit 0 markiert ist und die durchgezogene Kante

mit 1. Jede Eingabe für die Funktion f definiert im BDD einen Pfad von der Quelle zu einer Senke. Der im BDD in Abb. 1 markierte Pfad ist der Pfad, der für die Eingabe $x_1 = 0$, $x_2 = 1$, $x_3 = 0$ durchlaufen wird. Es gilt also $f(0, 1, 0) = 1$.



- 1: if x_1 then goto 3 else goto 2;
- 2: if x_2 then goto 4 else goto 8;
- 3: if x_2 then goto 5 else goto 4;
- 4: if x_3 then goto 6 else goto 8;
- 5: if x_3 then goto 7 else goto 6;
- 6: if x_1 then goto 8 else goto 7;
- 7: Ausgabe 0;
- 8: Ausgabe 1;

Abbildung 1: Beispiel für ein BDD und das äquivalente if-then-else-Programm

Wichtige Komplexitätsmaße für BDDs sind die Größe, d.h. die Anzahl der inneren Knoten, und die Tiefe, d.h. die Länge des längsten gerichteten Pfades von der Quelle zu einer Senke. BDDs sind zunächst nur für boolesche Funktionen mit einer festen Anzahl von Eingabebits definiert. Damit wir auch Funktionen f mit einer unbestimmten Anzahl von Eingabebits beschreiben können und das asymptotische Verhalten von Größe und Tiefe untersuchen können, zerlegen wir — wie in der Komplexitätstheorie üblich — die Funktion f in eine Folge (f_n) , wobei jede Funktion f_n eine endliche Funktion mit n Eingabebits ist. Die Funktion f wird dann durch eine Folge von BDDs beschrieben. Für jede Länge der Eingabe gibt es also ein spezielles BDDs, d.h., BDDs sind ein nichtuniformes Berechnungsmodell für boolesche Funktionen.

Was ist der wesentliche Unterschied zwischen uniformen Berechnungsmodellen (z.B. Turingmaschinen, Registermaschinen oder dem Maschinenmodell für C-Programme) und nicht-uniformen Berechnungsmodellen (z.B. Schaltkreise oder BDDs)? Bei uniformen Berechnungsmodellen erwarten wir, dass die Arbeitsweise für alle Eingabelängen „ähnlich“ ist, z.B. arbeitet Quicksort auf Eingaben verschiedener Länge im wesentlichen gleichartig. Bei Schaltkreisen müssen wir für jede Eingabelänge einen separaten Schaltkreis angeben. Auch wenn bei den gewöhnlichen Schaltkreiskonstruktionen die Schaltkreise für verschiedene Eingabelängen ähnlich aussehen, besteht nicht die Notwendigkeit, dass dies immer so ist. Betrachten wir z.B. das Halteproblem. Eingabe sind die Codierung einer Turingmaschine M und eine Eingabe x , und die Frage besteht darin, ob M auf x hält. Bekanntermaßen ist das Halteproblem nicht entscheidbar. Wir können aber M und x binär codieren und die Funktion $f^H(M, x)$ definieren, die genau den Wert 1 annimmt, wenn M auf x hält. Wenn wir nun diese Funktion in Teilfunktionen für jede Eingabelänge zerlegen, erhalten wir Funktionen f_n^H ,

die auf Eingaben der Länge n definiert sind. Da jede boolesche Funktion eine Darstellung z.B. als disjunktive Normalform hat, gibt es Schaltkreise für die nicht berechenbare Funktion f^H . Der Begriff der Berechenbarkeit ist also für nichtuniforme Berechnungsmodelle nicht sinnvoll, wobei natürlich klar sein sollte, dass es die *Schaltkreise* für f^H zwar gibt, diese aber nicht berechnet werden können. Bei vielen (aber nicht allen) Schaltkreisen ist es allerdings effizient möglich, zu jeder Eingabelänge einen Schaltkreis zu berechnen; in diesem Fall spricht man daher auch von *uniformen* Schaltkreisen.

Die Berechnung von Funktionswerten kann man nicht nur an der Quelle des BDDs, sondern an jedem beliebigen Knoten beginnen. Auf diese Weise erhält man für jeden Knoten v eine Funktion f_v . An den Knoten des BDDs in Abb. 1 werden die folgenden Funktionen berechnet:

$$\begin{aligned} f_{v_1} &= \bar{x}_2 \vee \bar{x}_3 \\ f_{v_2} &= x_1 \vee \bar{x}_2 \vee \bar{x}_3 \\ f_{v_3} &= x_1 \bar{x}_2 \vee \bar{x}_2 \bar{x}_3 \vee x_1 x_2 \bar{x}_3 \\ f_{v_4} &= x_1 \vee \bar{x}_3 \\ f_{v_5} &= x_1 \bar{x}_3 \\ f_{v_6} &= x_1 \end{aligned}$$

Für jeden Knoten v , der mit der Variablen x markiert ist und der die Nachfolger v_0 und v_1 hat, gilt $f_v = \bar{x}f_{v_0} \vee xf_{v_1}$, an einer 0-Senke wird die Nullfunktion und an einer 1-Senke die Einsfunktion berechnet. Auf diese Weise können wir die durch ein BDD dargestellte Funktion bestimmen ohne die Funktion für alle Eingaben auszuwerten.

Wir zeigen nun, dass jede boolesche Funktion $f(x_1, \dots, x_n)$ durch ein BDD dargestellt werden kann. Wir zerlegen f mit der so genannten Shannon-Zerlegung $f = \bar{x}_1 f_{|x_1=0} \vee x_1 f_{|x_1=1}$ und erzeugen einen mit x_1 markierten Knoten v . Am 0-Nachfolger dieses Knotens muss die Funktion $f_{|x_1=0}$ dargestellt werden; dazu berechnen wir rekursiv ein BDD für diese Funktion und wählen dessen Quelle als 0-Nachfolger von v . Analog berechnen wir für den 1-Nachfolger ein BDD für $f_{|x_1=1}$. Abb. 2 zeigt ein BDD, das auf diese Weise für $f = \bar{x}_2 \vee \bar{x}_3$ berechnet wird. Mit dieser Prozedur erhalten wir immer ein BDD, in dem alle Knoten den Eingangsgrad 1 haben. Derartige BDDs heißen Entscheidungs bäume, weil der zugrundeliegende Graph ein Baum ist.

Wenn wir auf diese Weise einen Entscheidungsbaum für eine Funktion $f : \{0, 1\}^n \rightarrow \{0, 1\}$ konstruieren, hat dieser Entscheidungsbaum $2^n - 1$ innere Knoten. In BDDs dürfen Knoten einen größeren Eingangsgrad als 1 haben; daher dürfen wir isomorphe Teilgraphen verschmelzen (siehe Abb. 2). BDDs sind daher eine kompaktere Darstellung als Entscheidungs bäume; man kann zeigen, dass jede boolesche Funktion mit BDDs der Größe $O(2^n/n)$ dargestellt werden kann. Andererseits ist bekannt, dass für fast alle booleschen Funktionen, d.h. einen Anteil von $1 - o(1)$ von allen booleschen Funktionen, die minimale Größe eines BDDs $\Omega(2^n/n)$, also exponentiell in n ist, siehe Abschnitt 3.

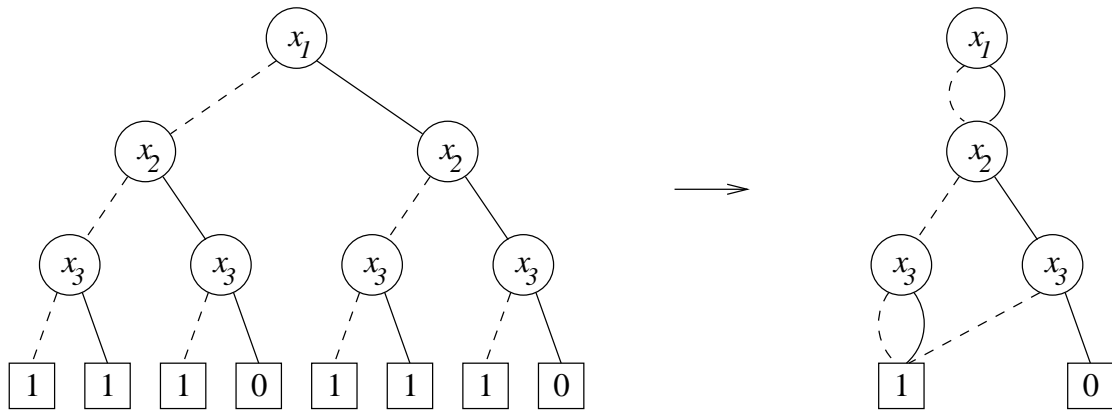


Abbildung 2: Ein Entscheidungsbaum und ein BDD für $\bar{x}_2 \vee \bar{x}_3$

1.2 Anforderungen an Datenstrukturen für boolesche Funktionen

Wir wollen zunächst eine der Anwendungen von BDDs kurz skizzieren um zu sehen, welche Anforderungen an Datenstrukturen für boolesche Funktionen gestellt werden.

Bei der Hardwareverifikation soll für einen gegebenen Schaltkreis getestet werden, ob er die gewünschte Funktion realisiert. Die gewünschte Funktion ist durch eine Spezifikation oder durch einen anderen Schaltkreis gegeben. Für den Test, ob die gewünschte und die realisierte Funktion gleich sind, kann man für beide Funktionen Darstellungen (in unserem Fall eine bestimmte Variante von BDDs) berechnen und auf Gleichheit testen. Der Test, ob zwei Schaltkreise dieselbe Funktion berechnen, ist jedoch co-NP-vollständig. Das Beste, was man erwarten kann, ist also, dass die Umrechnung der Schaltkreise in die gewünschte Darstellung und der Äquivalenztest für praktisch relevante Funktionen effizient durchführbar sind.

Für die Umrechnung eines Schaltkreises in die gewählte Darstellung wird man den Schaltkreis in einer topologischen Ordnung durchlaufen. Wenn man einen Baustein C erreicht, der zwei Funktionen f_1 und f_2 mit einem binären Operator \otimes verknüpft, muss man aus den Darstellungen für f_1 und f_2 eine Darstellung für $f_1 \otimes f_2$ berechnen. Diese Operation nennen wir Synthese. Die Synthese wird bei Umrechnungen von Schaltkreisen in die gewählte Darstellung für jeden Baustein einmal ausgeführt, sollte also besonders effizient ausführbar sein. Wenn der Schaltkreis aus Modulen aufgebaut ist, kann man auch zuerst Darstellungen für die Funktionen, die durch die Module realisiert werden, berechnen und aus diesen hinterher eine Darstellung für den Schaltkreis. Dazu benötigt man eine Operation, die aus Darstellungen für Funktionen f und g eine Darstellung für $f|_{x_i=g}$ berechnet. Da die Rechenzeit für die einzelnen Operationen von der Größe der Darstellung abhängt, sollte es außerdem effizient möglich sein, die Größe der Darstellung zu minimieren.

Wenn der untersuchte Schaltkreis nicht die gewünschte Funktion g , sondern eine fehlerhafte Funktion h realisiert, kann es für die Korrektur hilfreich sein, zu wissen, auf wievielen und auf welchen Eingaben der Schaltkreis falsch arbeitet. Es müssen also die Mächtigkeit und die Elemente der Menge $(g \oplus h)^{-1}(1)$ berechnet werden.

Zusammen mit Anforderungen aus anderen Anwendungen erhalten wir die im folgenden definierten Operationen auf booleschen Funktionen. Wir bezeichnen mit B_n die Menge aller booleschen Funktionen über n Variablen.

1. Auswertung: Gegeben sei eine Darstellung G für $f \in B_n$ und eine Eingabe $a \in \{0, 1\}^n$. Berechne $f(a)$.
2. Erfüllbarkeit: Gegeben sei eine Darstellung G für $f \in B_n$. Gibt es eine Eingabe $a \in \{0, 1\}^n$ mit $f(a) = 1$?
3. Erfüllbarkeit-Anzahl: Gegeben sei eine Darstellung G für $f \in B_n$. Berechne $|f^{-1}(1)|$.
4. Erfüllbarkeit-Alle: Gegeben sei eine Darstellung G für $f \in B_n$. Gib $f^{-1}(1)$ aus.
5. Reduktion/Minimierung: Gegeben sei eine Darstellung G für $f \in B_n$. Berechne für die gleiche Funktion eine Darstellung G' minimaler Größe. Falls G' für jede Funktion f eindeutig definiert ist, heißt diese Operation Reduktion.
6. Äquivalenztest: Gegeben seien Darstellungen G_f und G_g für $f, g \in B_n$. Teste, ob $f = g$.
7. Synthese: Gegeben seien Darstellungen G_f und G_g für $f, g \in B_n$ und eine Operation $\otimes \in B_2$. Berechne eine Darstellung für $f \otimes g$.
8. Ersetzung durch Konstanten: Gegeben sei eine Darstellung G für $f \in B_n$, eine Variable x_i und eine Konstante c . Berechne eine Darstellung für $f_{|x_i=c}$.
9. Ersetzung durch Funktionen: Gegeben seien Darstellungen G_f und G_g für $f, g \in B_n$ und eine Variable x_i . Berechne eine Darstellung für $f_{|x_i=g}$.
10. Quantifizierung: Gegeben sei eine Darstellung G für $f \in B_n$ und eine Variable x_i . Berechne eine Darstellung für $(\forall x_i : f) := f_{|x_i=0} \wedge f_{|x_i=1}$ (bzw. für $(\exists x_i : f) := f_{|x_i=0} \vee f_{|x_i=1}$).
11. Redundanztest: Gegeben sei eine Darstellung G für $f \in B_n$ und eine Variable x_i . Teste, ob die Variable x_i redundant ist, d.h., ob $f_{|x_i=0} = f_{|x_i=1}$ gilt.

Die Darstellungen sollten möglichst klein sein, um Speicherplatz und Rechenzeit zu sparen. Da es 2^{2^n} boolesche Funktionen über n Variablen gibt, muss jede Darstellung für fast alle booleschen Funktionen exponentielle Größe haben. Daher können wir nur erreichen, dass möglichst viele praktisch wichtige Funktionen kompakt darstellbar sind.

Bekannte Darstellungen für boolesche Funktionen sind neben BDDs und Schaltkreisen auch Formeln und Wertetabellen. Die Größe von Wertetabellen ist für alle booleschen Funktionen exponentiell in der Anzahl der Variablen. Also können nur Funktionen über sehr wenigen Variablen dargestellt werden. Für Schaltkreise und Formeln ist der Äquivalenztest co-NP-vollständig und das Erfüllbarkeitsproblem NP-vollständig. Das gleiche gilt für uneingeschränkte BDDs. Dagegen haben sich eingeschränkte Varianten von BDDs, insbesondere OBDDs (Ordered Binary Decision Diagrams, Bryant (1985, 1986)) als praktisch anwendbar erwiesen. Im Abschnitt 2 werden wir daher OBDDs definieren und Algorithmen für die wichtigen Operationen auf booleschen Funktionen für OBDDs beschreiben.

1.3 Beziehungen zwischen Branchingprogrammen und anderen nicht-uniformen Berechnungsmodellen

Bekannte nichtuniforme Berechnungsmodelle sind Schaltkreise, Formeln und nichtuniforme Turingmaschinen. Es ist klar, dass eine Funktion $f = (f_n)$, $f_n : \{0, 1\}^n \rightarrow \{0, 1\}$, durch eine Folge von Schaltkreisen oder Formeln repräsentiert werden kann, wobei es für jede Länge der Eingabe einen speziellen Schaltkreis oder eine spezielle Formel gibt. Nichtuniforme Turingmaschinen enthalten dagegen ein Orakelband, das zu Beginn einer Rechnung zusätzliche Informationen enthält, die nur von der Länge der Eingabe abhängen dürfen. Über die folgenden Simulationen (Cobham (1966) und Pudlák und Žák (1983)) zwischen BDDs und nichtuniformen Turingmaschinen erhalten wir Beziehungen zwischen der Größe von BDDs und dem Speicherbedarf von nichtuniformen sequentiellen Berechnungsmodellen.

Wir beschreiben zuerst die Simulation einer nichtuniformen Turingmaschine durch ein BDD. Für jede Konfiguration der Turingmaschine enthält das BDD einen Knoten; die Startkonfiguration entspricht dabei der Quelle. Jeder Knoten wird mit der Variablen markiert, die die Turingmaschine in der zugehörigen Konfiguration vom Eingabeband liest, die ausgehende Kante mit der Markierung 0 zeigt auf den Knoten, der die Nachfolgekonfiguration repräsentiert, wenn das gelesene Eingabebit gleich 0 ist. Analog wird die mit 1 markierte Kante definiert. Akzeptierende Endkonfigurationen werden zu 1-Senken, nicht akzeptierende Endkonfigurationen zu 0-Senken.

Der Berechnungspfad im BDD entspricht für jede Eingabe genau der Konfigurationsfolge der Turingmaschine, daher ist die Simulation korrekt. Die Größe des BDDs ist gleich der Anzahl der Konfigurationen der Turingmaschine. Wenn der Speicherplatzbedarf der Turingmaschine $s(n)$ ist, ist die Zahl der Konfigurationen und damit die Größe des BDDs durch $\max\{O(n), 2^{O(s(n))}\}$ beschränkt. Zugleich entspricht die Tiefe des BDDs der Rechenzeit der Turingmaschine.

Da wir bei dieser Simulation keine speziellen Eigenschaften der Turingmaschine benutzt haben, kann auf die gleiche Weise jedes andere nichtuniforme sequentielle Berechnungsmodell durch BDDs simuliert werden. Damit implizieren untere Schranken für die Größe und die Tiefe von BDDs untere Schranken für den Speicherplatz und die Rechenzeit von allen anderen nichtuniformen sequentiellen Berechnungsmodellen.

Auch die umgekehrte Simulation ist einfach. Die nichtuniforme Turingmaschine erhält das zu simulierende BDD auf dem Orakelband. Die Rechnung des BDDs kann direkt simuliert werden, wobei in jedem Schritt nur ein Zeiger auf den gerade erreichten Knoten gespeichert werden muss. Der Speicherplatzbedarf ist daher $O(\log \max\{BDD_f(n), n\})$, wobei $BDD_f(n)$ die Größe des BDDs bezeichnet. Diese Simulation zeigt, dass auch obere Schranken für die Größe von BDDs obere Schranken für den Speicherplatzbedarf von nichtuniformen sequentiellen Berechnungsmodellen implizieren.

Die Simulationen motivieren die Suche nach oberen und unteren Schranken für die Größe von BDDs für vorgegebene Funktionen. \mathcal{L} ist die Menge der booleschen Funktionen, die nichtuniform mit logarithmischem Speicherplatz berechnet werden können. Aus den Simulationen folgt, dass die Menge der booleschen Funktionen mit BDDs polynomieller Größe gleich \mathcal{L} ist. Wenn wir zeigen wollen, dass eine Funktion nicht in \mathcal{L} enthalten ist, genügt es,

zu zeigen, dass BDDs für diese Funktion superpolynomielle Größe haben. Die beste bekannte untere Schranke für die Größe von BDDs wurde bereits 1966 von Nečiporuk gezeigt und ist nur von der Größenordnung $\Omega(n^2 / \log^2 n)$. Man versucht daher, bessere untere Schranken für eingeschränkte Varianten von BDDs zu zeigen und diese Einschränkungen nach und nach abzuschwächen.

Es wurden auch Methoden für den Beweis unterer Schranken für die eingeschränkten Varianten von BDDs entwickelt, die als Datenstrukturen für boolesche Funktionen verwendet werden. Solche Schranken sind hilfreich, um zu bestimmen, welche Funktionen kompakt dargestellt werden können oder für welche Funktionen welche Variante von BDDs geeignet ist.

Die Simulationen zwischen BDDs und Schaltkreisen bzw. Formeln wollen wir nicht ausführen. Aus diesen Simulationen folgen Beziehungen zwischen der Größe BDD_f von BDDs der Schaltkreisgröße C_f und der Formelgröße L_f für boolesche Funktionen f . Es gilt

$$\frac{1}{3}C_f \leq BDD_f \leq L_f.$$

Also können alle booleschen Funktionen durch Schaltkreise dargestellt werden, die von der Größenordnung her mindestens so kompakt sind wie BDDs. Diese sind wiederum eine kompaktere Darstellung als Formeln.

1.4 Literatur

Drechsler, R. und Sieling, D. (2001). Binary decision diagrams in theory and practice. Int. Journal on Software Tools for Technology Transfer 3, 112–136.

Hachtel, G. und Somenzi, F. (1996). Logic synthesis and verification algorithms. Kluwer.

Minato, S. (1996). Binary decision diagrams and applications for VLSI CAD. Kluwer.

Wegener, I. (2000). Branching programs and binary decision diagrams, theory and applications. SIAM Monographs on Discrete Mathematics and Applications.

2 Ordered Binary Decision Diagrams

Fortune, Hopcroft und Schmidt haben bereits 1978 eine Variante von Ordered Binary Decision Diagrams (OBDDs) untersucht und gezeigt, dass Synthese und Äquivalenztest für OBDDs effizient möglich sind. Bryant (1985, 1986) war der Erste, der die Anwendbarkeit von OBDDs als Datenstruktur für boolesche Funktionen erkannte. Bevor wir auf Algorithmen für die in der Einleitung genannten Operationen eingehen, wollen wir OBDDs definieren und einige Eigenschaften von OBDDs kennenlernen.

Definition Ein OBDD ist ein BDD, in dem auf jedem Pfad alle Variablen höchstens einmal und gemäß einer vorgegebenen Ordnung getestet werden. D.h., es gibt eine Permutation $\pi \in S_n$, und für jede Kante, die von einem mit x_i markierten Knoten zu einem mit x_j markierten Knoten führt, gilt $\pi(i) < \pi(j)$.

Abb. 3 zeigt zwei OBDDs für die Funktion $f(x_1, x_2, x_3) = x_1x_2 \vee x_3$ mit den Variablenordnungen x_1, x_2, x_3 und x_1, x_3, x_2 . Obwohl beide OBDDs dieselbe Funktion darstellen, hat das eine OBDD drei, das andere aber vier innere Knoten. Wir werden später noch sehen, dass die Variablenordnung entscheidenden Einfluss auf die Größe eines OBDDs haben kann. Das BDD aus Abb. 1 ist dagegen kein OBDD, weil z.B. auf dem Pfad v_1, v_2, v_4, v_6 die Variable x_1 zweimal getestet wird.

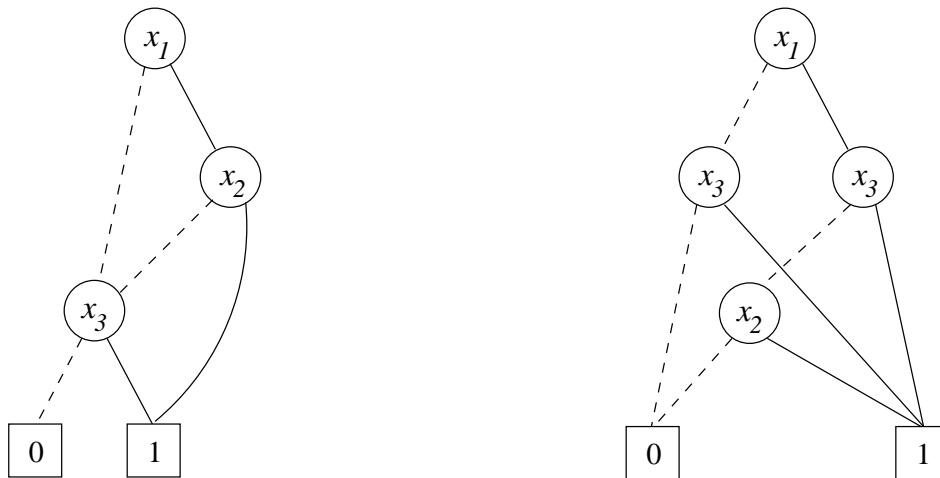


Abbildung 3: OBDDs für $f(x_1, x_2, x_3) = x_1x_2 \vee x_3$

In Abschnitt 1 haben wir gezeigt, dass es für jede boolesche Funktion ein BDD gibt, indem wir aus einer gegebenen booleschen Funktion einen Entscheidungsbaum konstruiert haben. Dieser Entscheidungsbaum ist auch ein OBDD, daher gibt es auch für jede boolesche Funktion ein OBDD. Weil OBDDs stark eingeschränkte BDDs sind, gibt es jedoch Funktionen, die mit BDDs kompakt, mit OBDDs aber nur in exponentieller Größe darstellbar sind.

2.1 Die Reduktion und die Eindeutigkeit von reduzierten OBDDs

Da wir mit Darstellungen für boolesche Funktionen möglichst effizient arbeiten wollen, sollte die Darstellung möglichst klein sein. Wir wollen in diesem Abschnitt zwei Reduktionsregeln kennenlernen, mit denen die Größe von BDDs verkleinert werden kann. Bei OBDDs haben die Reduktionsregeln eine besondere Bedeutung. Wir fixieren eine Variablenordnung und betrachten ein beliebiges OBDD für eine Funktion f mit dieser Variablenordnung. Wenn wir auf dieses OBDD die Reduktionsregeln anwenden, bis keine der Regeln mehr anwendbar ist, ist das entstandene OBDD (bis auf Isomorphie) eindeutig, und zwar unabhängig davon, mit welchem OBDD für f wir begonnen haben und in welcher Reihenfolge wir die Reduktionsregeln angewandt haben. Diese Eigenschaft macht den Äquivalenztest und den Erfüllbarkeitstest für OBDDs besonders einfach.

Für die erste Reduktionsregel betrachten wir die Situation in Abb. 4. Sei v ein Knoten in einem BDD, der mit x_i markiert ist und für den die 0-Kante und die 1-Kante auf denselben Knoten w zeigen. Wenn wir den Funktionswert für eine Eingabe berechnen und dabei den Knoten v erreichen, gehen wir unabhängig vom Wert von x_i zum Knoten w weiter. Der Test von x_i ist also überflüssig, und wir können das BDD vereinfachen, indem wir den Knoten v löschen und die Kanten, die zu v führen, direkt zu w zeigen lassen. Diese Reduktionsregel heißt *Deletion Rule*.

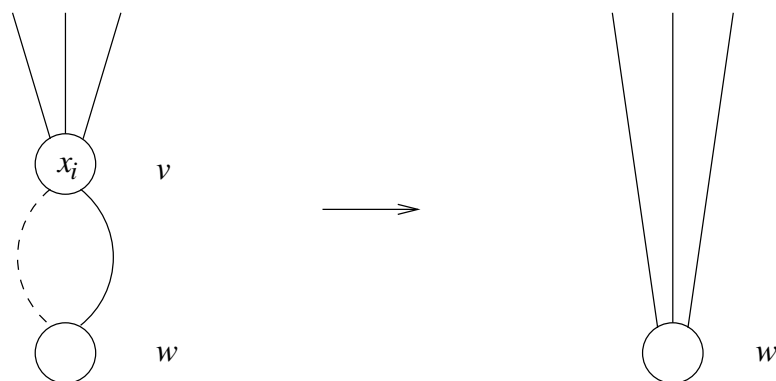


Abbildung 4: Deletion Rule

Die zweite Reduktionsregel, die *Merging Rule*, ist anwendbar, wenn es zwei Knoten v und w gibt, die mit der gleichen Variablen markiert sind und denselben 0-Nachfolger und denselben 1-Nachfolger haben (s. Abb. 5). Es ist klar, dass wir denselben Funktionswert erhalten unabhängig davon, ob wir am Knoten v oder am Knoten w starten, also dass $f_v = f_w$ gilt. Wir können also die Knoten v und w verschmelzen. Analog können wir Senken, die mit der gleichen Konstanten markiert sind, verschmelzen. Durch jede Anwendung einer Reduktionsregel wird die Zahl der Knoten im BDD um 1 verringert.

Bei der Simulation von nichtuniformen sequentiellen Berechnungsmodellen durch BDDs haben wir gesehen, dass die Knoten des BDDs Konfigurationen des sequentiellen Berechnungsmodells repräsentieren. Daher können beide Reduktionsregeln als Entfernen überflüssiger Konfigurationen interpretiert werden, d.h., es wird weniger überflüssige Information über bereits getestete Variablen gespeichert.

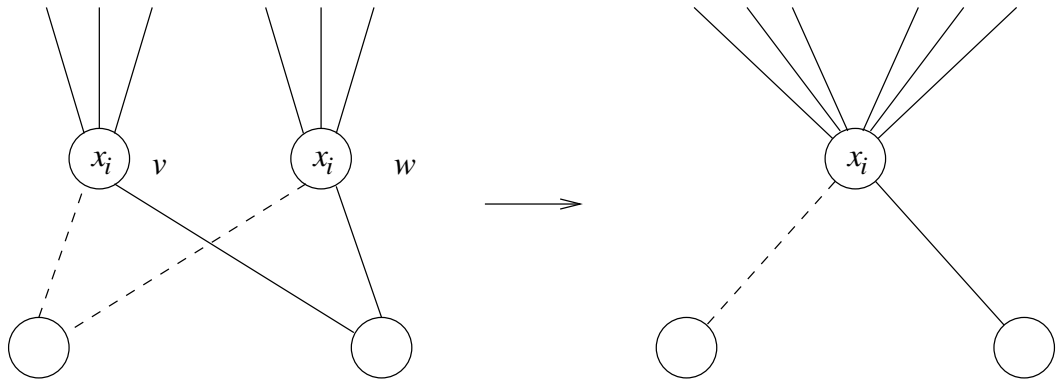


Abbildung 5: Merging Rule

Abb. 6 zeigt ein Beispiel für eine Reduktion in mehreren Schritten. Zunächst wird der mit x_4 markierte Knoten mit der Deletion Rule entfernt. Dann können die mit x_3 markierten Knoten verschmolzen werden, und dann kann die Deletion Rule auf den mit x_2 markierten Knoten angewandt werden. Man sieht an diesem Beispiel Abhängigkeiten zwischen den Reduktionsschritten, d.h., manchmal ist eine Regel erst anwendbar, nachdem andere Reduktionsschritte durchgeführt worden sind.

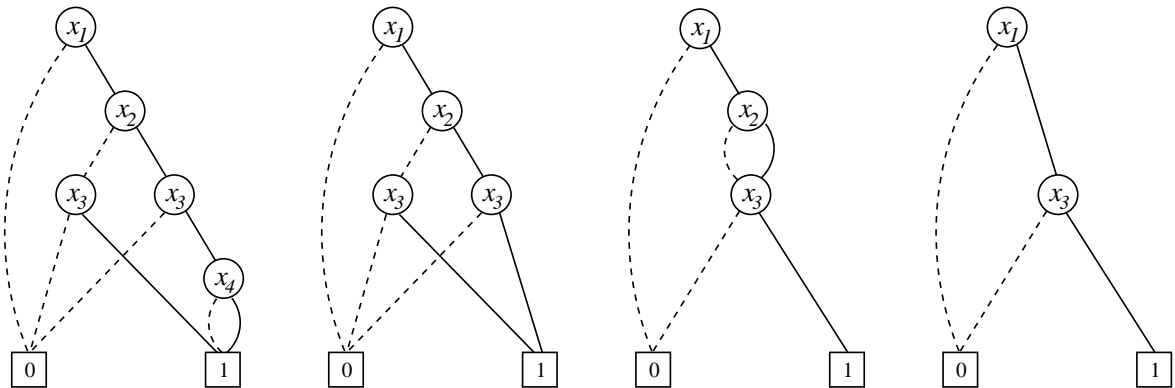


Abbildung 6: Beispiel für eine Reduktion

Wir wollen nun zeigen, dass es sinnvoll ist, von *dem* reduzierten OBDD für eine Funktion f und eine Variablenordnung π zu reden. Zuerst zeigen wir, dass die Funktionen, die an den Knoten eines OBDDs für f berechnet werden, bestimmte Subfunktionen von f sind. Über diese Subfunktionen können OBDDs minimaler Größe für f beschrieben werden. Wir können dann beweisen, dass alle minimalen OBDDs für f isomorph sind. Der Einfachheit halber sei im Rest dieses Abschnitts die Variablenordnung x_1, \dots, x_n . Dieses ist keine Einschränkung, denn wir können diese Ordnung aus jeder Ordnung erhalten, indem wir die Variablen umbenennen.

Sei v ein Knoten in einem OBDD für die Funktion f . Der Knoten v sei mit x_i markiert. Sei P ein Pfad, der von der Quelle zu v führt. Auf P dürfen nur die Variablen x_1, \dots, x_{i-1} getestet werden. Der Pfad P wird durchlaufen, wenn die Variablen, die auf P getestet werden, passende Werte haben, wenn also $x_1 = c_1, \dots, x_{i-1} = c_{i-1}$ für geeignete Konstanten

c_1, \dots, c_{i-1} gilt. Am Knoten v und an Knoten unterhalb von v dürfen nur die Variablen x_i, \dots, x_n getestet werden. Daher wird an v die Funktion $f|_{x_1=c_1, \dots, x_{i-1}=c_{i-1}}$ berechnet. Sollte es außer P noch einen Pfad Q geben, der von der Quelle zu v führt und der für $x_1 = d_1, \dots, x_{i-1} = d_{i-1}$ durchlaufen wird, wird an v zugleich die Funktion $f|_{x_1=d_1, \dots, x_{i-1}=d_{i-1}}$ berechnet. Da im OBDD an v und an Knoten unterhalb von v nur die Variablen x_i, \dots, x_n getestet werden dürfen, gilt $f|_{x_1=c_1, \dots, x_{i-1}=c_{i-1}} = f|_{x_1=d_1, \dots, x_{i-1}=d_{i-1}}$. An jedem mit x_i markierten Knoten wird also genau eine Subfunktion von f berechnet, die wir aus f erhalten können, indem wir x_1, \dots, x_{i-1} auf geeignete Weise konstantsetzen.

Wir können nun minimale OBDDs für jede boolesche Funktion und jede Variablenordnung beschreiben (Sieling und Wegener (1993a)). Der folgende Satz ist zwar nur für die Variablenordnung x_1, \dots, x_n formuliert, gilt aber für alle Variablenordnungen, da man die Variablen umbenennen kann.

Satz 2.1.1 Sei S_i die Menge der Subfunktionen von f , die wir erhalten, indem wir x_1, \dots, x_{i-1} konstantsetzen, und die essentiell von x_i abhängen. Es gibt bis auf Isomorphie genau ein OBDD minimaler Größe für f und die Variablenordnung x_1, \dots, x_n . Dieses OBDD enthält genau $|S_i|$ Knoten, die mit x_i markiert sind.

Beweis: Wir konstruieren zunächst ein OBDD für f , das für jedes i genau $|S_i|$ Knoten enthält, die mit x_i markiert sind. Wenn f eine konstante Funktion ist, besteht das OBDD aus einer Senke, die mit der entsprechenden Konstanten markiert ist. Anderenfalls enthält das OBDD eine 0-Senke, eine 1-Senke und für jede Subfunktion $g \in S_i$ genau einen Knoten v , der mit x_i markiert ist. Seien $g_0 := g|_{x_i=0}$ und $g_1 := g|_{x_i=1}$. Für diese Funktionen enthält das OBDD die Knoten v_0 und v_1 . Jeder dieser Knoten ist entweder eine Senke, wenn g_0 bzw. g_1 eine konstante Funktion ist, oder ein mit x_j markierter innerer Knoten, wobei $j > i$ gilt. Wir wählen als 0-Nachfolger von v den Knoten v_0 und als 1-Nachfolger v_1 .

Das konstruierte OBDD berechnet die Funktion f . Dazu genügt es zu zeigen, dass an jedem Knoten v die zugehörige Subfunktion berechnet wird. Wir beweisen diese Aussage mit Induktion, wobei wir die Knoten in einer umgekehrten topologischen Ordnung durchlaufen. Die Aussage gilt für die Senken, weil an den Senken die konstanten Funktionen berechnet werden. Für einen inneren Knoten v , der mit x_i markiert ist und an dem die Funktion g berechnet werden soll, gilt nach Induktionsannahme, dass am 0-Nachfolger die Funktion $g|_{x_i=0}$ und am 1-Nachfolger die Funktion $g|_{x_i=1}$ berechnet wird. Also wird an v die Funktion g berechnet.

Wenn das konstruierte OBDD nicht minimale Größe hat, gibt es ein OBDD für f , das für ein i weniger als $|S_i|$ Knoten enthält, die mit x_i markiert sind. Da es aber $|S_i|$ verschiedene Subfunktionen von f der Form $f|_{x_1=c_1, \dots, x_{i-1}=c_{i-1}}$ gibt, die essentiell von x_i abhängen, gibt es eine Zuweisung $c = (c_1, \dots, c_{i-1})$ zu x_1, \dots, x_{i-1} , für die $f|_{x_1=c_1, \dots, x_{i-1}=c_{i-1}} \in S_i$ gilt, so dass der zugehörige Pfad entweder zu einer Senke führt oder zu einem mit x_j markierten Knoten, wobei $j > i$, oder zu einem mit x_i markierten Knoten, an dem eine andere Subfunktion als $f|_{x_1=c_1, \dots, x_{i-1}=c_{i-1}}$ berechnet wird. Die ersten beiden Fälle führen zum Widerspruch, weil $f|_{x_1=c_1, \dots, x_{i-1}=c_{i-1}}$ essentiell von x_i abhängt, der letzte Fall, weil in einem OBDD an jedem Knoten nur eine Subfunktion berechnet werden kann.

Jedes minimale OBDD für f und die Variablenordnung x_1, \dots, x_n muss also für jede Subfunktion $g \in S_i$ einen mit x_i markierten Knoten enthalten, dessen Nachfolger Knoten sind, an denen $g|_{x_i=0}$ bzw. $g|_{x_i=1}$ berechnet wird. Ein OBDD für f , das zu dem oben konstruierten nicht isomorph ist, muss daher zusätzliche Knoten enthalten, ist also nicht minimal. \square

Die Eigenschaft, dass es für jede Funktion f und jede Variablenordnung π nur ein minimales OBDD gibt, kann nur dann praktisch genutzt werden, wenn das minimale OBDD effizient aus jedem anderen OBDD für f und π berechnet werden kann. Wir zeigen zunächst, dass wir das minimale OBDD erhalten, wenn wir ein gegebenes OBDD solange reduzieren, bis keine der Reduktionsregeln mehr anwendbar ist.

Lemma 2.1.2 Sei G ein OBDD für die Funktion f und die Variablenordnung π . G ist genau dann isomorph zum minimalen OBDD G^* für f und π , wenn auf G keine der Reduktionsregeln anwendbar ist.

Beweis: Auf ein minimales OBDD ist offensichtlich keine Reduktionsregel anwendbar; daher genügt es zu zeigen, dass auf einem OBDD G , das nicht minimal ist, eine der Regeln anwendbar ist. Sei f o.B.d.A. keine konstante Funktion. Wir gehen davon aus, dass G nicht mehrere 0- oder 1-Senken enthält, da diese sonst verschmolzen werden können. Außerdem setzen wir wieder voraus, dass die Variablenordnung x_1, \dots, x_n ist.

Aus dem Beweis von Satz 2.1.1 folgt, dass jedes OBDD für f für jede Subfunktion in S_i mindestens einen Knoten enthält, der mit x_i markiert ist. Wenn G zum minimalen OBDD G^* nicht isomorph ist, gibt es mindestens ein i , so dass G mehr als $|S_i|$ Knoten enthält, die mit x_i markiert sind. Sei i^* das größte derartige i . Dann werden in G alle Subfunktionen in S_j mit $j > i^*$ und die beiden konstanten Funktionen durch genau einen Knoten dargestellt. Da es mehr als $|S_i|$ mit x_i markierte Knoten gibt, enthält G entweder mindestens einen Knoten u , der mit x_i markiert ist und an dem eine Subfunktion g berechnet wird, die von x_i nicht essentiell abhängt, oder es gibt zwei Knoten v und w , die mit x_i markiert sind und an denen die gleiche Subfunktion $h \in S_i$ berechnet wird. Im ersten Fall gilt $g = g|_{x_i=0} = g|_{x_i=1}$. Da die Funktionen $g|_{x_i=0}$ und $g|_{x_i=1}$ am gleichen Knoten dargestellt werden, kann der Knoten u mit der Deletion Rule entfernt werden. Im zweiten Fall werden an den Nachfolgern von v und w die Funktionen $h|_{x_i=0}$ bzw. $h|_{x_i=1}$ berechnet. Für jede dieser Funktionen gibt es in G nur einen Knoten, daher können v und w aufgrund der Merging Rule verschmolzen werden. \square

Wir wollen jetzt zeigen, wie aus einem OBDD für die Funktion f und die Variablenordnung π , o.B.d.A. x_1, \dots, x_n , das minimale OBDD berechnet werden kann. Es genügt zwar, solange Reduktionsregeln anzuwenden, bis keine Regel mehr anwendbar ist, wir sollten aber verhindern, dass für einen Knoten häufig geprüft werden muss, ob eine der Regeln anwendbar ist. Der Beweis von Lemma 2.1.2 legt nahe, die Reduktion bottom-up durchzuführen, weil für Knoten endgültig entschieden werden kann, ob sie gelöscht oder mit anderen verschmolzen werden können, wenn auf ihre Nachfolger keine Regel mehr angewandt werden kann.

Jeder innere Knoten v des OBDDs G wird als Record gespeichert, das eine Knotennummer $v.id$, den Index $v.var$ der Variablen, mit der v markiert ist, und Zeiger $v.null$ und $v.eins$ auf den 0- und den 1-Nachfolger von v enthält. Da wir keine rückwärts gerichteten Zeiger haben, können Knoten des OBDDs nicht direkt gelöscht werden. Wenn wir einen Knoten v mit der Deletion Rule löschen wollen, müssen die Zeiger auf v auf den Nachfolger von v umgesetzt werden; um einen Knoten v mit einem anderen Knoten w zu verschmelzen, müssen die Zeiger auf v auf den Knoten w umgesetzt werden. Wir realisieren das Löschen, indem wir v die Nummer seines Nachfolgers bzw. die Nummer von w geben. Da es jetzt mehrere Knoten mit derselben Nummer gibt, speichern wir in einem zusätzlichen Array Z zu jeder Nummer, welcher Knoten mit dieser Nummer die übrigen repräsentiert. Wir erhalten den folgenden Algorithmus (Bryant (1986)).

Algorithmus 2.1.3

Eingabe: Ein OBDD G für f mit der Ordnung x_1, \dots, x_n .

Ausgabe: Das minimale OBDD G^* für f und die Ordnung x_1, \dots, x_n .

- Durchlaufe G von der Quelle mit einem depth first search Ansatz. Dabei
 - verschmelze alle 0-Senken zu einer 0-Senke und alle 1-Senken zu einer 1-Senke, und setze

$0\text{-Senke}.id := 0; Z[0\text{-Senke}.id] := 0\text{-Senke};$

$1\text{-Senke}.id := 1; Z[1\text{-Senke}.id] := 1\text{-Senke};$

- ordne alle mit x_i markierten Knoten in die Liste S_i ein ($i = 1, \dots, n$).
- $id := 1$.
- FOR $i := n$ DOWNTO 1 DO
 - *Anwendung der Deletion Rule:* Suche in S_i alle Knoten v , für die $v.null.id = v.eins.id$ gilt. Lösche v aus dem OBDD durch die Zuweisung $v.id := v.null.id$, und entferne v aus der Liste S_i .
 - Sortiere die Liste S_i , wobei jeder Knoten v als Schlüssel das Paar $(v.null.id, v.eins.id)$ erhält. Als Ordnung benutzen wir z.B. die lexikographische Ordnung auf den Paaren.
 - *Anwendung der Merging Rule:* Knoten, die verschmolzen werden können, stehen in der sortierten Liste hintereinander; daher muss die Liste nur einmal durchlaufen werden, um diese Knoten zu erkennen. Ein Knoten v_1 , der im reduzierten OBDD enthalten sein soll, bekommt die nächste freie Knotennummer, d.h.

$id := id + 1; v_1.id := id; Z[v_1.id] := v_1;$

Der 0- und der 1-Nachfolger von v_1 können gelöscht worden sein. Damit die Zeiger auf die Nachfolger von v_1 nicht auf gelöschte Knoten, sondern auf die nicht gelöschten Repräsentanten zeigen, setzen wir

$v_1.null := Z[v_1.null.id]; v_1.eins := Z[v_1.eins.id].$

Der Knoten v_1 kann nun mit anderen Knoten v_2, \dots, v_l durch die Zuweisungen

$$v_2.id := v_1.id, \dots, v_l.id := v_1.id$$

verschmolzen werden.

- Entferne die gelöschten Knoten.

Alle Operationen außer dem Sortieren benötigen pro Knoten nur konstante Zeit, insgesamt also $O(|G|)$. Bryant (1986) benutzt für das Sortieren ein allgemeines Sortierverfahren, die Rechenzeit beträgt daher $O(\sum_{i=1}^n |S_i| \log |S_i|) = O(|G| \log |G|)$. Für jeden Knoten v müssen $v.id$, $v.var$, $v.null$ und $v.eins$ gespeichert werden, außerdem benötigen wir das Array Z und die Zeiger auf die Nachfolger in den Listen S_i . Der Speicherbedarf beträgt dann $6|G| + O(n)$, wobei der Platz für die Eingabe mitgezählt wird. Die Zahl n der Variablen ist typischerweise viel kleiner als die Zahl der Knoten im OBDD G . Wir merken nur an, dass die Reduktion auch in linearer Zeit möglich ist (Sieling und Wegener (1993b)).

2.2 OBDD-Darstellungen von ausgewählten Funktionen

In diesem Abschnitt wollen wir reduzierte OBDDs für einige spezielle Funktionen angeben, um ein Gefühl dafür zu erhalten, wie OBDDs arbeiten und in welchen Fällen ihr Einsatz sinnvoll ist. Weiterhin wird deutlich, dass die Wahl der Variablenordnung von OBDDs ein wichtiges Problem ist.

Wir beginnen mit OBDDs für symmetrische Funktionen. Der Wert einer symmetrischen Funktion hängt nur von der Anzahl der Einsen in der Eingabe ab, nicht aber von den Positionen dieser Einsen. Daher kann jede symmetrische Funktion in B_n durch einen Wertevektor (v_0, \dots, v_n) dargestellt werden, wobei die Funktion den Wert v_i annimmt, wenn die Eingabe genau i Einsen enthält. Abbildung 7 zeigt ein (nicht reduziertes) OBDD für die symmetrische Funktion mit dem Wertevektor (v_0, \dots, v_4) . Man verifiziert leicht, dass die mit v_i markierte Senke genau für die Eingaben mit i Einsen erreicht wird. Die Verallgemeinerung für symmetrische Funktionen über n Variablen ist einfach. Dabei entsteht ein OBDD quadratischer Größe. (Auf der i -ten Ebene gibt es i Knoten, so dass die Gesamtzahl der Knoten gleich $\sum_{i=1}^{n+1} i = O(n^2)$ ist.) Je nach Wertevektor sind noch Verschmelzungen von Knoten möglich, so dass das OBDD auch viel kleiner werden kann, z.B. linear für die Parity-Funktion auf n Variablen. Da sich symmetrische Funktionen nicht verändern, wenn Variablen vertauscht werden, hängt die OBDD-Größe von symmetrischen Funktionen nicht von der Variablenordnung ab.

Das nächste Beispiel ist die Funktion DQF_n (disjunkte quadratische Form), die über den $2n$ Variablen x_1, \dots, x_{2n} durch

$$DQF_n(x_1, \dots, x_{2n}) = x_1 x_2 \vee \dots \vee x_{2n-1} x_{2n}$$

definiert ist. Wir betrachten die zwei Variablenordnungen $x_1, x_2, x_3, \dots, x_{2n}$ und $x_1, x_3, x_5, \dots, x_{2n-1}, x_2, x_4, \dots, x_{2n}$. OBDDs für beide Variablenordnungen und DQF_6 sind in Abbildung 8 gezeigt. Man sieht sofort, dass die erste Variablenordnung zu einem viel

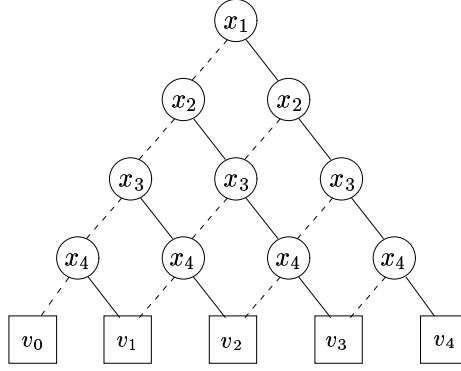


Abbildung 7: Ein OBDD für symmetrische Funktionen auf 4 Variablen

kleineren OBDD als die zweite Variablenordnung führt. Wir wollen nun die OBDD-Größen berechnen und beginnen mit der ersten Variablenordnung. Offensichtlich hat das OBDD einen Knoten, der mit x_1 markiert ist und an dem DQF_n berechnet wird. Am 0-Nachfolger v_0 dieses Knotens wird die Funktion

$$f^0 = \text{DQF}_{n|x_1=0} = x_3x_4 \vee \dots \vee x_{2n-1}x_{2n}$$

und am 1-Nachfolger v_1 die Funktion

$$f^1 = \text{DQF}_{n|x_1=1} = x_2 \vee x_3x_4 \vee \dots \vee x_{2n-1}x_{2n}$$

berechnet. Offensichtlich hängt die an v_0 berechnete Funktion nicht essentiell von x_2 ab, wohl aber von x_3 , so dass v_0 mit x_3 markiert ist. Dagegen ist v_1 mit x_2 markiert. Am 0-Nachfolger von v_1 wird die Funktion $f^1_{|x_2=0} = f^0$ berechnet, und am 1-Nachfolger von v_1 die Funktion $f^1_{|x_2=1} = 1$. Die 1-Kante von v_1 zeigt also zur 1-Senke. Insgesamt folgt, dass es jeweils einen mit x_1 und mit x_2 markierten Knoten gibt. Eine der ausgehenden Kanten zeigt zur 1-Senke und die andere zu einem Knoten, an dem $f^0 = \text{DQF}_{n-1}(x_3, \dots, x_{2n})$ berechnet wird. Also ist die OBDD-Größe $S(n)$ von DQF_n gleich der Summe von 2 und der OBDD-Größe $S(n-1)$ von DQF_{n-1} . Wegen $S(1) = 4$ folgt, dass reduzierte OBDDs von DQF_n für die erste Variablenordnung genau $2n + 2$ Knoten haben.

Der Einfachheit halber wollen wir die OBDD-Größe von DQF_n für die zweite Variablenordnung nur abschätzen. Dazu zählen wir nur die mit x_2 markierten Knoten. Die gesamte OBDD-Größe ist dann auf jeden Fall größer als die berechnete Anzahl von x_2 -Knoten. Nach Satz 2.1.1 ist die Anzahl von x_2 -Knoten gleich der Anzahl der Subfunktionen von DQF_n , die durch Konstantsetzen der Variablen $x_1, x_3, x_5, \dots, x_{2n-1}$ entstehen und die von x_2 essentiell abhängen. Sei

$$F[c_1, c_3, \dots, c_{2n-1}](x_2, x_4, \dots, x_{2n})$$

die Subfunktion von DQF_n , die entsteht, wenn $x_1, x_3, x_5, \dots, x_{2n-1}$ durch $c_1, c_3, c_5, \dots, c_{2n-1}$ ersetzt wurden. Sei $J(c_1, c_3, \dots, c_{2n-1})$ die Menge der ungeraden Zahlen j mit $c_j = 1$. Offensichtlich ist

$$F[c_1, c_3, \dots, c_{2n-1}](x_2, x_4, \dots, x_{2n}) = \bigvee_{j \in J(c_1, c_3, \dots, c_{2n-1})} x_{j+1}.$$

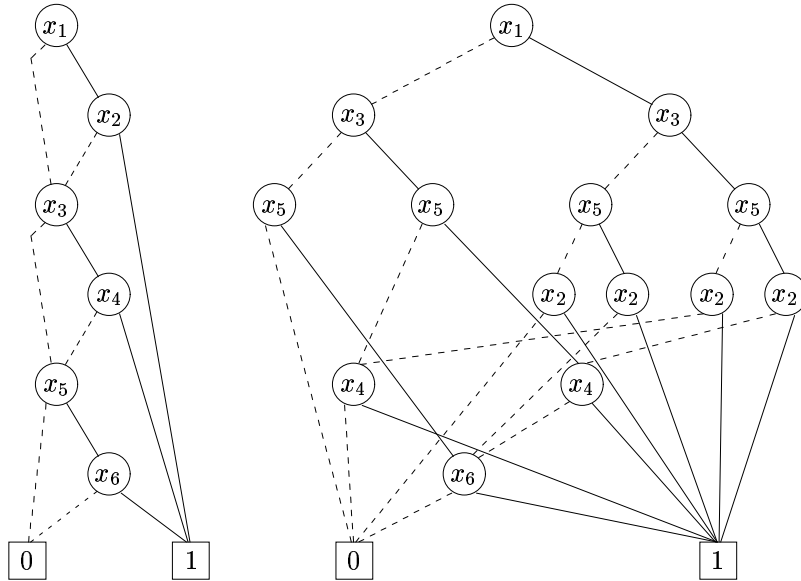


Abbildung 8: OBDDs für die Funktion DQF

Die Funktion hängt essentiell von x_2 ab, wenn $1 \in J$, also wenn $c_1 = 1$. Offensichtlich sind für verschiedene Belegungen von $c_1, c_3, \dots, c_{2n-1}$ die Funktionen $F[c_1, c_3, \dots, c_{2n-1}]$ verschieden. Es gibt 2^{n-1} Belegungen von $c_1, c_3, \dots, c_{2n-1}$ mit $c_1 = 1$. Also hat das OBDD 2^{n-1} mit x_2 markierte Knoten. Wir bemerken, dass man die OBDD-Größe von DQF_n für die zweite Variablenordnung auch genau ausrechnen kann; sie beträgt 2^{n+1} . Wir erhalten dieselbe OBDD-Größe für jede Variablenordnung, für die für jedes Paar (x_i, x_{i+1}) , i ungerade, eine der beiden Variablen in der ersten Hälfte der Variablenordnung und die andere in der zweiten Hälfte der Variablenordnung angeordnet ist.

Die Wahl der Variablenordnung kann also zwischen polynomieller und exponentieller Größe eines OBDDs entscheiden. Aus praktischer Sicht kann dies bedeuten, dass eine Anwendung in dem einen Fall erfolgreich beendet werden kann, während sie in dem anderen Fall wegen Speichermangel abgebrochen werden muss (man vergleiche die OBDD-Größen von DQF_n und die beiden o.g. Variablenordnungen für $n = 100$). Bei der Funktion DQF ist intuitiv klar, dass die Variablen x_i und x_{i+1} für ungerades i „zusammengehören“ und daher auch in der Variablenordnung zusammenstehen sollten.

Leider haben auch sehr wichtige Funktionen wie die Addition die Eigenschaft, dass die Variablenordnung zwischen polynomieller und exponentieller OBDD-Größe entscheiden kann. Wir bezeichnen im Folgenden das $(i-1)$ -te Bit der Summe von zwei i -Bit Zahlen x_{i-1}, \dots, x_0 und y_{i-1}, \dots, y_0 mit ADD_i . Mit ähnlichen Methoden wie bei der Funktion DQF kann gezeigt werden, dass bei einer Variablenordnung, bei der alle x -Variablen vor allen y -Variablen getestet werden, die OBDD-Größe für ADD_i mindestens $2^{\Omega(i)}$ beträgt (Übungsaufgabe). Wir betrachten hier nur die (natürlicheren) Variablenordnungen $x_0, y_0, \dots, x_{i-1}, y_{i-1}$ und $x_{i-1}, y_{i-1}, \dots, x_0, y_0$. OBDDs für die beiden Variablenordnungen sind in Abb. 9 gezeigt. Es ist leicht, analog zur Funktion DQF zu verifizieren, dass diese OBDDs wirklich die Funktion ADD_i darstellen. Bei beiden Variablenordnungen haben die OBDDs lineare Größe. Dennoch

gibt es einen Unterschied. Bei der zweiten Variablenordnung stimmen die unteren Teile der OBDDs für ADD_i und ADD_j für $i \neq j$ überein. Hieraus folgt, dass alle Bits der Summe zweier n -Bit Zahlen, also die Funktionen $\text{ADD}_i, i \in \{0, \dots, n-1\}$ (und auch das Übertragsbit), zusammen in einem OBDD linearer Größe dargestellt werden können. Man überzeugt sich leicht, dass bei der ersten Variablenordnung quadratische Größe erforderlich ist.

Schließlich gibt es auch viele Funktionen, für die OBDDs für jede Variablenordnung exponentielle Größe haben. Hierzu gehört z.B. die Multiplikation (genauer gesagt, die Funktion MUL_n , die das n -te Bit des Produktes zweier n -Bit Zahlen beschreibt). Wir wollen hier eine „künstliche“ Funktion mit dieser Eigenschaft vorstellen, um zu motivieren, warum es überhaupt passieren kann, dass Funktionen nur exponentielle OBDDs haben. Wir werden eine Methode vorstellen, mit der man beweisen kann, dass eine Funktion für alle Variablenordnungen exponentielle OBDDs hat. Mit ähnlichen Beweismethoden kann man zeigen, dass MUL_n nur exponentielle OBDDs hat; allerdings erfordert dieser Beweis mehr Tricks und Techniken.

Wir definieren nun die Funktion ACH , eine Variante der sogenannten Achillesfersenfunktion von Ashar, Ghosh und Devadas (1991). Sei $n = 2^k$. Für die Variablen s_{k-1}, \dots, s_0 bezeichne $|s|$ den Wert von (s_{k-1}, \dots, s_0) als Binärzahl interpretiert. Die Funktion ACH_n ist auf $2n + k$ Variablen $x_0, \dots, x_{n-1}, y_0, \dots, y_{n-1}, s_0, \dots, s_{k-1}$ definiert:

$$\text{ACH}_n(x, y, s) = x_0 y_{|s|} \vee x_1 y_{(|s|+1) \bmod n} \vee \dots \vee x_{n-1} y_{(|s|+n-1) \bmod n}.$$

Die Funktion wird also folgendermaßen ausgewertet: Zuerst wird y_0, \dots, y_{n-1} um $|s|$ Positionen nach links rotiert. Anschließend wird die Funktion DQF angewendet, wobei jedes Paar aus einer x -Variablen und einer y -Variablen besteht. Die Idee für den Beweis, dass ACH für jede Variablenordnung exponentielle Größe hat, besteht darin, dass man für jede Variablenordnung einen Wert für $|s|$ finden kann, für den viele Paare von x - und y -Variablen, deren Konjunktion zu berechnen ist, in der Variablenordnung getrennt werden. Dies wollen wir nun formalisieren.

Sei ein OBDD G für ACH_n gegeben. Wir betrachten nur die x - und y -Variablen, die gemäß der Variablenordnung von G in einer Folge angeordnet sind. Sei L die Menge der vorderen n Variablen dieser Folge, und sei R die Menge der hinteren n Variablen dieser Folge. O.B.d.A. enthält L eine Menge L' von mindestens $n/2$ x -Variablen (und damit R eine Menge R' von mindestens $n/2$ y -Variablen). Anderenfalls können wir den folgenden Beweis ebenfalls anwenden, müssen aber die Rollen der x - und y -Variablen vertauschen.

Sei $t \in \{0, \dots, n-1\}$. Sei $J(t)$ die Menge der Paare $(x_i, y_{(i+t) \bmod n})$, für die $x_i \in L'$ und $y_{(i+t) \bmod n} \in R'$. Die Menge $J(0) \cup \dots \cup J(n-1)$ enthält mindestens $n^2/4$ solche Paare (jedes Paar einer x -Variablen aus L' und einer y -Variablen aus R' kommt in dieser Menge vor). Dann gibt es ein t , für das $|J(t)| \geq n/4$. (Wären alle Mengen $J(\cdot)$ kleiner, hätte die Vereinigung dieser Mengen nicht $n^2/4$ Elemente.) Wir belegen nun die s -Variablen so, dass $|s| = t$, und berechnen das OBDD für die entsprechende Subfunktion von ACH . Wie oben erwähnt, ist dies die Funktion DQF . Sei $J(t) \supseteq \{(x_{i(1)}, y_{j(1)}), \dots, (x_{i(n/4)}, y_{j(n/4)})\}$. Wir setzen alle x -Variablen, die nicht in $\{x_{i(1)}, \dots, x_{i(n/4)}\}$ enthalten sind, und alle y -Variablen, die nicht in $\{y_{j(1)}, \dots, y_{j(n/4)}\}$ enthalten sind, auf 0. Nach Konstruktion entsteht die Subfunktion

$$x_{i(1)} y_{j(1)} \vee \dots \vee x_{i(n/4)} y_{j(n/4)},$$

wobei alle x -Variablen vor allen y -Variablen angeordnet sind. Nach den Betrachtungen zur Funktion DQF folgt, dass das OBDD G' für diese Subfunktion und eine solche Variablenordnung $2^{n/4+1}$ Knoten hat. Es ist leicht einzusehen, dass sich OBDDs beim Konstantsetzen von Variablen nicht vergrößern (siehe auch Abschnitt 5.5). Da das OBDD G' aus G durch Konstantsetzen von Variablen entstanden ist, folgt, dass auch G mindestens $2^{n/4+1}$ Knoten hat.

Wir haben schon bei der Funktion DQF gesehen, dass zusammengehörende Variablen zusammen in der Variablenordnung angeordnet sein sollten. Bei der Funktion ACH entscheidet die Belegung der s -Variablen, welche x - und y -Variablen zusammengehören, so dass es nicht möglich ist, eine Variablenordnung zu finden, bei der alle zusammengehörenden Paare dicht beieinander angeordnet sind. Als Faustregel kann man sagen, dass Funktionen, bei denen jede Variable mit vielen anderen zusammengehört, häufig nur exponentielle OBDDs haben.

2.3 Das Variablenordnungsproblem

An den Beispielen im letzten Abschnitt haben wir gesehen, dass die Wahl der Variablenordnung zwischen polynomieller und exponentieller OBDD-Größe entscheiden kann. Daher ist es wichtig, gute Algorithmen für das Variablenordnungsproblem zu haben. Man unterscheidet zwei verschiedene Varianten des Variablenordnungsproblems. Bei der ersten ist die darzustellende Funktion f durch einen Schaltkreis gegeben. Die Aufgabe besteht darin, ein möglichst kleines OBDD für f zu konstruieren. Wir bemerken nur, dass dies ein NP-hartes Problem ist und dass nur Heuristiken für dieses Problem bekannt sind. Die Heuristiken versuchen, aus der Schaltkreisbeschreibung Zusammenhänge zwischen der Variablen zu extrahieren und zusammengehörende Variablen zusammen anzuordnen. Ein Beispiel für eine solche Heuristik besteht darin, den Schaltkreis von den Ausgängen mit einem DFS-Durchlauf zu durchlaufen und die Variablen in der Reihenfolge anzuordnen, wie sie beim DFS-Durchlauf gefunden werden. Man überlegt leicht, dass bei der Funktion DQF z.B. aus der Polynomdarstellung in jedem Fall eine optimale Variablenordnung gefunden wird.

Bei der zweiten Variante des Variablenordnungsproblem geht man davon aus, dass die darzustellende Funktion bereits durch ein OBDD gegeben ist. Formal ist dieses Problem folgendermaßen definiert:

MinOBDD

Eingabe: Ein OBDD für f .

Ausgabe: Ein OBDD minimaler Größe für f .

Wir weisen noch einmal darauf hin, dass hier das Ziel ist, eine gute Variablenordnung zu berechnen, während bei der Reduktion die Variablenordnung fest ist. Die Motivation für das Problem MinOBDD ist die folgende: In der Einleitung haben wir beschrieben, wie ein Schaltkreis in ein OBDD umgerechnet werden kann. Im Laufe dieser Rechnung verändert sich die Menge der dargestellten Funktionen. Damit verändert sich auch die Menge der Variablenordnungen, bezüglich derer die Darstellung kompakt ist. Also ist es sinnvoll, im Laufe der Umformung die Variablenordnung zu verbessern. Dies ist genau das Problem MinOBDD. Die Idee, die Variablenordnung im Laufe von Berechnungen auf OBDDs zu ändern, wird in der Literatur als dynamische Variablenordnung bezeichnet (Rudell (1993)).

Leider gibt es auch für MinOBDD vermutlich keine effizienten Algorithmen. Bollig und Wegener (1996) haben bewiesen, dass das Problem NP-hart ist. Sieling (1998) hat bewiesen, dass auch die Existenz von polynomiellen Approximationsalgorithmen mit jeder konstanten Güte $c > 1$ impliziert, dass $P=NP$ ist. (Ein Approximationsalgorithmus mit Güte c für MinOBDD berechnet für alle Eingaben ein OBDD, das um höchstens den Faktor c größer als ein minimales OBDD ist.) Also müssen wir uns auch beim Problem MinOBDD mit Heuristiken zufrieden geben. Anstelle des Nichtapproximierbarkeitsergebnissen wollen wir hier nur das schwächere Ergebnis zeigen, dass die Berechnung einer optimalen Variablenordnung für ein OBDD, das mehrere Funktionen darstellt (diese nennt man auch SBDDs—shared BDDs), NP-hart ist. Wir betrachten also das Problem

MinSBDD

Eingabe: Ein SBDD für f_1, \dots, f_m .

Ausgabe: Ein SBDD minimaler Größe für f_1, \dots, f_m .

Satz 2.3.4 Falls es für MinSBDD einen polynomiellen Algorithmus gibt, folgt $P = NP$.

Beweis: Wir geben eine Turing-Reduktion von 3-SAT an, d.h., wir beweisen, dass 3-SAT einen polynomiellen Algorithmus hat, wenn MinSBDD einen polynomiellen Algorithmus hat. Das Problem 3-SAT ist folgendermaßen definiert:

3-SAT

Eingabe: Eine Menge U von Variablen und eine Menge C von Klauseln über den Variablen aus U , wobei (i) jede Klausel genau 3 Literale enthält, (ii) jede Variable in jeder Klausel höchstens einmal vorkommt und (iii) zwei Klauseln höchstens ein Literal gemeinsam haben.

Frage: Gibt es eine Belegung der Variablen, die alle Klauseln erfüllt?

Wir merken nur an, dass die Einschränkungen (ii) und (iii) an die Eingaben von 3-SAT in der Regel nicht gefordert werden, es ist aber eine einfache Übungsaufgabe zu zeigen, dass auch unsere Variante von 3-SAT NP-vollständig ist.

Sei (U, C) eine Eingabe für 3-SAT, d.h., $U = \{u_1, \dots, u_n\}$ ist eine Menge von Variablen und $C = \{C_1, \dots, C_m\}$ ist eine Menge von Klauseln über den Variablen. Die Frage ist, ob es eine Variablenbelegung gibt, für die alle Klauseln erfüllt sind.

Wir konstruieren nun ein SBDD. Das SBDD stellt die Funktionen $f_1, \dots, f_n, g_1, \dots, g_m$ dar, die auf den Variablen $x_1, \dots, x_n, y_1, \dots, y_n$ definiert sind. Dabei assoziieren wir x_i mit dem Literal u_i , und wir assoziieren y_i mit dem negierten Literal \bar{u}_i . Für die Funktionen wählen wir

$$f_i = x_i \wedge y_i \quad \text{für } 1 \leq i \leq n$$

und

$$g_j = \bigwedge_{u_i \in C_j} x_i \wedge \bigwedge_{\bar{u}_i \in C_j} y_i \quad \text{für } 1 \leq j \leq m.$$

Da jede Klausel genau 3 Literale enthält, ist g_j die Konjunktion von 3 Variablen. Wir zeigen zwei Lemmas über die SBDD-Größe der Funktionen $f_1, \dots, f_n, g_1, \dots, g_m$.

Lemma 2.3.5 Falls (U, C) erfüllbar ist, gibt es ein SBDD für $f_1, \dots, f_n, g_1, \dots, g_m$ mit höchstens $2n + 2m$ inneren Knoten.

Lemma 2.3.6 Falls (U, C) nicht erfüllbar ist, hat jedes SBDD für $f_1, \dots, f_n, g_1, \dots, g_m$ mit mindestens $2n + 2m + 1$ innere Knoten.

Wir zeigen zunächst, dass aus den Lemmas die Behauptung folgt. Falls es einen polynomiellen Algorithmus A für MinSBDD gibt, können wir einen polynomiellen Algorithmus B für 3-SAT konstruieren: Wir konstruieren aus der Eingabe (U, C) die Funktionen $f_1, \dots, f_n, g_1, \dots, g_m$ und daraus ein SBDD. Dies ist einfach möglich. Auf dieses SBDD wenden wir den Algorithmus A an. Resultat ist ein minimales SBDD für die Funktionen $f_1, \dots, f_n, g_1, \dots, g_m$. Falls dieses SBDD höchstens $2n + 2m$ innere Knoten hat, ist (U, C) erfüllbar und anderenfalls nicht. Also erhalten wir einen polynomiellen Algorithmus für 3-SAT. Da 3-SAT NP-hart ist, folgt die Behauptung.

Beweis von Lemma 2.3.5: Sei $X_0 = \{x_i | \sigma(u_i) = 0\} \cup \{y_i | \sigma(u_i) = 1\}$, also die Menge von Variablen, die mit den von σ nicht erfüllten Literalen über U assoziiert sind. Analog sei $X_1 = \{x_i | \sigma(u_i) = 1\} \cup \{y_i | \sigma(u_i) = 0\}$ die Menge von Variablen, die mit den von σ erfüllten Literalen assoziiert sind. Wir wählen nun eine beliebige Variablenordnung, bei der die Variablen in X_0 vor den Variablen in X_1 angeordnet sind, und bestimmen die Anzahl der inneren Knoten in einem SBDD für $f_1, \dots, f_n, g_1, \dots, g_m$ mit dieser Variablenordnung.

Jede Funktion f_i kann offensichtlich mit 2 inneren Knoten dargestellt werden und jede Funktion g_j mit 3 inneren Knoten. Da für jede Klausel mindestens ein Literal erfüllt ist, ist die letzte Variable x_k oder y_k (o.B.d.A. x_k), von der g_j essentiell abhängt, mit dem erfüllten Literal u_k assoziiert. Wegen der Wahl der Variablenordnung wird in dem OBDD für f_k die Variablen y_k vor x_k getestet. Damit können für die beiden x_k -Knoten verschmolzen werden. Da dies für jede Klausel geht, bleiben für die Funktionen g_j jeweils nur zwei innere Knoten übrig, d.h., die Anzahl der inneren Knoten beträgt $2n + 2m$. \square

Beweis von Lemma 2.3.6: Wir nehmen an, dass es ein SBDD für $f_1, \dots, f_n, g_1, \dots, g_m$ mit höchstens $2n + 2m$ inneren Knoten gibt und folgern hieraus, dass (U, C) erfüllbar ist. Sei also ein solches SBDD gegeben. Wir konstruieren eine Variablenbelegung σ . Falls x_i vor y_i in der Variablenordnung steht, wählen wir $\sigma(u_i) = 0$ und anderenfalls $\sigma(u_i) = 1$. Offensichtlich muss das SBDD $2n$ innere Knoten für die Darstellung von f_1, \dots, f_n enthalten, da diese Funktionen von insgesamt $2n$ Variablen essentiell abhängen. Wir überlegen nun, welche Verschmelzungen von Knoten des OBDDs für g_j mit anderen OBDDs möglich sind. Da in jeder Klausel drei verschiedene Variablen vorkommen, ist es nicht möglich, dass g_j von x_i und y_i essentiell abhängt. Also gibt es für die OBDDs für f_i und g_j maximal einen gemeinsamen Knoten. Analog folgt, dass die OBDDs für g_j und $g_{j'}$ maximal einen Knoten gemeinsam haben, da zwei Klauseln maximal ein Literal gemeinsam haben. D.h., für jede Funktion g_j gibt es im SBDD zwei innere Knoten, die nicht mit anderen inneren Knoten verschmolzen werden können. Da das SBDD nur $2n + 2m$ innere Knoten hat, muss es für jede Funktion g_j einen Knoten geben, der mit einem OBDD für eine Funktion f_i verschmolzen wird. (Wenn die OBDDs für g_j und $g_{j'}$ einen Knoten gemeinsam haben, haben sie zusammen 5 und nicht 4 innere Knoten.) Dann aber folgt sofort, dass die oben konstruierte Variablenbelegung σ erfüllend ist. \square

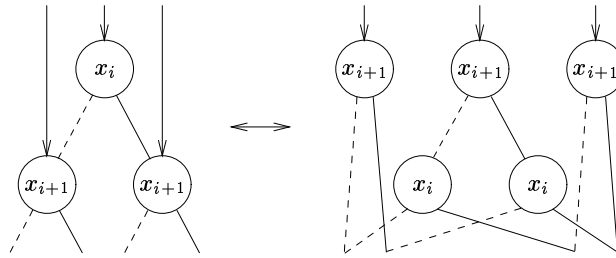


Abbildung 10: Ausführung der Operation $\text{SWAP}(x_i, x_{i+1})$

Eine elementare Operation beim Verändern der Variablenordnung von OBDDs ist das Vertauschen benachbarter Variablen, was auch als Swap bezeichnet wird: Sei x_1, \dots, x_n die Variablenordnung des gegebenen OBDDs. $\text{SWAP}(x_i, x_{i+1})$ verändert die Variablenordnung in $x_1, \dots, x_{i-1}, x_{i+1}, x_i, x_{i+2}, \dots, x_n$. Es folgt direkt aus Satz 2.1.1, dass sich nur die Ebenen mit den Variablen x_i und x_{i+1} verändern. In Abbildung 10 ist gezeigt, in welcher Weise die Kanten verändert werden müssen.

Eine naheliegende Heuristik ist eine lokale Suche. Wir nennen zwei Variablenordnungen benachbart, wenn sie sich durch eine Swap-Operation ineinander überführen lassen. Bei einer lokalen Suche probiert man alle möglichen Swap-Operationen für die aktuelle Variablenordnung aus. Wenn man eine Swap-Operation findet, bei der sich die OBDD-Größe verkleinert, wird die neue Variablenordnung zur aktuellen Variablenordnung. Dies wird solange iteriert, bis keine Verbesserung der OBDD-Größe mehr möglich ist. Dieser Algorithmus (und Verfeinerungen davon) wurde von Ishiura, Sawada und Yajima (1991) untersucht.

Es ist leicht einzusehen, dass die lokale Suche häufig in lokalen Optima „steckenbleibt“. Dasselbe gilt für den sogenannten Sifting-Algorithmus (Rudell (1993)), der eine lokale Suche mit einer anderen Nachbarschaftsbeziehung ist.

Beim Sifting-Algorithmus wird eine Variable x_i ausgewählt, und es wird nach einer optimalen Position für x_i gesucht. Dazu wird x_i durch iteriertes Anwenden der Swap-Operation probeweise an alle möglichen Stellen in der Variablenordnung geschoben, und es wird die OBDD-Größe bestimmt. Schließlich wird x_i an die Stelle geschoben, die zu minimaler OBDD-Größe führt. Dies wird (für andere Variablen) iteriert, bis keine Verbesserung der OBDD-Größe mehr möglich ist. (Eventuell werden Variablen auch mehrfach verschoben.) Auch der Sifting-Algorithmus wird häufig in lokalen Optima enden. Allerdings wird berichtet, dass der Sifting-Algorithmus sehr schnell ist und gute Ergebnisse liefert.

2.4 Algorithmen auf OBDDs

Auswertung

Eingabe: Ein OBDD G für $f \in B_n$, $a \in \{0, 1\}^n$.

Ausgabe: $f(a)$.

Rechenzeit: $O(n)$. **Speicherplatz:** $O(|G|)$.

Wir durchlaufen das OBDD wie oben beschrieben von der Quelle zu einer Senke und geben die Markierung der erreichten Senke aus. Weil jede Variable höchstens einmal getestet werden darf, ist die Rechenzeit durch $O(n)$ beschränkt.

Erfüllbarkeit

Eingabe: Ein OBDD G für $f \in B_n$.

Ausgabe: 1, falls f ungleich der Nullfunktion ist, anderenfalls 0.

Rechenzeit: $O(|G|)$. **Speicherplatz:** $O(|G|)$.

Durchlaufe das OBDD von der Quelle mit einem depth first search Ansatz, bis eine 1-Senke erreicht wird. Der gefundene Pfad von der Quelle zu der 1-Senke gibt eine Eingabe a mit $f(a) = 1$ an; wenn keine 1-Senke erreicht wird, gibt es auch keine derartige Eingabe.

Wenn G reduziert ist, ist der Erfüllbarkeitstest der Test, ob die Quelle von G ungleich der 0-Senke ist. Dieses ist sogar in konstanter Zeit möglich.

Erfüllbarkeit-Anzahl

Eingabe: Ein OBDD G für $f \in B_n$.

Ausgabe: $|f^{-1}(1)|$.

Rechenzeit: $O(|G|)$. **Speicherplatz:** $O(|G|)$.

Wir wollen für jede Kante und jeden Knoten von G berechnen, für wieviele Eingaben diese Kante oder dieser Knoten durchlaufen wird. Die Quelle wird offensichtlich für alle 2^n Eingaben durchlaufen, und die 1-Senken werden für insgesamt $|f^{-1}(1)|$ Eingaben erreicht. Wir betrachten nun einen inneren Knoten v , der mit x_i markiert ist und der für c Eingaben durchlaufen wird. Wenn der Knoten v für die Eingabe $a = (a_1, \dots, a_n)$ erreicht wird, wird er auch für $a' = (a_1, \dots, a_{i-1}, \bar{a}_i, a_{i+1}, \dots, a_n)$ erreicht, weil die Variable x_i nicht vor v getestet werden darf. Für a wird die a_i -Kante, die v verlässt, durchlaufen, für a' die \bar{a}_i -Kante. Die c Eingaben, für die v erreicht wird, werden also auf die 0-Kante und die 1-Kante gleichmäßig verteilt.

Wenn ein Knoten v die eingehenden Kanten e_1, \dots, e_k hat, müssen wir die Anzahl der Eingaben, für die e_1, \dots, e_k durchlaufen werden, addieren, um die Zahl der Eingaben zu erhalten, für die v erreicht wird. Im folgenden Algorithmus vermeiden wir, auch für die Kanten die Zahl der Eingaben, für die sie erreicht werden, zu speichern.

- Markiere die Quelle mit 2^n und die übrigen Knoten mit 0.
- Durchlaufe G in einer topologischen Ordnung. Wenn ein Knoten v erreicht wird, der mit c markiert ist, addiere zu den Markierungen beider Nachfolger von v den Wert $c/2$.
- Das Ergebnis ist die Markierung der 1-Senke bzw. die Summe der Markierungen der 1-Senken.

Erfüllbarkeit-Alle

Eingabe: Ein OBDD G für $f \in B_n$.

Ausgabe: $f^{-1}(1)$.

Rechenzeit: $O(|G| + n|f^{-1}(1)|)$. **Speicherplatz:** $O(|G|)$.

Wir suchen alle Pfade, die von der Quelle zu einer 1-Senke führen. Wir erhalten alle erfüllenden Belegungen für die Eingabevariablen von f , wenn wir für jeden derartigen Pfad die getesteten Variablen so belegen, dass der Pfad durchlaufen wird, und die nicht getesteten Variablen auf alle möglichen Weisen belegen.

Die Pfade von der Quelle zur 1-Senke können wir mit einem einfachen rekursiven Algorithmus berechnen, der an der Quelle beginnt. Für jeden erreichten inneren Knoten v rufen wir dieselbe Prozedur nacheinander für den 0-Nachfolger von v und den 1-Nachfolger von v rekursiv auf. Wenn wir eine 1-Senke erreichen, rekonstruieren wir den Pfad, auf dem wir zu dieser Senke gelangt sind, belegen die getesteten Variablen so, dass dieser Pfad durchlaufen wird, und die nicht getesteten Variablen auf alle möglichen Weisen. Da die Größe der Ausgabe $n|f^{-1}(1)|$ ist, ist die Rechenzeit $O(|G| + n|f^{-1}(1)|)$ optimal.

Äquivalenztest

Eingabe: OBDDs G_f und G_g für $f, g \in B_n$.

Ausgabe: 1, wenn $f = g$, anderenfalls 0.

Rechenzeit: $O(|G_f| + |G_g|)$. **Speicherplatz:** $O(|G_f| + |G_g|)$.

Da reduzierte OBDDs bei gleicher Variablenordnung bis auf Isomorphie eindeutig sind, genügt es, G_f und G_g zu reduzieren und die reduzierten OBDDs G_f^* und G_g^* auf Isomorphie zu testen. Die Reduktion ist mit dem oben erwähnten Algorithmus in linearer Zeit auf linearem Platz möglich. Der Isomorphietest ist ebenfalls in linearer Zeit möglich, weil OBDDs Graphen mit einer Quelle und markierten Knoten und Kanten sind. Wir durchlaufen G_f^* und G_g^* simultan mit depth first search beginnend bei den Quellen und testen, ob die jeweils erreichten Knoten die gleiche Markierung haben. Wenn f und g gemeinsam in einem reduzierten OBDD dargestellt werden, ist der Äquivalenztest sogar in konstanter Zeit möglich, da es genügt zu testen, ob die Zeiger für f und g auf denselben Knoten zeigen.

Synthese

Eingabe: OBDDs G_f und G_g für $f, g \in B_n$, ein Operator $\otimes \in B_2$.

Ausgabe: Ein OBDD G für $f \otimes g$.

Rechenzeit: $O(|G_f||G_g|)$. **Speicherplatz:** $O(|G_f||G_g|)$.

Wir wollen zunächst zeigen, wie wir für eine Eingabe $a \in \{0, 1\}^n$ mit einem simultanen Durchlauf durch G_f und G_g die Funktionswerte $f(a)$ und $g(a)$ berechnen können. Bei dieser Berechnung soll jedes Eingabebit x_i nur einmal abgefragt werden und dann gleichzeitig für die Rechnung in G_f und G_g benutzt werden. Wir benötigen dafür, dass beide OBDDs dieselbe Variablenordnung, o.B.d.A. x_1, \dots, x_n , haben. Wir beginnen in beiden OBDDs die

Rechnung an den Quellen. Wenn wir in G_f einen Knoten v_f und in G_g einen Knoten v_g erreicht haben, sind die folgenden Situationen möglich.

1. Fall: v_f und v_g sind mit x_i markiert.

Wir gehen in G_f und G_g gemäß dem Wert von a_i zum 0- oder 1-Nachfolger von v_f bzw. v_g .

2. Fall: v_f ist mit x_i markiert, v_g ist mit x_j markiert, und $i > j$.

Die Variable x_j steht in der Variablenordnung vor x_i , daher warten wir in G_f am Knoten v_f , gehen aber in G_g gemäß dem Wert von a_j zum 0- oder 1-Nachfolger von v_g .

3. Fall: v_f ist eine Senke, v_g ist mit x_j markiert.

Wie im 2. Fall warten wir an v_f und gehen in G_g gemäß dem Wert von a_j zum 0- oder 1-Nachfolger.

4. Fall: v_f ist mit x_i markiert, v_g ist mit x_j markiert, und $i < j$.

Analog zum 2. Fall warten wir in G_g am Knoten v_g und gehen in G_f gemäß dem Wert von a_i zum 0- oder 1-Nachfolger von v_f .

5. Fall: v_f ist mit x_i markiert, v_g ist eine Senke.

Wir gehen gemäß dem Wert von a_i zum 0- oder 1-Nachfolger von v_f und warten an v_g .

6. Fall: v_f und v_g sind Senken.

Die Markierungen von v_f und v_g sind die gesuchten Funktionswerte $f(a)$ und $g(a)$. Wir können $(f \otimes g)(a)$ berechnen, indem wir die Markierungen der Senken mit \otimes verknüpfen.

Wir konstruieren nun ein OBDD G , in dem jeder Pfad einer derartigen simultanen Rechnung entspricht. G enthält für jedes Paar von einem Knoten v_f aus G_f und v_g aus G_g einen Knoten (v_f, v_g) . Die neue Quelle ist das Paar der Quellen von G_f und G_g . Die Markierung jedes Knotens (v_f, v_g) und seine Nachfolger erhalten wir durch dieselbe Fallunterscheidung wie oben.

1. Fall: v_f und v_g sind mit x_i markiert.

Der Knoten (v_f, v_g) wird mit x_i markiert, sein 0-Nachfolger ist der Knoten $(v_{f,0}, v_{g,0})$, sein 1-Nachfolger $(v_{f,1}, v_{g,1})$. Dabei bezeichnen $v_{f,c}$ bzw. $v_{g,c}$ den c -Nachfolger von v_f bzw. v_g .

2. Fall: v_f ist mit x_i markiert, v_g ist mit x_j markiert, und $i > j$.

Der Knoten (v_f, v_g) wird mit x_j markiert, seine Nachfolger sind die Knoten $(v_f, v_{g,0})$ bzw. $(v_f, v_{g,1})$, da am Knoten v_f gewartet wird.

3. Fall: v_f ist eine Senke, v_g ist mit x_j markiert.

Der Knoten (v_f, v_g) wird mit x_j markiert, seine Nachfolger sind die Knoten $(v_f, v_{g,0})$ bzw. $(v_f, v_{g,1})$.

- 4. Fall:** v_f ist mit x_i markiert, v_g ist mit x_j markiert, und $i < j$.
 Der Knoten (v_f, v_g) wird mit x_i markiert, seine Nachfolger sind die Knoten $(v_{f,0}, v_g)$ bzw. $(v_{f,1}, v_g)$.
- 5. Fall:** v_f ist mit x_i markiert, v_g ist eine Senke.
 Der Knoten (v_f, v_g) wird mit x_i markiert, seine Nachfolger sind die Knoten $(v_{f,0}, v_g)$ bzw. $(v_{f,1}, v_g)$.
- 6. Fall:** v_f und v_g sind Senken.
 Der Knoten (v_f, v_g) ist eine Senke; die Markierung dieser Senke erhalten wir, indem wir die Markierungen von v_f und v_g mit \otimes verknüpfen.

Das konstruierte OBDD wird auch Produktgraph von G_f und G_g genannt. Es simuliert die simultane Rechnung in G_f und G_g und ist daher ein OBDD für die Funktion $f \otimes g$. Der Produktgraph enthält in der Regel Knoten, die von der Quelle aus nicht erreichbar sind. Im Produktgraphen in Abb. 11 sind nur die fett gezeichneten Knoten und Kanten von der Quelle (die dem Paar der Quellen von G_f und G_g entspricht) erreichbar. Nachdem die nicht erreichbaren Knoten und Kanten entfernt sind, ist der entstandene Graph nicht notwendigerweise ein reduziertes OBDD, auch wenn G_f und G_g reduziert sind. Nachdem in dem Produktgraphen in Abb. 11 die nicht erreichbaren Knoten entfernt worden sind, können noch die Knoten (v_3, v_6) und (v_4, v_7) verschmolzen werden.

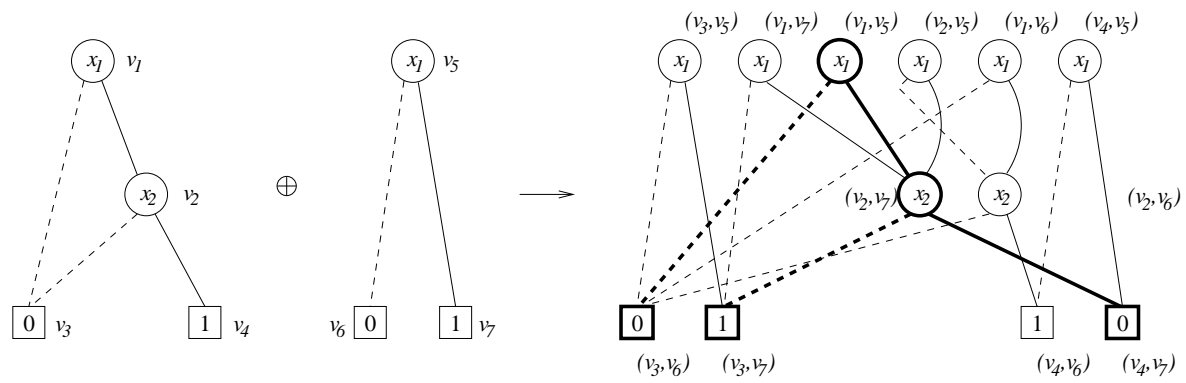


Abbildung 11: Beispiel für die Berechnung eines Produktgraphen

Der Algorithmus von Bryant (1986) vermeidet die Erzeugung von nicht erreichbaren Knoten, indem er die Berechnung des OBDDs für $f \otimes g$ an der Quelle beginnt und nur Nachfolger von bereits konstruierten Knoten erzeugt. Das OBDD wird in einer depth first search Ordnung aufgebaut. Der rekursive Algorithmus wird mit dem Paar der Quellen von G_f und G_g aufgerufen und gibt einen Zeiger auf das erzeugte OBDD zurück. Auf einem Paar (v_f, v_g) werden folgende Schritte ausgeführt:

- Erzeuge einen Knoten für (v_f, v_g) .
- Bestimme mit der Fallunterscheidung von oben die Markierung von (v_f, v_g) .

- Wenn (v_f, v_g) ein innerer Knoten ist, bestimme mit der Fallunterscheidung von oben die Paare, die die Nachfolger von (v_f, v_g) sind. Teste nacheinander für jedes der beiden Paare, ob bereits ein Knoten erzeugt wurde.
 - Wenn ja, setze den Zeiger für den entsprechenden Nachfolger von (v_f, v_g) auf den erzeugten Knoten.
 - Wenn nein, erzeuge den entsprechenden Nachfolger mit einem rekursiven Aufruf, und setze den Zeiger für den Nachfolger von (v_f, v_g) auf den erzeugten Knoten.
- Gib an die aufrufende Prozedur einen Zeiger auf (v_f, v_g) zurück.

Der Test, ob für ein Paar bereits ein Knoten erzeugt wurde, verhindert, dass Teile des konstruierten OBDDs mehrfach erzeugt werden. Wir können dafür ein spezielles Array (die sog. Computed Table) der Größe $|G_f||G_g|$ benutzen, das für jedes Paar von Knoten aus G_f und G_g einen Zeiger auf den eventuell erzeugten Knoten oder ein *nil*-Zeiger, wenn noch kein Knoten erzeugt wurde, enthält. Daher ist die Größenordnung von Rechenzeit und Speicherplatz bei der Berechnung des Produktgraphen wie auch beim Synthesealgorithmus von Bryant $O(|G_f||G_g|)$. Aber auch wenn das erzeugte OBDD viel kleiner ist, ist die Größenordnung der Rechenzeit $|G_f||G_g|$, da das Array initialisiert werden muss. Man kann Speicherplatz sparen, wenn man statt des Arrays eine Hashtabelle benutzt, die kleiner als $|G_f||G_g|$ ist. Nun kann aber nicht mehr garantiert werden, dass der Test, ob für ein Paar bereits ein Knoten erzeugt wurde, in konstanter Zeit möglich ist, und damit auch nicht die Rechenzeit $O(|G_f||G_g|)$.

Eine weitere Verfeinerung des Algorithmus von Bryant testet im 3. Fall, wenn v_f eine Senke ist, (und analog im 5. Fall,) ob die Konstante c , mit der v_f markiert ist, für den Operator \otimes dominierend ist, d.h., ob $c \otimes 0 = c \otimes 1$ gilt. In diesem Fall kann die Rekursion abgebrochen werden, und (v_f, v_g) ist eine Senke mit der Markierung $c \otimes 0$.

Auch beim Synthesealgorithmus von Bryant ist das erzeugte OBDD in der Regel nicht reduziert. Die Reduktion kann aber in die Synthese integriert werden (Brace, Rudell, Bryant (1991)). Sobald für einen Knoten (v_f, v_g) beide Nachfolger berechnet worden sind, kann getestet werden, ob beide Nachfolger identisch sind, also ob die Deletion Rule anwendbar ist. In diesem Fall wird der Knoten (v_f, v_g) wieder entfernt und ein Zeiger auf seinen Nachfolger (v_f^*, v_g^*) an die aufrufende Prozedur zurückgegeben. Zudem wird in der Computed Table an der Stelle (v_f, v_g) gespeichert, dass der Knoten (v_f, v_g) durch (v_f^*, v_g^*) repräsentiert wird. Wenn dieses nicht gespeichert würde, müssten, wenn der Algorithmus später noch einmal für das Paar (v_f, v_g) aufgerufen wird, die Nachfolger von (v_f, v_g) noch einmal berechnet werden. Für die Merging Rule benötigen wir eine weitere Hashtabelle, die sog. Unique-Table. Wenn für einen Knoten (v_f, v_g) beide Nachfolger berechnet worden sind und die Deletion Rule nicht anwendbar ist, wird mit Hilfe der Unique Table getestet, ob es bereits einen Knoten (\hat{v}_f, \hat{v}_g) gibt, der mit derselben Variablen markiert ist und denselben 0- und denselben 1-Nachfolger hat. Wenn dieses der Fall ist, können die beiden Knoten verschmolzen werden, d.h., der Algorithmus gibt einen Zeiger auf (\hat{v}_f, \hat{v}_g) zurück und speichert wie eben in der Computed Table, dass (v_f, v_g) durch (\hat{v}_f, \hat{v}_g) repräsentiert wird. Anderenfalls ist der Knoten

(v_f, v_g) notwendig, und wir speichern in der Unique Table für den erzeugten Knoten das Tripel aus der Markierung des Knotens (v_f, v_g) , seinem 0-Nachfolger und seinem 1-Nachfolger.

Wenn wir die Reduktion in die Synthese integrieren, ist es wichtig, dass das synthetisierte OBDD in einer depth first search Ordnung aufgebaut wird; bei der Synthese ohne Reduktion ist z.B. auch eine breadth first search Ordnung möglich. Der Vorteil der depth first search Ordnung liegt darin, dass es im erzeugten OBDD nur einen Pfad gibt, auf dem Knoten liegen, für die noch nicht beide Nachfolger berechnet worden sind. Es werden daher gleichzeitig höchstens n Knoten gespeichert, die eventuell später aufgrund einer Reduktionsregel wieder gelöscht werden können. Die Anzahl der Knoten, die insgesamt gespeichert werden müssen, ist daher durch $|G^*| + n$ beschränkt, wobei G^* das reduzierte OBDD für $f \otimes g$ ist. Bei einem Aufbau mit breadth first search ist es dagegen möglich, dass viel mehr Knoten gespeichert werden müssen, für die noch nicht beide Nachfolger berechnet worden sind und die damit zu einem späteren Zeitpunkt eventuell wieder gelöscht werden.

***m*-äre Synthese**

Die Synthese kann auch auf *m*-äre Operatoren erweitert werden:

Eingabe: OBDDs G_1, \dots, G_m für die Funktionen $f_1, \dots, f_m \in B_n$, eine Funktion $h \in B_m$.

Ausgabe: Ein OBDD für $h(f_1, \dots, f_m)$.

Rechenzeit: $O(|G_1| |G_2| \dots |G_m|)$. **Speicherplatz:** $O(|G_1| |G_2| \dots |G_m|)$.

Der simultane Durchlauf durch m OBDDs ist genauso möglich wie der simultane Durchlauf durch zwei OBDDs. Dieser Durchlauf kann wiederum durch einen Produktgraphen von G_1, \dots, G_m simuliert werden. Dieser Produktgraph enthält für jedes m -Tupel (v_1, \dots, v_m) von Knoten aus G_1, \dots, G_m einen Knoten. Sei x_i die Variable, die von den Variablen, mit denen v_1, \dots, v_m markiert sind, als erste in der Variablenordnung steht. Dann wird am Knoten (v_1, \dots, v_m) die Variable x_i getestet. Die Nachfolger von (v_1, \dots, v_m) sind (v_1^0, \dots, v_m^0) bzw. (v_1^1, \dots, v_m^1) , wobei v_j^c der c -Nachfolger von v_j im Graphen G_j ist, falls an v_j die Variable x_i getestet wird. Wenn an v_j nicht x_i getestet wird, müssen wir beim simultanen Durchlauf am Knoten v_j warten, daher setzen wir v_j^c auf v_j . Wenn alle Knoten v_1, \dots, v_m Senken sind, die mit c_1, \dots, c_m markiert sind, ist auch (v_1, \dots, v_m) eine Senke, die mit $h(c_1, \dots, c_m)$ markiert ist.

Wie bei der binären Synthese kann mit dem DFS-Ansatz von Bryant die Konstruktion nicht erreichbarer Knoten verhindert werden, und es kann auch die Reduktion integriert werden.

Ersetzung durch Konstanten

Eingabe: Ein OBDD G für $f \in B_n$, eine Variable x_i , eine Konstante c .

Ausgabe: Ein OBDD G' für $f|_{x_i=c}$.

Rechenzeit: $O(|G|)$. **Speicherplatz:** $O(|G|)$.

Wenn die Quelle mit x_i markiert ist, definieren wir den c -Nachfolger der Quelle als neue Quelle. Anderenfalls durchlaufen wir G mit depth first search und testen für jeden Knoten, ob einer seiner Nachfolger w mit x_i markiert ist. In diesem Fall wird der Zeiger auf w auf den

c -Nachfolger von w umgesetzt. Anschließend können die mit x_i markierten Knoten gelöscht werden. Nun sind möglicherweise nicht mehr alle Knoten des OBDDs von der Quelle aus erreichbar, z.B. ist im OBDD in Abb. 12 nach der Ersetzung von x_2 durch 0 die 1-Senke nicht mehr erreichbar. Die nicht erreichbaren Knoten werden ebenfalls gelöscht.

Auch wenn das OBDD G reduziert ist, ist das neue OBDD G' nicht notwendigerweise reduziert (s. Abb. 12). Es genügt, die Reduktion bottom-up auf den Schichten des OBDDs durchzuführen, die oberhalb von der Schicht der mit x_i markierten Knoten liegen. Unterhalb dieser Schicht wurden nur nicht erreichbare Knoten entfernt, aber keine Zeiger auf Nachfolger verändert. Daher gibt es dort keine Knoten, auf die die Deletion Rule oder die Merging Rule anwendbar ist.

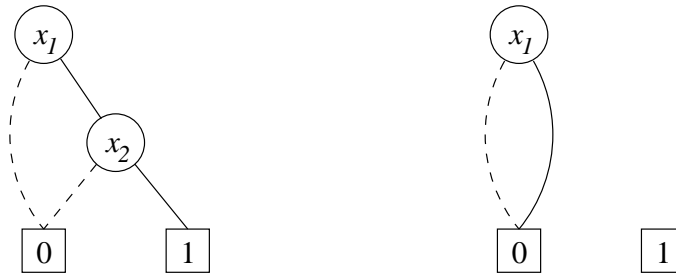


Abbildung 12: OBDDs für $f(x_1, x_2) = x_1x_2$ und $f|_{x_2=0}$

Ersetzung durch Funktionen

Eingabe: OBDDs G_f und G_g für die Funktionen $f, g \in B_n$, eine Variable x_i .

Ausgabe: Ein OBDD G für $f|_{x_i=g}$.

Rechenzeit: $O(|G_f|^2|G_g|)$. **Speicherplatz:** $O(|G_f|^2|G_g|)$.

Die Funktion $f|_{x_i=g}$ kann mit der Shannon-Zerlegung geschrieben werden als

$$f|_{x_i=g} = \bar{g}f|_{x_i=0} \vee gf|_{x_i=1} = ite(g, f|_{x_i=1}, f|_{x_i=0}).$$

Dabei ist der ternäre Operator *ite* (if-then-else) definiert durch

$$ite(a, b, c) := \bar{a}c \vee ab.$$

Wir wenden daher den Synthesalgorithmus auf den ternären Operator *ite* und die OBDDs für die Funktionen $g, f|_{x_i=1}$ und $f|_{x_i=0}$ an und erhalten in der Zeit $O(|G_f|^2|G_g|)$ ein OBDD für $f|_{x_i=g}$. Die OBDDs für $f|_{x_i=1}$ und $f|_{x_i=0}$ müssen nicht explizit berechnet werden, wenn wir den Synthesalgorithmus so modifizieren, dass an mit x_i markierten Knoten direkt zum 1- bzw. 0-Nachfolger weitergegangen wird.

Quantifizierung

Eingabe: Ein OBDD G für $f \in B_n$, eine Variable x_i .

Ausgabe: Ein OBDD für $(\exists x_i : f) = f|_{x_i=0} \vee f|_{x_i=1}$ bzw. $(\forall x_i : f) = f|_{x_i=0} \wedge f|_{x_i=1}$.

Rechenzeit: $O(|G|^2)$. **Speicherplatz:** $O(|G|^2)$.

Wir wenden den Synthesalgorithmus auf OBDDs für $f|_{x_i=0}$ und $f|_{x_i=1}$ und \vee bzw. \wedge an und erhalten in der Zeit $O(|G|^2)$ ein OBDD für die gewünschte Funktion. Wie bei der Ersetzung durch Funktionen können wir vermeiden, dass die OBDDs für $f|_{x_i=0}$ und $f|_{x_i=1}$ berechnet und gespeichert werden, indem wir die Konstantsetzung von x_i in die Synthese integrieren.

Redundanztest

Eingabe: Ein OBDD G für $f \in B_n$, eine Variable x_i .

Ausgabe: 1, falls $f|_{x_i=0} = f|_{x_i=1}$, anderenfalls 0.

Rechenzeit: $O(|G|)$. **Speicherplatz:** $O(|G|)$.

Ein reduziertes OBDD für f enthält genau dann mit x_i markierte Knoten, wenn f essentiell von x_i abhängt, also $f|_{x_i=0} \neq f|_{x_i=1}$ gilt: Wenn ein OBDD für f keinen mit x_i markierten Knoten enthält, wird für $x_i = 0$ und $x_i = 1$ derselbe Funktionswert berechnet, die Funktion hängt also nicht essentiell von x_i ab. Wenn f nicht essentiell von x_i abhängt, gilt dieses auch für alle Subfunktionen von f . Daher ist die Menge S_i leer, und nach Satz 2.1.1 enthält das reduzierte OBDD für f keinen mit x_i markierten Knoten.

Es genügt also, mit einem Graphdurchlauf nach mit x_i markierten Knoten zu suchen, nachdem G reduziert worden ist. Dazu genügen lineare Rechenzeit und linearer Speicherplatz.

Worst-case Beispiele

Wir wollen diskutieren, wie groß die erzeugten OBDDs bei den Operationen, bei denen OBDDs ausgegeben werden, sein können. Bei der Berechnung von OBDDs aus Schaltkreisen wird der Synthesalgorithmus sehr oft nacheinander auf die erzeugten OBDDs angewandt. Daher ist die Frage wichtig, wie stark sich OBDDs bei der Synthese vergrößern können. Die Größe des Produktgraphen von G_f und G_g ist durch $|G_f||G_g|$ beschränkt, aber wir haben an einem Beispiel gesehen, dass die meisten Knoten des Produktgraphen von der Quelle aus gar nicht erreichbar sind und auf den erreichbaren Knoten noch Reduktionen möglich sind. Wir wollen an einem Beispiel zeigen, dass ein reduziertes OBDD für $f \otimes g$ wirklich die Größe $\Theta(|G_f||G_g|)$ haben kann.

Die Operationen Ersetzung durch Funktionen und Quantifizierung benutzen zwar den Synthesalgorithmus, aber nur für spezielle Eingaben, nämlich für die Funktionen $f|_{x_i=0}$ und zugleich $f|_{x_i=1}$. Wir werden auch für diese Operationen zeigen, dass die Größe der Ausgabe bei Ersetzung durch Funktionen $\Theta(|G_f|^2|G_g|)$ und bei Quantifizierung $\Theta(|G|^2)$ sein kann. Bei Ersetzung durch Konstanten ist dagegen klar, dass sich die Anzahl der Knoten des OBDDs nicht vergrößern kann.

Für alle Beispiele benutzen wir die Funktion INDEX. Die Funktion $\text{INDEX}_n \in B_{n+k}$ ist auf $n = 2^k$ Variablen x_{n-1}, \dots, x_0 und auf k Variablen a_{k-1}, \dots, a_0 definiert. Es ist

$$\text{INDEX}_n(x, a) = x_{|a|},$$

wobei $|a|$ den Wert von a als Binärzahl interpretiert bezeichnet. Das reduzierte OBDD für INDEX_n und die Variablenordnung $a_{k-1}, \dots, a_0, x_{n-1}, \dots, x_0$ hat $2n - 1$ innere Knoten.

Dieses OBDD besteht aus einem vollständigen binären Baum, in dem a_{k-1}, \dots, a_0 getestet werden. Der Baum hat $n - 1$ innere Knoten und n Blätter, wobei für jeden Wert von $|a|$ eines der Blätter erreicht wird. An jedem Blatt wird dann die entsprechende x -Variable getestet.

Für das worst-case Beispiel für die Synthese benutzen wir die Variablenordnung $a_{k-1}, \dots, a_0, b_{k-1}, \dots, b_0, x_{n-1}, \dots, x_0$ und berechnen ein OBDD für

$$h_n = \text{INDEX}_n(x, a) \vee \text{INDEX}_n(x, b)$$

aus OBDDs für $\text{INDEX}_n(x, a)$ und $\text{INDEX}_n(x, b)$. Diese OBDDs haben lineare Größe, das OBDD für h_n hat quadratische Größe, denn in diesem OBDD müssen die Werte von a und b gespeichert werden, um die beiden richtigen x -Variablen auszuwählen. Dazu sind $\Omega(n^2)$ Knoten nötig. Dieses können wir auch formaler mit dem Satz 2.1.1 beweisen. Für jede Konstantsetzung von a und b , für die $|a| < |b|$ gilt, entsteht eine andere Subfunktion von h_n , weil mindestens eine andere x -Variable ausgewählt wird. Jede dieser Subfunktionen hängt von zwei x -Variablen essentiell ab. Da es $\Omega(n^2)$ Konstantsetzungen für a und b mit $|a| < |b|$ gibt, gibt es im reduzierten OBDD für h_n auch $\Omega(n^2)$ Knoten, die mit einer x -Variablen markiert sind.

Für die Quantifizierung betrachten wir die Funktion

$$f_n = \bar{s}\text{INDEX}_n(x, a) \vee s\text{INDEX}_n(x, b)$$

mit der Variablenordnung $s, a_{k-1}, \dots, a_0, b_{k-1}, \dots, b_0, x_{n-1}, \dots, x_0$. Die Funktion kann von einem OBDD mit höchstens $4n - 1$ Knoten dargestellt werden. Zuerst wird die Variable s getestet und dann gemäß dem Wert von s in ein OBDD für $\text{INDEX}_n(x, a)$ oder für $\text{INDEX}_n(x, b)$ verzweigt. Ein OBDD für $\exists s : f_n$ hat quadratische Größe, denn $\exists s : f_n$ ist gleich der Funktion h_n aus dem letzten Absatz.

Für Ersetzung durch Funktionen benutzen wir die Funktion f_n aus dem letzten Absatz und die Funktion $g_n = \text{INDEX}_n(x, c)$ für die Variablenordnung

$$s, a_{k-1}, \dots, a_0, b_{k-1}, \dots, b_0, c_{k-1}, \dots, c_0, x_{n-1}, \dots, x_0$$

und berechnen ein OBDD für $f_{n|s=g_n}$. Es ist

$$f_{n|s=g_n} = \overline{\text{INDEX}_n(x, c)} \text{INDEX}_n(x, a) \vee \text{INDEX}_n(x, c) \text{INDEX}_n(x, b).$$

Wir erhalten hier für jede Konstantsetzung von a, b und c mit $|a| < |b| < |c|$ eine andere Subfunktion, die von drei der x -Variablen essentiell abhängt. Daher ist die Größe eines OBDDs für $f_{n|s=g_n}$ kubisch in n .

2.5 Schlussbemerkungen

OBDDs erlauben die kompakte Darstellung von vielen interessanten booleschen Funktionen, und es gibt effiziente Algorithmen für die Manipulation von Funktionen, die durch OBDDs dargestellt sind. Hierbei sollten wir allerdings nicht vergessen, dass z.B. bei der

Umformung eines Schaltkreises in ein OBDD solche Operationen, insbesondere die Synthese, häufig hintereinander ausgeführt werden, so dass exponentielle Vergrößerungen auftreten können. Ebenso gibt es auch Funktionen, die nur exponentielle OBDDs haben. OBDDs sind also nicht *die* Datenstruktur für boolesche Funktionen, mit der sich alle Probleme lösen lassen, sondern nur ein heuristischer Ansatz, mit dem man manche Probleme lösen kann, die mit konventionellen Ansätzen (z.B. Darstellung von Funktionen durch Wertetabellen oder Schaltkreise) nicht praktisch lösbar sind. Manchmal können Probleme auch nur dadurch gelöst werden, dass OBDDs zusammen mit weiteren „Tricks“ eingesetzt werden. Ein weiterer Lösungsansatz besteht in der Entwicklung von Erweiterungen von OBDDs, die die kompakte Darstellung von einer größeren Klasse von Funktionen als OBDDs erlauben.

3 Methoden zum Beweis unterer Schranken für BDDs

3.1 Untere Schranken für BDDs ohne weitere Einschränkungen

Wir starten mit dem Beweis unterer Schranken für nicht weiter eingeschränkte BDDs. Die bislang beste untere Schranke hat nur die Größenordnung $\Omega(n^2 / \log^2 n)$ und stammt aus dem Jahr 1966. Dies deutet an, dass das Problem des Beweises unterer Schranken schwierig zu sein scheint. Warum ist dies so? Zunächst einmal bemerken wir, dass Beweise oberer Schranken häufig einfach sind, weil es genügt, ein BDD mit der gewünschten Größe *anzugeben*. Bei einer unteren Schranke müssen wir zeigen, dass *alle* BDDs kleiner Größe die gewünschte Funktion nicht berechnen. Argumentationen über alle BDDs sind ähnlich wie Argumentationen über alle Algorithmen aber eher schwierig.

Wir wollen das Problem des Beweises exponentieller unterer Schranken noch weiter einordnen. Nach den Simulationen in Abschnitt 1 folgt aus einer superpolynomiellen unteren Schranke für die BDD-Größe für eine Funktion in P, dass $L \neq P$ ist, wobei L die Menge aller Sprachen mit logarithmisch speicherplatzbeschränkten Algorithmen ist. Die Separation von L und P (oder auch nur von L und NP) ist ein Problem, das von der Schwierigkeit her mit der $P \neq NP$ -Frage vergleichbar zu sein scheint.

Wir hatten in Abschnitt 1 auch angemerkt, dass fast alle boolesche Funktionen exponentielle BDD-Größe haben. Dies kann man mit Zählargumenten leicht zeigen, wie wir gleich sehen werden. Auf diese Weise bekommt man nur Existenzaussagen, d.h. Aussagen der Form, dass es viele Funktionen mit exponentieller BDD-Größe gibt, allerdings erfahren wir nichts darüber, welche Funktionen dies sind. Vielleicht sind dies nur abstruse oder praktisch nicht relevante Funktionen. Zur Veranschaulichung der Unterschiede zwischen Existenzaussagen und Aussagen über konkrete Funktionen betrachten wir ein Problem aus der Mathematik: Es ist einfach zu zeigen, dass $\mathbb{Q} \neq \mathbb{R}$ ist: \mathbb{Q} enthält abzählbar viele Zahlen, \mathbb{R} überabzählbar viele. Durch diesen Beweis erfahren wir nichts darüber, wie irrationale Zahlen aussehen. Der bekannte Beweis für $\sqrt{2} \in \mathbb{R} \setminus \mathbb{Q}$ ist dagegen viel instruktiver, da er auch zeigt, dass es „natürliche“ Beispiele von irrationalen Zahlen gibt. Aus denselben Gründen sucht die Komplexitätstheorie nach Beweisen von unteren Schranken für *explizit definierte Funktionen*. Aus solchen Beweisen erfahren wir auch etwas über die schwierigen Funktionen. Der Begriff explizit definiert ist natürlich etwas unklar. Manchmal bezeichnet man damit Funktionen, die man „angeben“ kann, eine andere Interpretation ist, dass hiermit Probleme aus NP gemeint sind, weil für derartige Probleme keine Zählargumente bekannt sind.

In diesem Abschnitt wollen wir beide Beweistypen vorführen, ein Zählargument und die Methode von Nečiporuk (1966) zum Beweis unterer Schranken für explizit definierte Funktionen.

Lemma 3.1.1 Die Anzahl der Funktionen $f : \{0, 1\}^n \rightarrow \{0, 1\}$, die BDDs mit Größe höchstens s haben, beträgt höchstens $n^s (s!)^2$.

Beweis: Wir zählen zunächst die syntaktisch verschiedenen BDDs der Größe s . Jedes BDD für nicht-konstante Funktionen kann durch eine Liste von $s - 2$ Knoten beschrieben werden. Für jeden Knoten haben wir n Möglichkeiten für die zu testende Variable. Für den i -ten Knoten gibt es $s - i$ Möglichkeiten, jeden der beiden Nachfolger zu wählen. Also beträgt die Anzahl der syntaktisch verschiedenen BDDs mit s Knoten höchstens $n^{s-2}((s-1)!)^2$. Da jedes solche BDD höchstens s Funktionen darstellt, folgt die behauptete Schranke. \square

Satz 3.1.2 Die BDD-Größe von allen Funktionen aus B_n außer einem exponentiell kleinem Anteil beträgt $\Omega(2^n/n)$.

Beweis: Wir bemerken zunächst, dass B_n genau 2^{2^n} Funktionen enthält. Davon betrachten wir die $2^{2^n - 2^{n/2}}$ Funktionen mit den kleinsten BDDs und zeigen, dass mindestens eine dieser Funktionen die BDD-Größe $\Omega(2^n/n)$ hat. Da $2^{2^n - 2^{n/2}} = 2^{2^n} \cdot 2^{-2^{n/2}}$ ist, betrachten wir nur einen exponentiell kleinen Anteil und alle anderen Funktionen haben BDDs, die mindestens genauso groß sind.

Annahme: Alle betrachteten Funktionen haben BDDs der Größe $s = o(2^n/n)$.

Die Anzahl der Funktionen mit BDD-Größe höchstens s beträgt höchstens $n^s(s!)^2$. Also folgt $n^s(s!)^2 \geq 2^{2^n - 2^{n/2}}$. Durch Logarithmieren und Anwenden der Stirling-Formel (aus der $\log(s!) = O(s \log s)$ folgt) erhalten wir

$$s \log n + O(s \log s) \geq 2^n - 2^{n/2}.$$

Unter der Annahme $s = o(2^n/n)$ ist die linke Seite von der Größenordnung $o(2^n)$. Widerspruch. \square

Wir kommen nun zur Technik von Nečiporuk. Unter einer S -Subfunktion von f verstehen wir eine Funktion, die aus f durch Konstantsetzen der Variablen *außerhalb* von S entsteht.

Satz 3.1.3 Sei $f : \{0, 1\}^n \rightarrow \{0, 1\}$ eine Funktion, die von allen n Variablen essentiell abhängt, und seien S_1, \dots, S_k disjunkte Teilmengen der Variablenmenge. Dann gilt

$$\text{BDD}_f = \Omega\left(\sum_{i=1}^k \frac{\log s_i}{\log \log s_i}\right),$$

wobei s_i die Anzahl der S_i -Subfunktionen von f bezeichnet.

Beweis: Sei G ein BDD für f mit minimaler Größe. Sei t_i die Anzahl der Knoten in G , die mit Variablen aus S_i markiert sind. Dann ist $\text{BDD}_f \geq t_1 + \dots + t_k + 2$, und es genügt $t_i = \Omega((\log s_i)/(\log \log s_i))$ zu zeigen. Da f von allen Variablen essentiell abhängt, folgt $t_i \geq |S_i|$. Für jede S_i -Subfunktion von f können wir ein BDD mit höchstens t_i Knoten aus f berechnen, da alle Variablen außerhalb von S_i konstantgesetzt werden. Aus dem obigen Lemma folgt dann

$$s_i \leq |S_i|^{t_i} (t_i!)^2 \leq t_i^{t_i} (t_i!)^2 \leq (t_i)^{3t_i}.$$

Hieraus folgt die behauptete untere Schranke für t_i . \square

Wir müssen jetzt nur noch die Methode für konkrete Funktionen anwenden. Dazu betrachten wir die so genannte indirekte Adressierungsfunktion ISA. Sei $n = 2^l$. Die Eingabe für ISA_n besteht aus den Variablen $y_{l-1}, \dots, y_0, x_{n-1}, \dots, x_0$. Die Folge der x -Variablen wird in $\lfloor n/l \rfloor$ Blöcke $B_0, \dots, B_{\lfloor n/l \rfloor - 1}$ der Länge l aufgeteilt. y_{l-1}, \dots, y_0 wird als Binärzahl s aus $\{0, \dots, n-1\}$ aufgefasst. Falls $s \geq \lfloor n/l \rfloor$ ist die Ausgabe 0. Anderenfalls wird der Block B_s wiederum als Binärzahl t aus $\{0, \dots, n-1\}$ aufgefasst. Ausgabe ist dann x_t .

Für die Mengen S_i wählen wir die Blöcke aus der Definition von ISA. Die Anzahl der S_i -Subfunktionen beträgt dann mindestens 2^{n-l} : Wir wählen die y -Variablen so, dass der betrachtete Block adressiert wird. Für jede Belegung der x -Variablen außerhalb des betrachteten Blocks entsteht eine andere Subfunktion, da durch geeignete Belegung der Variablen in S_i jede solche x -Variable ausgegeben werden kann. Also haben alle s_i den Wert 2^{n-l} . Weiterhin ist $k = \lfloor n/l \rfloor$. Also erhalten wir mit der Nečiporuk-Methode die untere Schranke $\Omega(n^2 / \log^2 n)$. Man kann zeigen, dass mit der Nečiporuk-Methode keine größeren Schranken gezeigt werden können. Wir haben also die folgende Aussage bewiesen.

Satz 3.1.4 $BDD_{ISA_n} = \Omega(n^2 / \log^2 n)$.

3.2 Beweis unterer Schranken mit Kommunikationskomplexität

Die Kommunikationskomplexität ist ein Teilbereich der theoretischen Informatik, die Techniken zum Beweis unterer Schranken für explizit definierte Funktionen bereitstellt. Man betrachtet dabei das folgende Modell für die Auswertung einer Funktion $f : X \times Y \rightarrow \{0, 1\}$, deren Eingabe aus den zwei Teilen $x \in X$ und $y \in Y$ besteht (wobei hier immer $X = \{0, 1\}^n$, $Y = \{0, 1\}^m$ ist). Gegeben sind zwei Rechner oder Personen, die üblicherweise mit Alice und Bob bezeichnet werden. Alice erhält x und Bob erhält y . Die Rechner sollen gemeinsam $f(x, y)$ berechnen, wobei sie Informationen gemäß eines vorab vereinbarten Protokolls austauschen dürfen. Am Ende der Rechnung muss Bob den Funktionswert kennen. Bei dem Modell interessiert man sich nur für die Anzahl ausgetauschter Bits, d.h., Rechenzeit und Speicherplatz von Alice und Bob werden in der Regel vernachlässigt.

Definition 3.2.5 Die Kommunikationskomplexität von $f : X \times Y \rightarrow \{0, 1\}$ für ein gegebenes Protokoll P ist die maximale Anzahl von Bits, die das Protokoll für die Auswertung von f benötigt, wobei das Maximum über alle Eingaben gebildet wird. Die Kommunikationskomplexität von f ist die minimale Kommunikationskomplexität von f für ein Protokoll P , wobei das Minimum über alle Protokolle gebildet wird.

Im einfachsten Fall sendet Alice ihre Eingabe x an Bob. Dann kann Bob den Funktionswert $f(x, y)$ ausrechnen. Die Anzahl der gesendeten Bits ist gleich $\lceil \log |X| \rceil$. Wie man sofort sieht, genügt diese Anzahl für die Berechnung jeder Funktion f . Die Frage ist nun, für welche Funktionen weniger Bits genügen und wie man untere Schranken für die Anzahl der auszutauschenden Bits beweisen kann. In einem zweiten Schritt werden wir dann zeigen, wie man derartige untere Schranken für den Beweis von unteren Schranken für die OBDD-Größe nutzen kann.

Bevor wir Techniken zum Beweis unterer Schranken diskutieren, behandeln wir noch zwei Beispiele. Beim ersten Beispiel werden x und y als Binärzahlen mit den Werten $|x|$ und $|y|$ interpretiert, und es ist 1 auszugeben, wenn $|x| - |y| \equiv 0 \pmod{5}$ ist. Dann genügt es, dass Alice nicht die ganze Zahl x sendet, sondern nur $|x| \pmod{5}$. Dazu genügen 3 Bits. Damit kann Bob dann testen, ob $|x| - |y| \equiv 0 \pmod{5}$ gilt.

Als zweites Beispiel betrachten wir den Gleichheitstest von x und y mit $x, y \in \{0, 1\}^n$. D.h., $f(x, y) = 1$ genau dann, wenn $x = y$. Hier gibt es keine nahe liegende Idee, wie man mit weniger als n Bits an Kommunikation die Funktion f auswerten kann. Allerdings ist zunächst nicht klar, ob Alice nicht an Stelle von x irgendeine Funktion auf x berechnen kann und den Funktionswert, der vielleicht aus weniger als n Bits besteht, an Bob schicken kann. Nehmen wir also an, dass dies möglich ist, d.h., für jede Eingabe x sendet Alice einen Wert $g(x)$, der aus weniger als n Bits besteht. Dann aber gibt es zwei verschiedene Eingaben x und x' , für die $g(x) = g(x')$ gilt. Falls nun Bob die Eingabe x und die Kommunikation $g(x)$ erhält, kann er nicht feststellen, ob Alice auch x oder stattdessen x' als Eingabe hat. Also kann er die Funktion f nicht berechnen. Widerspruch. Wir beachten allerdings, dass wir die Möglichkeit, dass auch Bob Information an Alice schicken kann, bei dieser Art der Beweisführung außer Acht gelassen haben.

Im Folgenden wollen wir diese Art von Argumenten zum Beweis von unteren Schranken weiter formalisieren. Dazu betrachten wir zunächst eine andere Beschreibungsform von Kommunikationsprotokollen, nämlich die so genannten Protokollbäume.

Definition 3.2.6 Sei $f : X \times Y \rightarrow \{0, 1\}$. Ein Protokollbaum für f ist ein binärer Baum, bei dem jeder innere Knoten v entweder Alice oder Bob zugeordnet ist. Falls v Alice zugeordnet ist, gibt es eine Funktion $g_v : X \rightarrow \{0, 1\}$, wobei $g_v(x) \in \{0, 1\}$ angibt, ob die Berechnung am linken oder rechten Kind fortzusetzen ist. Falls v Bob zugeordnet ist, gibt es eine Funktion $g_v : Y \rightarrow \{0, 1\}$ mit analoger Bedeutung. Ähnlich wie in einem BDD gibt es für jede Eingabe einen Pfad von der Wurzel zu einem Blatt und die Markierung des Blatts gibt den Funktionswert an.

Man sieht leicht, dass ein Protokollbaum für f ein Kommunikationsprotokoll für f beschreibt und dass es für jedes Kommunikationsprotokoll einen Protokollbaum gibt. An jedem Knoten v wertet der zugehörige Spieler die Funktion g_v aus und sendet das Ergebnis an den anderen Spieler. Es ist dann klar, an welchem Knoten weiterzurechnen ist und welcher Spieler das nächste Bit zu senden hat. Die Länge des Pfades für jede Eingabe x gibt dann die Anzahl der ausgetauschten Bits an. Wir interessieren uns für den worst-case, d.h., die Kommunikationskomplexität der Funktion f ist die Länge des längsten Pfades von der Wurzel zu einem Blatt.

Für die weiteren Argumente benötigen wir noch zwei weitere Definitionen.

Definition 3.2.7 Die Kommunikationsmatrix M_f von $f : X \times Y \rightarrow \{0, 1\}$ ist eine $|X| \times |Y|$ -Matrix, deren Zeilen mit den Werten aus X und deren Spalten mit den Werten aus Y markiert sind. Der Eintrag an der Position (x, y) ist dann gleich $f(x, y)$.

Beispiel: Wenn $f(x, y)$ die Gleichheitsfunktion ist, ist die zugehörige Kommunikationsmatrix die $2^n \times 2^n$ -Einheitsmatrix. Die Kommunikationsmatrix für die Funktion $\text{DQF}_3(x_1, x_2, x_3, y_1, y_2, y_3) = x_1y_1 \vee x_2y_2 \vee x_3y_3$ ist in Abbildung 13 gezeigt.

		x_1, x_2, x_3							
		000	001	010	011	100	101	110	111
y_1, y_2, y_3	000	0	0	0	0	0	0	0	0
	001	0	1	0	1	0	1	0	1
	010	0	0	1	1	0	0	1	1
	011	0	1	1	1	0	1	1	1
	100	0	0	0	0	1	1	1	1
	101	0	1	0	1	1	1	1	1
	110	0	0	1	1	1	1	1	1
	111	0	1	1	1	1	1	1	1

Abbildung 13: Die Kommunikationsmatrix und ein monochromatisches Rechteck für die Funktion DQF_3

Definition 3.2.8 Ein (kombinatorisches) Rechteck R in M_f ist eine Menge von Einträgen mit Indizes in $X' \times Y'$, wobei $X' \subseteq X$ und $Y' \subseteq Y$. Ein Rechteck R heißt monochromatisch, wenn alle Einträge von M_f in R gleich sind.

Wir beachten, dass ein kombinatorisches Rechteck R (im Gegensatz zu einem geometrischen Rechteck) nicht aus zusammenhängenden Einträgen von M_f bestehen muss, die einzige Bedingung ist, dass aus $(x, y) \in R$ und $(x', y') \in R$ folgt, dass auch $(x, y') \in R$ und $(x', y) \in R$ gilt. Ein Beispiel für ein kombinatorisches Rechteck mit $X' = \{010, 011, 110, 111\}$ und $Y' = \{010, 011, 110, 111\}$ findet sich in Abbildung 13. Für den Begriff monochromatisch werden die Funktionswerte von f mit Farben assoziiert, also darf ein Rechteck nur eine Farbe enthalten.

Wir untersuchen nun den Zusammenhang zwischen Protokollbäumen und Rechtecken.

Lemma 3.2.9 Die Menge der Eingaben (x, y) , für die ein Knoten v des Protokollbaumes erreicht wird, ist ein Rechteck von M_f . Die Menge der Eingaben (x, y) , für die ein Blatt erreicht wird, ist ein monochromatisches Rechteck.

Beweis: Wir beweisen die erste Aussage mit Induktion über die Tiefe des Knotens v . Für den Induktionsanfang betrachten wir die Wurzel des Baumes, die für alle Eingaben aus $X \times Y$ erreicht wird, was offensichtlich auch ein Rechteck ist. Für den Induktionsschritt betrachten wir einen Knoten v . Nach Induktionsannahme wird der Vorgänger v' für genau die Eingaben

aus einem Rechteck $X' \times Y'$ erreicht. Wenn v' o.B.d.A. Alice zugeordnet ist und v o.B.d.A. der 0-Nachfolger von v' ist, wird v für alle Eingaben aus $(x, y) \in X' \times Y'$ erreicht, für die zusätzlich $g_{v'}(x) = 0$ ist. Dies ist eine Menge der Form $X'' \times Y'$, wobei $X'' = g^{-1}(0) \cap X'$ ist. Damit wird auch v genau für die Eingaben aus einem Rechteck erreicht.

Wenn ein Blatt für verschiedene Eingaben erreicht wird, müssen die Funktionswerte für diese Eingaben übereinstimmen, also ist das zu dem Blatt gehörende Rechteck sogar monochromatisch. \square

Da für jede Eingabe genau ein Blatt erreicht wird, bilden die zu den Blättern gehörenden Rechtecke eine Partition von M_f . Mit anderen Worten: Jeder Protokollbaum induziert eine Partition von M_f in monochromatische Rechtecke. Wenn nun jede derartige Partition aus vielen Rechtecken besteht, muss der Protokollbaum viele Blätter haben und wir erhalten darüber untere Schranken für die Tiefe und damit für die Kommunikationskomplexität.

Beispiel: Wir haben oben die Gleichheitsfunktion betrachtet. Die zugehörige Kommunikationsmatrix ist die Einheitsmatrix. Man sieht leicht ein, dass zwei 1-Einträge der Einheitsmatrix nicht in demselben Rechteck liegen können. D.h., jede Partition der $2^n \times 2^n$ -Einheitsmatrix in monochromatische Rechtecke enthält mindestens 2^n Rechtecke. Damit enthält der zugehörige Protokollbaum mindestens 2^n Blätter. Da Protokollbäume binär sind, hat der zugehörige Protokollbaum mindestens die Tiefe n . D.h., unsere obige Vermutung, dass für die Berechnung der Gleichheitsfunktion mindestens n Bits an Kommunikation nötig sind, ist damit bewiesen.

Im Beispiel der Gleichheitsfunktion konnte man die untere Schranke für die Anzahl der monochromatischen Rechtecke in einer Zerlegung direkt sehen. Eine andere Methode benutzt den Rang der Kommunikationsmatrix.

Satz 3.2.10 Die Anzahl der 1-Rechtecke einer Zerlegung von M_f ist mindestens gleich dem Rang von M_f , wobei wir den Rang über den reellen Zahlen betrachten.

Beweis: Wir betrachten einen beliebigen Protokollbaum für f . Für jedes 1-Blatt l definieren wir M_l als die $2^n \times 2^n$ -Matrix, die genau an den Positionen eine 1 hat, für die l erreicht wird. Da l genau für Eingaben aus einem Rechteck erreicht wird, ist der Rang von M_l gleich 1. Weiterhin gilt offensichtlich $M_f = \sum_{l \text{ 1-Blatt}} M_l$. Da der Rang subadditiv ist, folgt,

$$\text{rang}(M_f) \leq \sum_{l \text{ 1-Blatt}} \text{rang}(M_l).$$

Letzteres ist gleich der Anzahl der 1-Blätter des Protokollbaumes. Damit folgt, dass die Anzahl der 1-Blätter des Protokollbaumes und damit die Anzahl der Rechtecke in einer Partition von M_f in monochromatische Rechtecke durch $\text{rang}(M_f)$ nach unten beschränkt ist. \square

Für das Beispiel der Gleichheitsfunktion ist wieder leicht zu sehen, dass der Rang der Kommunikationsmatrix gleich 2^n ist. Der Rang der Kommunikationsmatrix von DQF ist dagegen nicht so leicht zu sehen. Daher betrachten wir als zweite Methode für den Beweis unterer Schranken für die Anzahl der monochromatischen Rechtecke in einer Zerlegung von M_f die Methode der Unterscheidungsmengen.

Definition 3.2.11 Eine Menge $S \subseteq X \times Y$ heißt c -Unterscheidungs-*menge* (engl. fooling set), wenn für alle $(x, y) \in S$ gilt, dass $f(x, y) = c$ ist und für alle $(x, y), (x', y') \in S$ mit $x \neq x'$ und $y \neq y'$ gilt, dass $f(x, y') \neq c$ oder $f(x', y) \neq c$.

Man sieht leicht, dass jedes c -Rechteck höchstens ein Element einer c -Unterscheidungs-*menge* enthalten kann. Wenn es zwei verschiedene Elemente (x, y) und (x', y') enthielte, enthielte es wegen der Rechteckeigenschaft auch (x', y) und (x, y') und wäre nach Definition der Unterscheidungs-*menge* nicht monochromatisch. Widerspruch. Damit folgt die folgende Aussage.

Satz 3.2.12 Die Größe einer c -Unterscheidungs-*menge* ist eine untere Schranke für die Anzahl der Rechtecke mit der Farbe c in einer Partition von M_f in monochromatische Rechtecke.

Es ist wieder leicht, die Methode der Unterscheidungs-*menge* auf die Gleichheitsfunktion anzuwenden. Stattdessen betrachten wir die Funktion $\text{DQF}(x_1, \dots, x_n, y_1, \dots, y_n) = x_1 y_1 \vee \dots \vee x_n y_n$. Wir behaupten, dass $\{(x, \bar{x}) \mid x \in \{0, 1\}^n\}$ eine 0-Unterscheidungs-*menge* ist, wobei \bar{x} die komponentenweise Negation von x bezeichnet. Man sieht sofort, dass $\text{DQF}(x, \bar{x}) = 0$ ist, da in jeder Konjunktion $x_i y_i = x_i \bar{x}_i$ eines der beiden Bits gleich 0 ist. Dagegen enthält für $x \neq x'$ das Paar (x, x') oder das Paar (x, \bar{x}') Einsen an derselben Position. Also haben wir eine Unterscheidungs-*menge* mit 2^n Elementen gefunden und es folgt eine untere Schranke von n für die Kommunikationskomplexität von DQF.

Wie kann man nun untere Schranken für die Kommunikationskomplexität nutzen um untere Schranken für die OBDD-Größe zu zeigen? Wir betrachten wieder eine Funktion $f : \{0, 1\}^n \times \{0, 1\}^m \rightarrow \{0, 1\}$. Wir bezeichnen die n Variablen, die den ersten Teil der Eingabe angeben, als L -Variablen und die m Variablen, die den zweiten Teil der Eingabe angeben, als R -Variablen. Sei nun eine Variablenordnung gegeben, in der alle L -Variablen vor allen R -Variablen angeordnet sind. Angenommen es gibt ein OBDD für f mit einer solchen Variablenordnung und mit höchstens s Knoten. Dann hat die Funktion f ein Kommunikationsprotokoll mit höchstens $\lceil \log s \rceil$ gesendeten Bits: Alice simuliert die Rechnung des OBDDs auf ihrem Teil der Eingabe, bis sie einen Knoten erreicht, der mit einer Bob gehörenden Variablen markiert ist. Dann sendet sie die Nummer des Knotens an Bob. Dazu genügen $\lceil \log s \rceil$ Bits. Anschließend kann Bob die Rechnung fortführen, bis er eine Senke erreicht, und den Funktionswert ausgeben. Aus einer OBDD-Größe s erhalten wir eine obere Schranke $\lceil \log s \rceil$ für die Kommunikationskomplexität. Wenn wir andererseits eine untere Schranke von z.B. $\Omega(n)$ für die Kommunikationskomplexität bewiesen haben, folgt sofort die untere Schranke $2^{\Omega(n)}$ für die OBDD-Größe.

Damit haben wir exponentielle untere Schranken für DQF und den Gleichheitstest für die Variablenordnungen bewiesen, bei denen alle x -Variablen vor allen y -Variablen stehen. Diese Beweise entsprechen in einem gewissen Sinne dem Zählen von Subfunktionen, wie wir es in Kapitel 2 vorgeführt haben; sie sind jedoch mit den entsprechenden Grundlagen der Kommunikationskomplexität einfacher zu führen und auch viel anschaulicher, da sie auch einen intuitiven Zusammenhang zwischen OBDD-Größe und Kommunikation herstellen.

Wir beobachten allerdings, dass in dem Kommunikationsprotokoll, das wir aus einem OBDD konstruieren können, nur Alice Information an Bob sendet, aber nichts von Bob erhält. Für diese so genannte Einweg-Kommunikationskomplexität kann man bessere untere Schranken beweisen.

Definition 3.2.13 Sei $f : X \times Y \rightarrow \{0, 1\}$. Eine Menge $S \subseteq X$ heißt Einweg-Unterscheidungs-
menge, wenn es für $x, x' \in S$ mit $x \neq x'$ ein $y \in Y$ gibt, so dass $f(x, y) \neq f(x', y)$
gilt.

Satz 3.2.14 Sei $f : X \times Y \rightarrow \{0, 1\}$ und sei S eine Einweg-Unterscheidungs-
menge für f . Jedes Kommunikationsprotokoll für f , bei dem nur Alice Information an Bob schickt,
benötigt mindestens $\log |S|$ Bits an Kommunikation.

Beweis: Falls Alice für alle $x \in X$ weniger als $\log |S|$ Bits sendet, gibt es verschiedene
Teileingaben x und x' in S , für die Alice dieselbe Kommunikation erzeugt. D.h., Bob kann
 x und x' nicht unterscheiden, was, da es y mit $f(x, y) \neq f(x', y)$ gibt, dazu führt, dass Bob
nicht $f(x, y)$ berechnen kann. Widerspruch. \square

Eine einfache Funktion, die bei den Beweisen von unteren Schranken für die OBDD-Größe
eine wichtige Rolle spielt, ist die Funktion INDEX. Sei $n = 2^k$. Die Funktion $\text{INDEX}_n : \{0, 1\}^n \times \{0, 1\}^k \rightarrow \{0, 1\}$
ist wie folgt definiert. Sei $(x, a) = (x_0, \dots, x_{n-1}, a_0, \dots, a_{k-1})$
eine Eingabe. Der zweite Teil a aus $\log n$ Bits wird als Binärzahl $|a| \in \{0, \dots, n-1\}$ aufge-
fasst. Ausgabe ist dann das Bit $x_{|a|}$. Zur Veranschaulichung benutzen wir auch die Begriffe
Adressvariablen für die Variablen, die die Position des auszugebenden Bits bestimmen, und
den Begriff Datenvariablen für die Bits, die ausgegeben werden können.

Wir beobachten, dass $\{0, 1\}^n$ eine Einweg-Unterscheidungs-
menge für INDEX_n ist: Sei-
en $x, x' \in \{0, 1\}^n$, so dass $x \neq x'$. Dann gibt es eine Position i , für die sich x und x'
unterscheiden. Damit folgt $\text{INDEX}_n(x, i) \neq \text{INDEX}_n(x', i)$. Die Einweg-Kommunikations-
komplexität von INDEX_n beträgt also mindestens n . Wir sehen sofort, dass die OBDD-
Größe für INDEX_n mindestens 2^n beträgt, wenn die Datenvariablen vor den Adressvariablen
getestet werden. Allerdings ist es nicht schwer zu zeigen, dass die OBDD-Größe für alle Va-
riablenordnungen, für die die Adressvariablen vor den Datenvariablen getestet werden, linear
ist.

Bei der Untersuchung von OBDDs möchte man jedoch auch untere Schranken erhalten, die
für alle Variablenordnungen gelten. Auf die Kommunikationskomplexität bezogen „genügt“
es dann untere Schranken zu beweisen, die für alle Partitionen der Variablenmenge in zwei
(ungefähr) gleich große Mengen gelten. Eine andere Möglichkeit besteht darin, Variablen in
geeigneter Weise konstant zu setzen, so dass die entstehende Subfunktion große OBDDs hat,
also dass z.B. die Funktion INDEX oder der Gleichheitstest für eine schlechte Variablenord-
nung entsteht. In Kapitel 2 haben wir bei der Achillesfersenfunktion Variablen so konstant
gesetzt, dass DQF mit schlechter Variablenordnung entsteht. Diese Vorgehensweise wol-
len wir weiterführen. Zunächst stellen wir ein Werkzeug für die Übertragung von unteren
Schranken für die Kommunikationskomplexität zwischen verschiedenen Funktionen vor.

Definition 3.2.15 Seien $f : X \times Y \rightarrow \{0, 1\}$ und $g : A \times B \rightarrow \{0, 1\}$ Funktionen. Wir sagen, dass g rechteckreduzierbar auf f ist ($g \leq_{\text{rect}} f$), wenn es Funktionen $p : A \rightarrow X$ und $q : B \rightarrow Y$ gibt, so dass für alle $(a, b) \in A \times B$ gilt: $f(p(a), q(b)) = g(a, b)$.

Satz 3.2.16 Falls die Kommunikationskomplexität von $g : A \times B \rightarrow \{0, 1\}$ mindestens s beträgt und $g \leq_{\text{rect}} f$, beträgt auch die Kommunikationskomplexität von f mindestens s . Die analoge Aussage gilt für die Einweg-Kommunikationskomplexität.

Beweis: Falls die Kommunikationskomplexität von f den Wert t hat, können wir auch ein Protokoll für g mit Komplexität höchstens t konstruieren. Um $g(a, b)$ zu berechnen, berechnet Alice ohne Kommunikation $p(a)$, und Bob berechnet ohne Kommunikation $q(b)$. Anschließend können Alice und Bob das Protokoll für f simulieren. Da $f(p(a), q(b)) = g(a, b)$, können sie hiermit $g(a, b)$ berechnen. \square

Wir benutzen nun die Rechteckreduktionen zum Beweis von unteren Schranken für die OBDD-Größe für alle Variablenordnungen. Zur Vereinfachung der Darstellung betrachten wir im folgenden Satz nur Variablenmengen mit einer geraden Anzahl von Variablen. Sei $f : \{0, 1\}^n \rightarrow \{0, 1\}$ und sei (L, R) eine Partition der Variablenmenge von f . Dann bezeichnen wir mit $f^{(L,R)} : \{0, 1\}^{|L|} \times \{0, 1\}^{|R|} \rightarrow \{0, 1\}$ dieselbe Funktion, wobei Alice die Variablen aus L und Bob die Variablen aus R bekommt.

Satz 3.2.17 Sei n gerade und sei $f : \{0, 1\}^n \rightarrow \{0, 1\}$ eine Funktion über der Variablenmenge Z . Falls es für jede Partition (L, R) von Z mit $|L| = |R|$ eine Funktion $g : A \times B \rightarrow \{0, 1\}$ mit Einweg-Kommunikationskomplexität mindestens s und eine Rechteckreduktion $g \leq_{\text{rect}} f^{(L,R)}$ gibt, beträgt die OBDD-Größe für f und alle Variablenordnungen mehr als 2^{s-1} .

Beweis: Wir nehmen an, dass es ein OBDD für f mit einer Größe von höchstens 2^{s-1} gibt. Sei (L, R) die Partition der Variablenmenge, bei der L die erste Hälfte der Variablen in der Variablenordnung enthält und R die übrigen Variablen. Sei $g : A \times B \rightarrow \{0, 1\}$ die Funktion, die es nach Voraussetzung für diese Partition gibt. Wir konstruieren wie oben ein Protokoll für die Berechnung von $g(a, b)$. Alice berechnet $p(a)$ und Bob berechnet $q(b)$. Wieder ist keine Kommunikation nötig. Alice wertet das OBDD für f auf $p(a)$ aus, bis sie den ersten mit einer R -Variablen markierten Knoten erreicht, und sendet die Nummer des erreichten Knotens an Bob. Bob vervollständigt dann die Auswertung der Funktion und erhält den Funktionswert, der nach Voraussetzung gleich $g(a, b)$ ist. Für die Codierung der Knotennummer und damit für die gesamte Kommunikation genügen $s - 1$ Bits im Widerspruch zur angenommenen unteren Schranke für die Kommunikationskomplexität von g . \square

Wir wollen nun diese Methode für eine konkrete Funktion anwenden. Die Hidden-Weighted-Bit-Funktion ist definiert durch

$$\text{HWB}(z_1, \dots, z_n) = z_s \quad \text{mit } s = z_1 + \dots + z_n \text{ und } z_0 = 0.$$

Satz 3.2.18 Jedes OBDD für HWB_n hat mindestens $2^{n/8-1}$ Knoten.

Beweis: Sei o.B.d.A. n eine Zweierpotenz und $n \geq 8$. Sei eine Partition (L, R) der Variablenmenge $Z = \{z_1, \dots, z_n\}$ gegeben, so dass $|L| = |R| = n/2$ gilt. Dabei sei $L = \{z_{i(1)}, \dots, z_{i(n/2)}\}$ mit $i(1) < \dots < i(n/2)$.

1. Fall: Unter $z_1, \dots, z_{n/2}$ befinden sich mindestens $n/4$ der Variablen in L .

Dann sind $z_{i(1)}, \dots, z_{i(n/4)}$ in der ersten Hälfte der Variablen enthalten. Insbesondere gilt dann

$$n/8 + 1 \leq i(n/8 + 1) < \dots < i(n/4) \leq n/2.$$

Wir definieren nun die Funktionen $p : A \rightarrow \{0, 1\}^{|L|}$ und $q : B \rightarrow \{0, 1\}^{|R|}$ der Rechteckreduktion von $\text{INDEX}_{n/8}$ auf HWB_n , wobei A die Menge aller Belegungen der Datenvariablen und B die Menge aller Belegungen der Adressvariablen von $\text{INDEX}_{n/8}$ sind. Für die Definition von p sei

$$\begin{aligned} z_{i(n/8+1)} &:= a_0, \\ &\vdots \\ z_{i(n/4)} &:= a_{n/8-1}. \end{aligned}$$

Die übrigen Variablen in L werden so gewählt, dass in allen L -Variablen zusammen genau $n/8$ Einsen vorkommen. Dies ist möglich, da nur $n/8$ der L -Variablen durch die Datenvariablen belegt werden.

Wir definieren nun q . Seien nun $b_0, \dots, b_{\log n - 4}$ die Adressvariablen von $\text{INDEX}_{n/8}$, die einen Wert $b \in \{0, \dots, n/8 - 1\}$ beschreiben. Dann werden die R -Variablen so gewählt, dass sie genau $i(b+n/8+1) - n/8$ Einsen enthalten. Nach Konstruktion liegt $i(b+n/8+1)$ im Bereich zwischen $n/8 + 1$ und $n/2$. Also gibt es auch eine derartige Belegung der R -Variablen.

Die konstruierte Eingabe für HWB enthält dann genau $i(b + n/8 + 1)$ Einsen. Also ist das Bit $z_{i(b+n/8+1)} = a_b$ auszugeben, d.h., es wird wirklich die Funktion INDEX berechnet.

2. Fall: Unter $z_{n/2+1}, \dots, z_n$ befinden sich mindestens $n/4$ der Variablen in L .

Dann sind insbesondere $z_{i(n/4+1)}, \dots, z_{i(n/2)}$ in der zweiten Hälfte der Variablen enthalten. Daraus folgt

$$n/2 + 1 \leq i(n/4 + 1) < \dots < i(3n/8) \leq 7n/8.$$

Wir definieren wieder die Funktionen $p : A \rightarrow \{0, 1\}^{|L|}$ und $q : B \rightarrow \{0, 1\}^{|R|}$ der Rechteckreduktion von $\text{INDEX}_{n/8}$ auf HWB_n . Für p wählen wir

$$\begin{aligned} z_{i(n/4+1)} &:= a_0 \\ &\vdots \\ z_{i(3n/8)} &:= a_{n/8-1}. \end{aligned}$$

Die übrigen L -Variablen werden so belegt, dass alle L -Variablen zusammen genau $n/2 - n/8 = 3n/8$ Einsen enthalten. Da nur $n/8$ der $n/2$ L -Variablen durch die Datenvariablen bestimmt werden, ist dies möglich. Seien nun $b_0, \dots, b_{\log n - 4}$ die Adressvariablen von $\text{INDEX}_{n/8}$, die einen Wert $b \in \{0, \dots, n/8 - 1\}$ beschreiben. Dann werden die R -Variablen so gewählt, dass sie genau $i(b + n/4 + 1) - n/2 + n/8$ Einsen enthalten. Nach Konstruktion

liegt $i(b + n/4 + 1)$ im Bereich zwischen $n/2 + 1$ und $7n/8$. Also gibt es auch eine derartige Belegung. Die konstruierte Eingabe für HWB enthält dann genau $i(b + n/4 + 1)$ Einsen. Also ist das Bit $z_{i(b+n/4+1)} = a_b$ auszugeben, d.h., es wird wirklich die Funktion INDEX berechnet.

Da die untere Schranke für die Kommunikationskomplexität für $\text{INDEX}_{n/8}$ gleich $n/8$ ist, folgt mit Satz 3.2.17 die untere Schranke $2^{n/8-1}$ für die OBDD-Größe für HWB_n und alle Variablenordnungen. \square

4 Anwendungen

4.1 Verifikation des Pentium-Dividierers

Der Dividierer im Pentium ist ein sog. SRT-Dividierer, der von Atkins (1969) beschrieben wurde. Wir behandeln den Dividierer als Beispiel dafür, dass OBDD-Methoden auch für den Schaltkreisentwurf nützlich sein können.

Die Schulmethode für die Division benötigt für die Berechnung eines Quotienten mit n Bits n Iterationen. Jede Iteration besteht aus einem Vergleich, ob der noch vorhandene Rest kleiner ist als der Divisor. In diesem Fall ist das berechnete Bit 0 und anderenfalls 1. Im letzteren Fall wird der Divisor vom noch vorhandenen Rest subtrahiert. Der Rest wird mit 2 multipliziert. Aus dieser Darstellung folgt, dass jede Iteration logarithmische Tiefe benötigt, da sie einen Vergleich und eine Subtraktion enthält. Der SRT-Dividierer variiert die Schulmethode für die Division, so dass konstante Tiefe für jede Iteration genügt. Dazu wird eine redundante Zahlendarstellung, nämlich eine eingeschränkte Radix-4 Darstellung, für den Quotienten benutzt.

Die Vorteile der Radix-4-Darstellung und ihre Eigenschaften wollen wir hier nicht diskutieren.¹ Für das Verständnis des Pentium-Dividierers genügen uns die folgenden Eigenschaften. Sei $m = 4^n + 1$. Die Ziffern der eingeschränkten Radix-4-Darstellung kommen aus der Menge $\{-2, -1, 0, 1, 2\}$. Die Zahl Radix-4-Zahl (x_n, \dots, x_0) stellt dabei die Zahl $(x_n 4^n + x_{n-1} 4^{n-1} + \dots + x_0) \bmod m$ dar.

Für die Beschreibung des Dividierers gehen wir von folgenden Voraussetzungen aus. Der Divisor D liegt im Intervall $[1, 2)$ und der (Divisions-)Rest R im Intervall $[-2, 2)$. Der Rest ist zu Beginn mit dem Dividenden initialisiert. Falls die zu dividierenden Zahlen nicht in diesen Bereichen liegen, ist ein Preprocessing und ein Postprocessing nötig und möglich, um sicherzustellen, dass Dividend und Divisor in den angegebenen Bereichen liegen. Der Divisor wird als Fixkommazahl dargestellt, wobei die Stelle vor dem Komma gleich 1 ist. Für den (Divisions-)Rest R benutzen wir eine redundante Zahlendarstellung, nämlich die Summe von zwei Fixkommazahlen S und C , die beide im Bereich $[-8, 8)$ liegen. Diese Zahlen bestehen aus dem Vorzeichenbit, drei Vorkommastellen und den Nachkommastellen.

Wir beschreiben die i -te Stufe des SRT-Dividierers ($i \geq 0$). Inputs sind der Divisionsrest, gegeben als $S_i + C_i$, sowie der Divisor D . Ausgabe ist die Stelle $Q_{i+1} \in \{-2, \dots, 2\}$ des Quotienten, sowie ein neuer Rest $S_{i+1} + C_{i+1}$. Wir beachten, dass Q_{i+1} eine Stelle einer Radix-4 Zahl ist, deren einzige Vorkommastelle Q_1 und deren Nachkommastellen $Q_2 Q_3 \dots$ sind. Hierbei genügt eine eingeschränkte Radix-4 Darstellung mit Ziffern aus dem Bereich $\{-2, \dots, 2\}$. Alle anderen Zahlen sind Binärzahlen (Zweierkomplement).

Für die Addition von Binärzahlen benutzen wir einen so genannten Carry-Save-Adder (CSA). Für die Addition von drei n -Bit-Zahlen werden n Volladdierer benutzt, wobei der i -te Volladdierer die i -ten Bits der drei Zahlen als Eingabe erhält. Ausgabe sind die Summenbits und die Übertragsbits der Volladdierer. Wenn man die Zahl, die sich aus den Übertragsbits ergibt um eine Stelle nach links verschiebt (mit 2 multipliziert) erhält man zwei Zahlen, der Summe

¹Weitere Einzelheiten finden sich in I. Wegener, Effiziente Algorithmen für grundlegende Funktionen, Teubner 1987, Abschnitt 3.10.

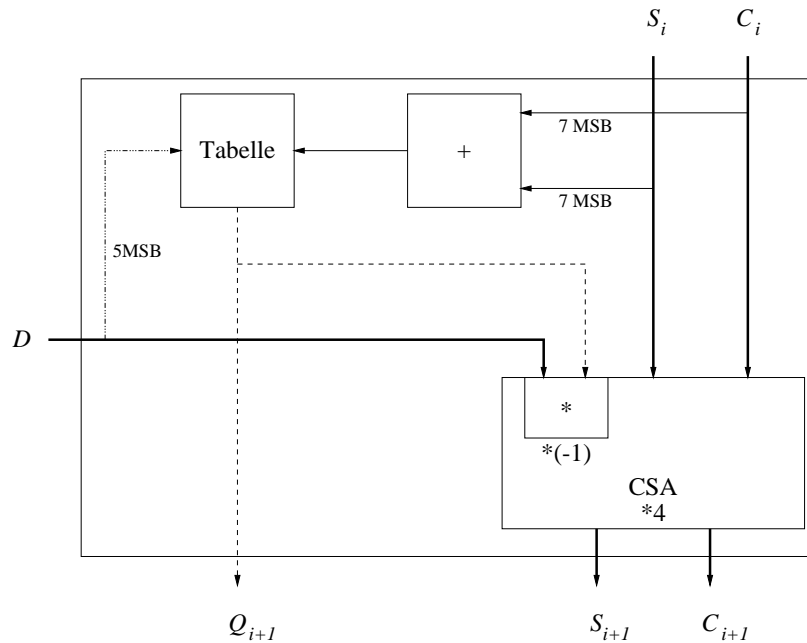


Abbildung 14: Schaltkreis für die Realisierung einer Stufe des SRT-Dividierers

gleich der Summe der drei Zahlen in der Eingabe ist. Die Tiefe eines CSA ist konstant und die Größe linear. Ergebnis sind allerdings zwei Zahlen, was aber in der betrachteten Situation genügt, da der Divisionsrest als Summe von zwei Zahlen dargestellt wird.

Die i -te Stufe arbeitet folgendermaßen (siehe auch Abbildung 14): Seien S'_i und C'_i die Zahlen, die aus den 4 Vorkommabits und den ersten 3 Nachkommabits von S_i und C_i bestehen. Sei D' die Zahl, die aus den 5 höchstwertigen Bits von D besteht. Die Stufe enthält eine Tabelle, die aus $S'_i + C'_i$ und aus D' eine geeignete Zahl $Q_{i+1} \in \{-2, -1, 0, 1, 2\}$ bestimmt. Von dem Divisionsrest $S_i + C_i$ wird $D \cdot Q_{i+1}$ subtrahiert und das Ergebnis wird mit 4 multipliziert, so dass der ein korrekter Divisionsrest entsteht. Der Divisionsrest wird wieder als Summe von zwei Zahlen S_{i+1} und C_{i+1} dargestellt.

Bevor wir diskutieren, warum dieser Schaltkreis korrekt implementiert werden kann (dies ist nichttrivial), überlegen wir, warum eine Stufe in konstanter Tiefe realisiert werden kann. Die Addition von S'_i und C'_i ist in konstanter Tiefe möglich, da beide Zahlen nur aus 7 Bits bestehen. Da die Tabelle konstante Größe hat, kann auch sie in konstanter Tiefe realisiert werden. Die Multiplikation von D mit 0, 1 oder 2 ist einfach. Eine eventuelle Multiplikation mit -1 kann mit Hilfe des nachgeschalteten Carry-Save-Adders durchgeführt werden, da bei der Zweierkomplementdarstellung eine Multiplikation mit -1 durch das Negieren aller Bits und Addition der Zahl mit einer 1 im niederwertigsten Bit realisiert werden kann. Schließlich ist die Addition von S_i , C_i und $-Q_{i+1} \cdot D$ mit einem Carry-Save-Adder in konstanter Tiefe möglich, da das Ergebnis als Summe von zwei Zahlen dargestellt wird.

Wir beobachten, dass die Stelle Q_{i+1} des Quotienten nur aus konstant vielen Bits von S_i , C_i und D bestimmt wird. Es ist a priori nicht klar, dass dies überhaupt möglich ist, also dass es eine geeignete Tabelle gibt, die ein korrektes Q_{i+1} bestimmt. Damit dies möglich ist,

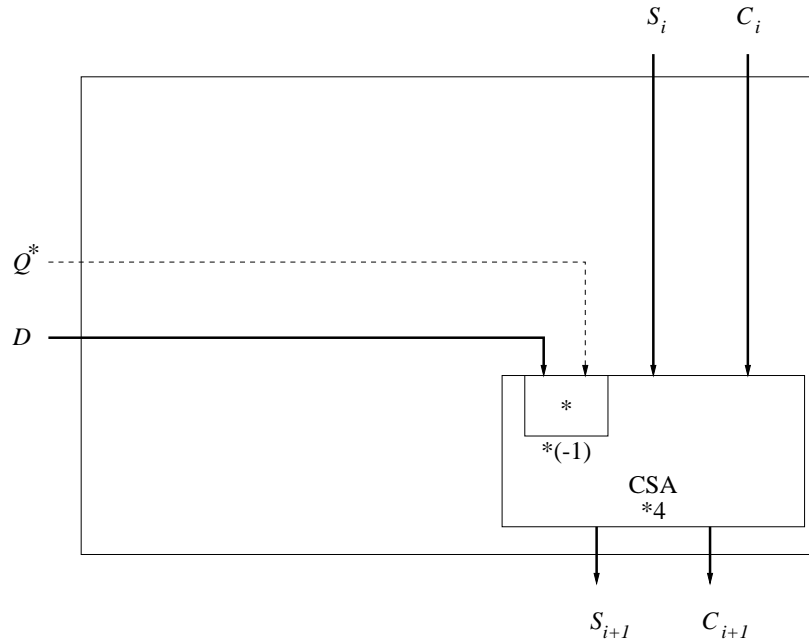


Abbildung 15: Hilfsschaltkreis zur Berechnung der zulässigen Tabelleneinträge

benötigen wir noch eine weitere Voraussetzung. Wir beschreiben im Folgenden auch, was es überhaupt heißt, dass Q_{i+1} korrekt ist.

Die weitere Voraussetzung, die wir benötigen, besteht darin, dass

$$-8D \leq 3(S_i + C_i) \leq 8D$$

gilt. Da vor der ersten Iteration $S_0 + C_0$ gleich dem Dividenden ist und damit im Intervall $[-2, 2)$ liegt und nach Voraussetzung $D \in [1, 2)$ ist, ist diese Voraussetzung zu Beginn erfüllt. Die i -te Stufe arbeitet korrekt, wenn gilt: Falls $-8D \leq 3(S_i + C_i) \leq 8D$ gilt, dann folgt

$$\begin{aligned} -8D \leq 3(S_{i+1} + C_{i+1}) \leq 8D & \quad \text{und} \\ (S_{i+1} + C_{i+1}) = 4 * (S_i + C_i - Q_{i+1}D) & \quad \text{wobei } S_{i+1}, C_{i+1} \in [-8, 8). \end{aligned}$$

Die erste Bedingung stellt sicher, dass die Voraussetzung auch für die nachfolgende Stufe gilt, und die zweite Bedingung, dass korrekt dividiert wurde und kein Überlauf eintritt.

Aus den Bedingungen folgt zusammen mit $D \in [1, 2)$, dass $-16/3 \leq S_i + C_i \leq 16/3$. Es ist dann naheliegend, dass dann auch $S'_i + C'_i$ nicht viel kleiner als $-16/3$ und nicht viel größer als $16/3$ sein kann. Daher genügt eine Tabelle mit 2^7 Zeilen (für $S'_i + C'_i$) und 2^4 Spalten (für D' , hierbei beachten wir, dass das höchstwertige Bit von D' gleich 1 ist).

Zur Berechnung der korrekten Tabelleneinträge konstruieren wir zunächst den in Abbildung 15 gezeigten Teilschaltkreis, der beschreibt, wie aus dem in der Tabelle gelesenen Ergebnis Q^* , sowie aus den Eingaben S_i , C_i und D die Outputs S_{i+1} und C_{i+1} der i -ten Stufe berechnet werden. Wir interpretieren die Outputs als boolesche Funktionen $S_{i+1}(Q^*, S_i, C_i, D)$ und $C_{i+1}(Q^*, S_i, C_i, D)$. Für diese Funktionen können OBDDs berechnet werden.

Was ist ein korrekter Tabelleneintrag an der Stelle (R^*, D^*) ? Der Eintrag Q^* ist zulässig, falls für alle Zahlen S_i, C_i und D , aus $R^* = S'_i + C'_i$, aus $D^* = D'$ und aus $-8D \leq 3(S_i + C_i) \leq 8D$ folgt, dass $-8D \leq 3(S_{i+1}(Q^*, S_i, C_i, D) + C_{i+1}(Q^*, S_i, C_i, D)) \leq 8D$ und $(S_{i+1}(Q^*, S_i, C_i, D) + C_{i+1}(Q^*, S_i, C_i, D)) = 4 * (S_i + C_i - Q^*D)$, wobei $S_{i+1}(Q^*, S_i, C_i, D), C_{i+1}(Q^*, S_i, C_i, D) \in [-8, 8]$ gilt. Dies ist offensichtlich eine Bedingung, die nicht leicht von Hand getestet werden kann (da sie für *alle* Belegungen von S_i, C_i und D geprüft werden muss). Im Folgenden bezeichne [Bedingung] den Wert 1, wenn die Bedingung wahr ist, und 0 anderenfalls. Die Funktionen

$$\begin{aligned} f_1 &= [R^* = S'_i + C'_i] \\ f_2 &= [D^* = D'] \\ f_3 &= [-8D \leq 3(S_i + C_i) \leq 8D] \\ f_4 &= [-8D \leq 3(S_{i+1}(Q^*, S_i, C_i, D) + C_{i+1}(Q^*, S_i, C_i, D)) \leq 8D] \\ f_5 &= [(S_{i+1}(Q^*, S_i, C_i, D) + C_{i+1}(Q^*, S_i, C_i, D)) = 4 * (S_i + C_i - Q^*D)] \\ &\quad \wedge [S_{i+1}(Q^*, S_i, C_i, D), C_{i+1}(Q^*, S_i, C_i, D) \in [-8, 8]] \end{aligned}$$

können mit OBDDs dargestellt werden. Dann wird ein OBDD für die folgende Funktion konstruiert.

$$(\forall S_i, C_i, D) : f_1 \wedge f_2 \wedge f_3 \Rightarrow f_4 \wedge f_5$$

Der Ausdruck rechts vom Allquantor hat die folgende Interpretation: Wenn die vorliegende Eingabe S_i, C_i, D nicht korrekt ist oder für diese Eingabe nicht auf den Tabelleneintrag (R^*, D^*) zugegriffen wird, nehmen f_1, f_2 oder f_3 den Wert 0 an, und damit nimmt der Ausdruck den Wert 1 an (dieser Fall interessiert nicht). Wenn die Eingabe S_i, C_i, D korrekt ist und dafür auf den Tabelleneintrag (R^*, D^*) zugegriffen wird, nimmt der Ausdruck den Wert 1 an, wenn die i -te Stufe für den Tabelleneintrag Q^* korrekt arbeitet. Durch die Quantifizierung erhält man eine Funktion, die nur noch von R^*, D^* und Q^* abhängt und den Wert 1 annimmt, wenn für alle Eingaben S_i, C_i und D_i der Tabelleneintrag Q^* ein korrekter Eintrag für die Position (R^*, D^*) ist. Aus dem berechneten OBDD können also die korrekten Tabelleneinträge entnommen werden.

Bryant war in der Lage, alle erlaubten Tabelleneinträge zu bestimmen. Da kein Tabelleneintrag leer blieb, folgt daraus, dass die konstant vielen Bits von S_i, C_i und D ausreichen, um einen zulässigen Quotienten Q_{i+1} zu bestimmen.

Wir wollen hier nur „theoretisch“ überlegen, wie die Tabelle für die Berechnung des höchstwertigen Bits Q_1 aussieht². Die größte Zahl Q , die in der Radix-4 Darstellung mit Ziffern aus dem Bereich $\{-2, \dots, 2\}$ mit einer Stelle vor dem Dezimalpunkt darstellbar ist, ist kleiner als

$$\sum_{-\infty < i \leq 0} 2 \cdot 4^i = 8/3.$$

²Die Tabelle befindet sich in R. E. Bryant, „Bit-Level Analysis of an SRT Divider Circuit“, 33rd Design Automation Conference, 1996, 661–665, www.cs.cmu.edu/~bryant/pubdir/dac96b.ps

Dies ist der Grund, warum $-8D \leq 3(S_0 + C_0) \leq 8D$ vorausgesetzt wird. Falls $Q_1 = 2$, ist der Quotient kleiner als $8/3$ und größer als

$$2 + \sum_{-\infty < i \leq -1} (-2) \cdot 4^i = 4/3.$$

Daher ist $Q_1 = 2$ nur erlaubt, wenn $4D \leq 3(S_0 + C_0) \leq 8D$. Ähnliche Berechnungen führen zu

- $Q_1 = 1$ ist nur erlaubt, wenn $D \leq 3(S_0 + C_0) \leq 5D$ gilt.
- $Q_1 = 0$ ist nur erlaubt, wenn $-2D \leq 3(S_0 + C_0) \leq 2D$ gilt.
- $Q_1 = -1$ ist nur erlaubt, wenn $-5D \leq 3(S_0 + C_0) \leq -D$ gilt.
- $Q_1 = -2$ ist nur erlaubt, wenn $-8D \leq 3(S_0 + C_0) \leq -4D$ gilt.

Anschaulich bedeutet die Bedingung für $Q_1 = 2$, dass man in die Tabelle Geraden mit den Steigungen $4/3$ und $8/3$ einzeichnen kann, und der Wert $Q_1 = 2$ nur in den Tabelleneinträgen zwischen diesen Geraden erlaubt ist. (Hierbei gehen wir davon aus, dass auf der x -Achse D und auf der y -Achse $S_0 + C_0$ aufgetragen ist.) Analoges gilt für die anderen Werte von Q_1 . Die Werte oberhalb der Geraden mit der Steigung $8/3$ und unterhalb der Geraden mit der Steigung $-8/3$ sind unbedeutend, da diese Situation nach der Voraussetzung nicht vorkommen kann.

Wir beachten allerdings, dass tatsächlich $Q_1 = 2$ nicht in allen Tabelleneinträgen zwischen den Geraden mit den Steigungen $4/3$ und $8/3$ erlaubt sein muss, da wir nur mit Näherungen für den Divisionsrest (eben den höchstwertigen 7 Bits von S_0 und C_0) arbeiten. Aus demselben Grund gibt es auch Tabelleneinträge oberhalb der Geraden mit der Steigung $8/3$, die den Wert 2 haben müssen und nicht, wie oben angedeutet, unbedeutend sind. Es wird berichtet, dass der Fehler im Pentium-Dividierer darin bestand, dass einige Tabelleneinträge oberhalb der Geraden mit der Steigung $8/3$, die den Wert 2 haben müssen, fälschlicherweise den Wert 0 hatten.

4.2 Verifikation von sequentiellen Schaltkreisen

4.2.1 Flip-Flops und das Huffman-Modell für sequentielle Schaltkreise

Sequentielle Schaltkreise (manchmal auch Schaltwerke genannt) unterscheiden sich von (den sog. kombinatorischen) Schaltkreisen dadurch, dass sie auch Rückkopplungen enthalten dürfen, d.h., der Ausgang jedes Bausteins G darf auch wieder die Eingänge von G beeinflussen. Bei dieser allgemeinen Variante von sequentiellen Schaltkreisen handelt man sich viele Probleme ein, z.B. kann es passieren, dass ein solcher Schaltkreis keine stabilen Zustände hat: Man betrachte nur einen Negationsbaustein, dessen Ausgang mit seinem Eingang verbunden ist. Daher beschränkt man sich in der Regel auf sequentielle Schaltkreise, deren Rückkopplungen in speziellen Speicherbausteinen, den Flip-Flops realisiert werden. Wir werden im Folgenden zunächst zwei einfache Flip-Flop Varianten diskutieren und dann

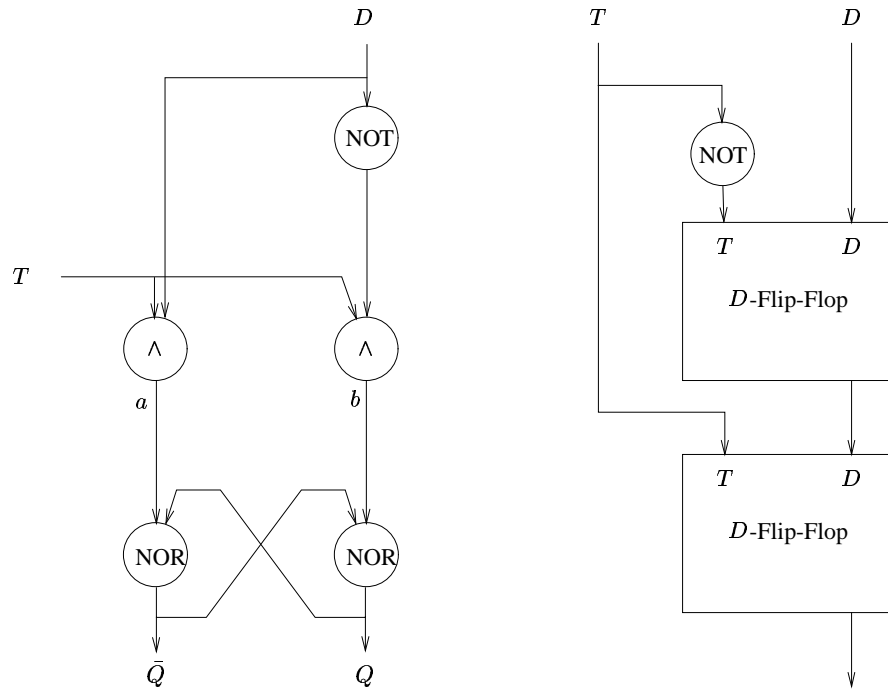


Abbildung 16: Ein D -Flip-Flop und ein Master-Slave Flip-Flop

das Huffman-Modell für sequentielle Schaltkreise. Am Ende dieses Abschnitts geben wir auch ein Beispiel für einen einfachen sequentiellen Schaltkreis an.

Schon aus dieser Beschreibung von sequentiellen Schaltkreisen folgt, dass es bei ihrer Verifikation andere und weitere Problem als bei der Verifikation von gewöhnlichen Schaltkreisen gibt. Als Beispiel für ein solches Problem behandeln wir im Abschnitt 4.2.2 die Erreichbarkeitsanalyse von sequentiellen Schaltkreisen.

Die linke Seite von Abbildung 16 zeigt ein sogenanntes D -Flip-Flop. Man erarbeitet sich leicht die Funktionsweise. Nehmen wir zunächst an, dass der Input T auf 1 liegt. Dann ist $a = D$ und $b = \bar{D}$. Durch Unterscheidung der Fälle $D = 0$ und $D = 1$ verifiziert man leicht, dass $Q = D$ und $\bar{Q} = \bar{D}$ gilt. Nehmen wir nun an, dass der Input T auf 0 wechselt. Offensichtlich hat der Input D nun keinen Einfluss mehr auf die Outputs. Die Rückkopplungen von den Ausgängen Q und \bar{Q} sorgen nun dafür, dass sich bei dem Wechsel von $T = 1$ nach $T = 0$ die Outputs nicht ändern. Insgesamt ergibt sich, dass bei $T = 1$ der Eingang D zum Ausgang Q „durchgeschaltet“ wird. Bei einem Wechsel von $T = 1$ nach $T = 0$ wird der Zustand der Ausgänge erhalten, und solange $T = 0$ ist, hat der Eingang D keinen Einfluss mehr auf die Ausgänge. Also ist ein D -Flip-Flop ein Speicherbaustein, der den Eingang D zu dem Zeitpunkt des Wechsels von T von 1 auf 0 speichert.

Eine Zusammenschaltung von zwei D -Flip-Flops ist in Abbildung 16 rechts gezeigt. Diese Anordnung heißt auch „Master-Slave Flip-Flop“. Wir beachten, dass die T -Eingänge der beiden Flip-Flops immer verschieden sind. Dies führt dazu, dass in Abhängigkeit vom Input T immer eines der beiden Flip-Flops den Eingang „durchschaltet“, während das andere seinen jeweiligen Zustand speichert. Bei festem T ändert sich der Ausgang nicht, wenn sich der

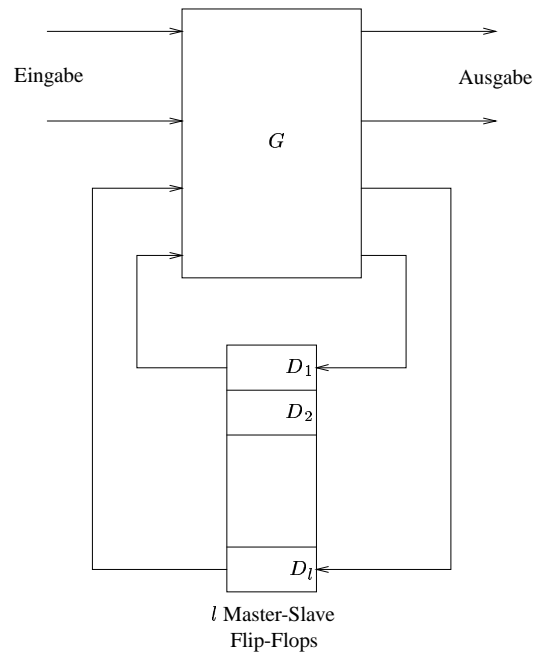


Abbildung 17:

Eingang D verändert, da immer eines der beiden Flip-Flops „speichert“. Zu dem Zeitpunkt, wo T von 0 auf 1 wechselt, speichert das erste Flip-Flop den Wert der Leitung D , und das zweite Flip-Flop schaltet diesen Wert zum Ausgang durch. Wenn dann T von 1 auf 0 wechselt, speichert das zweite Flip-Flop den Wert. Insgesamt erhalten wir einen Speicherbaustein, der das Speichern bei einem „0-1-Takt“ an T ausführt und ansonsten Veränderungen am Eingang D nicht weiterleitet. Dies ist die Eigenschaft, die D -Flip-Flops nicht haben und die wir im Folgenden benötigen.

Im Huffman-Modell für sequentielle Schaltkreise betrachtet man Schaltkreise, deren Rückkopplung nur über Master-Slave Flip-Flops erfolgt (siehe auch Abbildung 17). Der Schaltkreis besteht also aus zwei Teilen, nämlich aus l Flip-Flops in der Rückkopplung und einem kombinatorischem Schaltkreis G . Alle Flip-Flops werden mit demselben Takt T angesteuert. Der Inhalt der Flip-Flops wird auch als Zustand des Schaltkreises bezeichnet. Der kombinatorische Schaltkreis G berechnet aus den Ausgängen der Flip-Flops (also dem Zustand) und der Eingabe die Ausgabe und den neuen Input für die Flip-Flops, also den neuen Zustand s' . Zu dem Zeitpunkt, wo T von 0 auf 1 wechselt, wird dieser Nachfolgezustand in den Flip-Flops gespeichert, d.h., s' wird zum aktuellen Zustand. Die Wechsel von T von 0 auf 1 passieren in genügend großen Zeitabständen, so dass G jeweils den Nachfolgezustand des aktuellen Zustands berechnen kann.

Formal kann der sequentielle Schaltkreis beschrieben werden durch sein Eingabealphabet $I = \{0, 1\}^n$, sein Ausgabealphabet $O = \{0, 1\}^m$, seine Zustandsmenge $S = \{0, 1\}^l$, seine Übergangsfunktion $\delta : I \times S \rightarrow S$, seine Ausgabefunktion $\lambda : I \times S \rightarrow O$ und seinen Startzustand. Offensichtlich hat der Schaltkreis 2^l Zustände.

Zur Illustration der sequentiellen Schaltkreise behandeln wir das einfache Beispiel einer

Ampelschaltung für eine Fußgängerampel. Die Ampel für die Straße kann Rot (SR~Straße-rot), Gelb (SY), Grün (SG) und Rot-Gelb (SRY) anzeigen, die Fußgängerampel Rot (FR) und Grün (FG). Die Ampel steht in der Regel auf SG und FR. Es gibt einen Drucktaster für die Fußgänger, und wenn diese Taste gedrückt wird, durchläuft die Ampel die folgende Folge: 1 Zeittakt SY, FR, 1 Zeittakt SR, FR, 7 Zeittakte SR, FG, 1 Zeittakt SR, FR, 1 Zeittakt SRY, FR und geht anschließend wieder in den Zustand SG, FR.

Zum Entwurf eines sequentiellen Schaltkreises legen wir zunächst I und O fest. Wir wählen $I = \{0, 1\}$ (gibt an, ob die Taste gedrückt wurde), und $O = \{0, 1\}^5$ (gibt für die 5 Lampen in der Ampel an, ob sie leuchten oder nicht). Für die Zustandsmenge wählen wir $S = \{0, 1\}^6$ mit der folgenden Interpretation: Das erste Flip-Flop s_1 repräsentiert FR, das zweite Flip-Flop s_2 SR, das dritte Flip-Flop s_3 SY und die übrigen drei Flip-Flops s_4, s_5, s_6 bilden einen 3-Bit-Zähler, um die Zeit zu messen, in der die Fußgängerampel auf Grün steht. Die Ausgabefunktion λ ergibt sich offensichtlich auf die folgende Weise:

$$\begin{aligned}\lambda_{FR}(t, s_1, \dots, s_6) &= s_1 \\ \lambda_{FG}(t, s_1, \dots, s_6) &= \bar{s}_1 \\ \lambda_{SR}(t, s_1, \dots, s_6) &= s_2 \\ \lambda_{SY}(t, s_1, \dots, s_6) &= s_3 \\ \lambda_{SG}(t, s_1, \dots, s_6) &= \bar{s}_2 \wedge \bar{s}_3\end{aligned}$$

Von den 64 möglichen Zuständen kommen nur 12 vor. (Man könnte also auch mit 4 Flip-Flops auskommen.) Nach der obigen Beschreibung genügt es, eine Übergangsfunktion zu benutzen, für die folgendes gilt:

$$\begin{aligned}\delta(0; 1, 0, 0, 0, 0, 0) &= (1, 0, 0, 0, 0, 0) \\ \delta(1; 1, 0, 0, 0, 0, 0) &= (1, 0, 1, 0, 0, 0) \\ \delta(X; 1, 0, 1, 0, 0, 0) &= (1, 1, 0, 0, 0, 0) \\ \delta(X; 1, 1, 0, 0, 0, 0) &= (0, 1, 0, 1, 1, 0) \\ \delta(X; 0, 1, 0, 1, 1, 0) &= (0, 1, 0, 1, 0, 1) \\ &\vdots \\ \delta(X; 0, 1, 0, 0, 0, 0) &= (1, 1, 0, 0, 0, 1) \\ \delta(X; 1, 1, 0, 0, 0, 1) &= (1, 1, 1, 0, 0, 0) \\ \delta(X; 1, 1, 1, 0, 0, 0) &= (1, 0, 0, 0, 0, 0)\end{aligned}$$

Hierbei steht X als Abkürzung für beide Belegungen des jeweiligen Inputs. Der Startzustand ist $(1, 1, 0, 0, 0, 1)$. Die Funktionsweise ist leicht zu verstehen: Sobald die Taste gedrückt wurde, durchläuft die Ampel den oben genannten Zyklus. Sobald die Fußgänger Grün haben, zählt der Zähler rückwärts von 6 bis 0. Der Zustand SR, FR kommt zweimal in dem Zyklus vor. Um diese beiden Zustände zu unterscheiden, wird der zweite dieser Zustände mit einer 1 in s_6 codiert. Es ist nun einfach mit Standardmethoden Schaltkreise (z.B. PLAs) für die Übergangs- und Ausgabefunktion zu konstruieren.

Bei diesem kleinen Beispiel ist die Verifikation der Korrektheit einfach, da es möglich ist, alle Zustände zu betrachten. Allerdings wächst die Anzahl der (theoretisch möglichen) Zustände

exponentiell in der Anzahl der Flip-Flops, so dass es schon für ca. 40 bis 50 Flip-Flops unmöglich wird, alle Zustände zu betrachten. Wir diskutieren zwei Probleme, für die es hilfreich ist, alle Zustände zu betrachten.

Das erste ist der Äquivalenztest von zwei sequentiellen Schaltkreisen

$C_1 = (I, O, S_1, \delta_1, \lambda_1, s^1)$ und $C_2 = (I, O, S_2, \delta_2, \lambda_2, s^2)$. Diese Schaltkreise haben also dasselbe Eingabealphabet und dasselbe Ausgabealphabet, dürfen aber verschiedene Zustandsmengen (und verschiedene Anzahlen von Flip-Flops), verschiedene Ausgabefunktionen und verschiedene Startzustände haben. Aus der Automatentheorie ist bekannt, dass man zu zwei endlichen Automaten einen Produktautomaten konstruieren kann (dies ist ähnlich zur OBDD-Synthese), und auch hier können wir einen Produktautomaten

$$C = (I, \{0, 1\}, S_1 \times S_2, \delta, \lambda, (s^1, s^2))$$

konstruieren, der einen simultanen Lauf von C_1 und C_2 simuliert, d.h.

$$\delta(\sigma, (s, t)) = (\delta_1(\sigma, s), \delta_2(\sigma, t)),$$

und genau dann eine 1 ausgibt, wenn C_1 und C_2 verschiedene Ausgaben berechnen, d.h.

$$\lambda(\sigma, (s, t)) = 1 \Leftrightarrow [\lambda_1(\sigma, s) \neq \lambda_2(\sigma, t)].$$

Offensichtlich sind die Schaltkreise C_1 und C_2 genau dann äquivalent, wenn in C kein Zustand (s, t) erreichbar ist, für den es einen Input $\sigma \in I$ mit $\lambda(\sigma, (s, t)) = 1$ gibt.

Als zweite Anwendung der Erreichbarkeitsanalyse betrachten wir das Erkennen von „unsicheren“ Zuständen. Es ist leicht einzusehen, dass in dem Ampelbeispiel der Zustand $(0, 0, 0, 0, 0, 0)$ ein unsicherer Zustand (beide Ampeln zeigen grün) ist, der unter allen Umständen vermieden werden muss. Wir haben verifiziert, dass dieser Zustand nicht auftritt, indem wir alle vom Startzustand erreichbaren Zustände aufgezählt haben. Wenn dies bei 2^{50} Zuständen nicht mehr in akzeptabler Zeit möglich ist, benötigen wir andere Methoden, um sicherzustellen, dass unsichere Zustände nicht erreichbar sind. Dies ist offensichtlich eine Anwendung der Erreichbarkeitsanalyse.

4.2.2 Die Erreichbarkeitsanalyse

Wir verwenden das folgende Programm, um die Menge der erreichbaren Zustände zu berechnen:

```

i := 0;  $S_{-1} = \emptyset$ ;
 $S_0 = \{\text{Startzustand}\}$ ;
Solange  $S_{i-1} \neq S_i$ 
    i := i + 1;
     $S_i := S_{i-1} \cup \bigcup_{\sigma \in I} \delta(\sigma, S_{i-1})$ ;
Ausgabe: Die Menge der erreichbaren Zustände ist  $S_i$ .

```

Es ist leicht zu sehen, dass S_i jeweils die Menge der Zustände ist, die in höchstens i Schritten vom Startzustand aus erreichbar sind (Induktion). Die Schleife realisiert eine Fixpunktberechnung, d.h., es wird solange das i erhöht, bis keine neuen Zustände mehr dazukommen.

Dieses Programm ist natürlich nicht praktisch ausführbar, wenn die betrachteten Mengen sehr groß werden und das Programm sie *explizit*, d.h. durch Listen, Bäume o.Ä. speichert. Eine in vielen Fällen kompaktere Darstellung sind OBDDs oder Erweiterungen von OBDDs. Wir untersuchen nun, wie die Mengen und die Übergangsfunktion durch OBDDs dargestellt werden können.

Die Menge aller Zustände ist eine Teilmenge von $\{0, 1\}^l$. Wir können Mengen M von Zuständen durch ihre charakteristische Funktion $\chi_M : \{0, 1\}^l \rightarrow \{0, 1\}$ darstellen, d.h., $s \in M$ genau dann, wenn $\chi_M(s) = 1$. Zur Vereinfachung der Notation identifizieren wir im Folgenden Mengen und ihre charakteristischen Funktionen, d.h., es ist genau dann $M(s) = 1$, wenn $s \in M$. Eine Mengenvereinigung entspricht dann der Synthese der charakteristischen Funktionen mit dem Operator \vee , der Mengendurchschnitt der Synthese mit \wedge , der Test, ob $s \in M$, ist die Auswertung der charakteristischen Funktion $M(s)$, und die Berechnung von $|M|$ ist die Operation Erfüllbarkeit-Anzahl.

In ähnlicher Weise können wir Übergangsfunktionen oder (allgemeiner) Übergangsrelationen durch ihre charakteristischen Funktionen darstellen. Eine Übergangsrelation δ ist eine Teilmenge von $I \times S \times S$, und $(\sigma, s, t) \in \delta$ genau dann, wenn t ein Nachfolgezustand von s bei Eingabe σ ist. Die charakteristische Funktion von δ ist dann eine Funktion $\delta : I \times S \times S \rightarrow \{0, 1\}$.

Im ersten Schritt beseitigen wir die Abhängigkeit von der Eingabe des Schaltkreises. Die Relation $\delta' \subseteq S \times S$ ist definiert durch $(s, t) \in \delta'$ genau dann, wenn es eine Eingabe $\sigma \in I$ mit $(\sigma, s, t) \in \delta$ gibt. Die Relation gibt also an, ob es überhaupt möglich ist, in einem Schritt von s nach t zu kommen. Die charakteristische Funktion von δ' kann mit der Operation Quantifizierung auf die folgende Weise berechnet werden:

$$\delta'(s, t) = (\exists \sigma_1) \cdots (\exists \sigma_n) : \delta((\sigma_1, \dots, \sigma_n), s, s').$$

Hierbei ist $\sigma = (\sigma_1, \dots, \sigma_n)$ die Zerlegung der Eingabe in ihre Bits. Man erhält S_i aus S_{i-1} durch die folgende Beschreibung der Menge S_i :

$$S_i = S_{i-1} \cup \{t \mid (\exists s \in S_{i-1}) : (s, t) \in \delta'\}.$$

Auch dies ist durch Quantifizierung realisierbar. Sei $s = (s_1, \dots, s_l)$ und $t = (t_1, \dots, t_l)$ die Beschreibung der Zustände durch Bitfolgen. Die folgende Berechnung von S_i aus S_{i-1} wird als Image-Computation bezeichnet:

$$S_i(t_1, \dots, t_l) = S_{i-1}(t_1, \dots, t_l) \vee (\exists s_1) \cdots (\exists s_l) : S_{i-1}(s_1, \dots, s_l) \wedge \delta'(s_1, \dots, s_l, t_1, \dots, t_l).$$

Also haben wir die Berechnungen im oben aufgeführten Programm auf die Ausführung von Operationen auf booleschen Funktionen zurückgeführt. Wenn wir die booleschen Funktionen durch OBDDs darstellen, haben wir außerdem eine Variablenordnung zu wählen. Es ist naheliegend, *interleaved variable orderings* zu benutzen, also Variablenordnungen, wo s_i

und t_i zusammen getestet werden. Natürlich sind durch die längeren Folgen von Quantifizierungen exponentielle blow-ups der OBDD-Größe möglich. Daher verwendet man weitere Verfeinerungen, z.B. kompaktere OBDD-Varianten, oder man verwendet Heuristiken, die „geschickte“ Reihenfolgen für die Ausführung der Quantifizierungen bestimmen.

Wenn schließlich die Menge der erreichbaren Zustände berechnet worden ist, kann mit einer Durchschnittsberechnung getestet werden, ob ein Zustand mit der Ausgabe 1 oder ein unsicherer Zustand erreichbar ist. (Es ist natürlich auch möglich und naheliegender, in jeder Iteration des oben aufgeführten Programms zu testen, ob S_i einen solchen Zustand enthält.)

Eine andere Variante besteht darin, mit der Menge der unerwünschten Zustände (der Zustände mit Ausgabe 1 oder der unsicheren Zustände) zu beginnen und die Menge der Zustände zu berechnen, von denen aus ein solcher Zustand erreichbar ist. Am Ende (oder während jeder Iteration) kann dann getestet werden, ob der Startzustand in dieser Menge enthalten ist. Wir benutzen also folgendes Programm:

Sei T_0 die Menge der unsicheren Zustände; $i := 0$; $T_{-1} = \emptyset$.

Solange $T_i \neq T_{i-1}$

$i := i + 1$;

$T_i = T_{i-1} \cup \{s \mid \exists t \in T_{i-1} : (s, t) \in \delta'\}$;

Ausgabe: T_i ist die Menge der Zustände, von denen ein unsicherer Zustand aus erreichbar ist

In diesem Programm ist T_i die Menge der Zustände, von denen aus in höchstens i Schritten ein unsicherer Zustand erreichbar ist. Die Berechnung der Menge der möglichen Vorgänger von Zuständen in diesem Programm wird auch als Pre-Image Computation bezeichnet. Die Darstellung der Mengen und Relationen durch ihre charakteristischen Funktionen wird wie bei dem ersten Programm realisiert.

Wieviele Iterationen brauchen die Programme? Wenn der sequentielle Schaltkreis lange Zyklen enthält, was zum Beispiel bei Zählern der Fall ist, brauchen diese Programme auch viele Iterationen, bis sie terminieren. Als Ausweg wurde ein Verfahren vorgeschlagen, das man als iteriertes Quadrieren bezeichnet. Wir bezeichnen mit $P(R, S, k)$ die Funktion, die den Wert 1 annimmt, wenn ein Zustand in der Menge R von einem Zustand in der Menge S in höchstens 2^k Schritten erreichbar ist. Es ist leicht einzusehen, dass

$$P(R, S, k) = (\exists t) : P(R, \{t\}, k - 1) \wedge P(\{t\}, S, k - 1),$$

was einen rekursiven Algorithmus zur Berechnung von $P(R, S, k)$ nahelegt. Die Tiefe der Rekursion dieses Algorithmus ist nur logarithmisch in der Länge des längsten Zyklus des Schaltkreises. Es wird berichtet, dass die OBDD-Darstellungen der Funktionen $P(\cdot)$ viel größer als die OBDD-Darstellungen bei den vorher behandelten Algorithmen sind, so dass der Ansatz des iterierten Quadrierens nur bei einfachen Schaltkreisen wie Zählern erfolgreich angewendet werden kann.

Wir haben an den Darstellungen der Programme für die Erreichbarkeitsanalyse gesehen, dass OBDDs nur eine Möglichkeit für die Darstellung der betrachteten Mengen und Relationen

sind, da die Programme unabhängig von der Darstellungsform der charakteristischen Funktionen sind. Weiterhin sind OBDDs nur eine Heuristik, die in manchen Fällen anwendbar ist, wo konventionelle Ansätze nicht mehr durchführbar sind. Man sollte also nicht erwarten, dass sich jedes Problem mit OBDDs lösen lässt. In vielen Fällen ist eine Lösung auch nur mit vielen Implementierungstricks und weiteren Verfeinerungen möglich. Da bei den vielen Quantifizierungen die OBDDs sehr groß werden können, ist es schwierig, theoretisch zu untersuchen, in welchen Fällen OBDDs erfolgreich sein können.