Begleitmaterial zur Vorlesung

Binary Decision Diagrams

Wintersemester 2006/2007

Detlef Sieling

Universität Dortmund

FB Informatik, LS 2

44221 Dortmund

Inhaltsverzeichnis

1 Einleitung und Überblick

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.

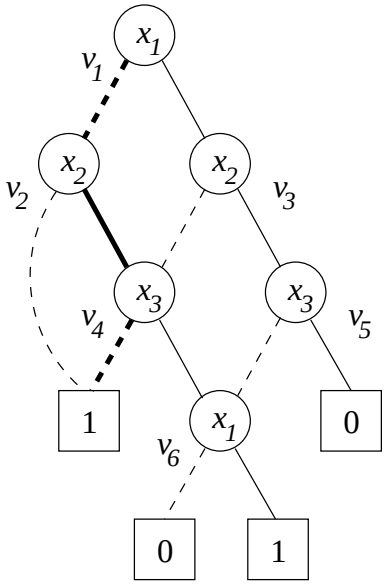
Darstellungen von booleschen Funktionen, wie auch von Mengen und Relationen werden auch in vielen Programmen benötigt. Mengen und Relationen können dabei durch ihre charakteristischen Funktionen dargestellt werden, die wiederum boolesche Funktionen sind. Varianten von BDDs werden insbesondere in Programmen für Hardwareentwurf und -verifikation, Logiksynthese, Analyse von endlichen Automaten oder Testmustergenerierung benutzt. Man kann auch Probleme wie die Kanalverdrahtung, das Färben von Graphen oder die Berechnung von maximalen Flüssen durch boolesche Funktionen codieren und mit Hilfe von BDDs 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 Branchingproramm 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 ein äquivalentes 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 eines 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 nichtuniformen Berechnungsmodellen (z.B. Schaltkreisen 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 dann den Wert 1 annimmt, wenn M auf x hält. Wenn wir nun diese Funktion in Teilfunktionen für jede Eingabelänge zerlegen, erhalten wir Funktio-

nen 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 (von Turingmaschinen) 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:

$$\begin{aligned} f_{v_1} &= \bar{x}_2 \vee \bar{x}_3 \\ f_{v_2} &= x_1 \vee \bar{x}_2 \vee \bar{x}_3 \\ f_{v_3} &= x_1\bar{x}_2 \vee \bar{x}_2\bar{x}_3 \vee x_1x_2\bar{x}_3 \\ f_{v_4} &= x_1 \vee \bar{x}_3 \\ f_{v_5} &= x_1\bar{x}_3 \\ f_{v_6} &= x_1 \end{aligned}$$

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 $f(x_1, \ldots, x_n)$ durch ein BDD dargestellt werden kann. Wir zerlegen f mit der sogenannten 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 Entscheidungsbäume, weil der zugrundeliegende Graph ein Baum ist.

Wenn wir auf diese Weise einen Entscheidungsbaum für eine Funktion $f : \{0,1\}^n \to \{0,1\}$ 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 Entscheidungsbäume; 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 16.

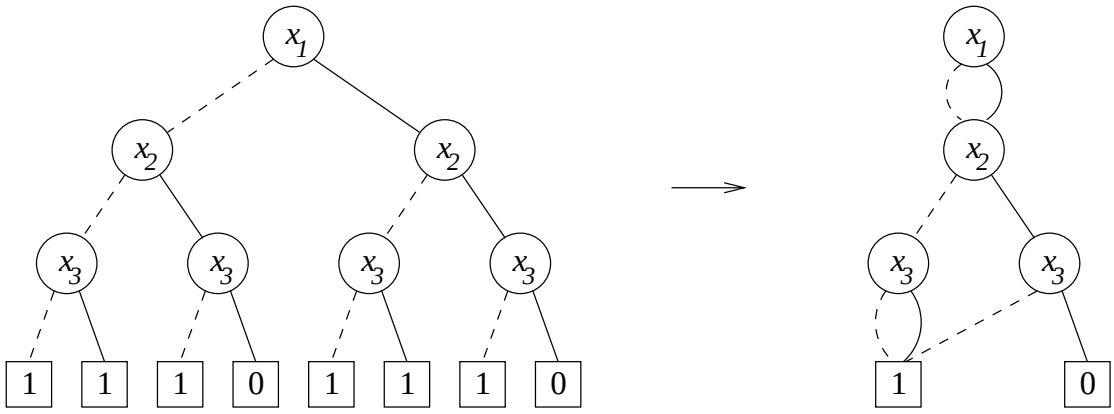


Abbildung 2: Ein Entscheidungsbaum und ein BDD für $\bar{x}_2 \vee \bar{x}_3$

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 wie vielen 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. (Formeln sind Schaltkreise, bei denen alle Bausteine einen Ausgangsgrad von 1 haben; diese Schaltkreise sind also baumförmig.) 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) als praktisch anwendbar erwiesen. Im Abschnitt 2 werden wir daher OBDDs definieren. Später werden wir Algorithmen für die wichtigsten Operationen auf booleschen Funktionen für OBDDs beschreiben.

Beziehungen zwischen BDDs und anderen nichtuniformen Berechnungsmodellen

Bekannte nichtuniforme Berechnungsmodelle sind Schaltkreise, Formeln und nichtuniforme Turingmaschinen. Es ist klar, dass eine Funktion $f = (f_n)$, $f_n : \{0,1\}^n \to \{0,1\}$, 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 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 mit Speicherplatz $s(n)$ durch ein BDD. Wir betrachten nur den Fall $s(n) \geq \log n$. Die Anzahl der Konfigurationen der Turingmaschine ist dann durch $2^{c \cdot s(n)}$ beschränkt, wobei c eine geeignete Konstante ist. Mit einem zusätzlichen Zähler und Speicherplatz $O(s(n))$ kann die Turingmaschine die Rechenschritte zählen und die Berechnung beenden und verwerfen, falls die Zahl der Schritte größer als $2^{c \cdot s(n)}$ ist. Wir können also davon ausgehen, dass es keine Endlosschleifen gibt.

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 Konfigurationenfolge der Turingmaschine, daher ist die Simulation korrekt. Die Größe des BDDs ist gleich der Anzahl der Konfigurationen der Turingmaschine, also $2^{O(s(n))}$. 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. Wir betrachten nur den Fall, dass das gegebene BDD mindestens die Größe n hat. 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 $O(\log \mathrm{BDD}_f(n))$, wobei $\mathrm{BDD}_f(n)$ 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 $\Omega(n^2/\log^2 n)$ (siehe Abschnitt 16). 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

$$\frac{1}{3}\mathrm{C}_f \leq \mathrm{BDD}_f \leq \mathrm{L}_f + 3.$$

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.

Überblick über den Inhalt der Vorlesung

Die Vorlesung besteht aus insgesamt vier Teilen. Der erste Teil beschäftigt sich mit OBDDs, der am weitesten verbreiteten Datenstruktur für boolesche Funktionen. Wir werden zunächst die Definition und einige wichtige Eigenschaften der OBDDs behandeln. Anschließend entwerfen wir Algorithmen für die wichtigen Operationen auf booleschen Funktionen, die durch OBDDs dargestellt sind. Da OBDDs für viele wichtige Funktionen zu groß werden, wurden zahlreiche Erweiterungen von OBDDs als Datenstruktur für boolesche Funktionen vorgeschlagen. Einige dieser Erweiterungen behandeln wir im zweiten Teil der Vorlesung. Im dritten Teil der Vorlesung erarbeiten wir Methoden zum Beweis von unteren Schranken für

die BDD-Größe. Diese unteren Schranken sind zunächst durch die Beziehung zwischen der BDD-Größe und dem Speicherbedarf von sequentiellen Berechnungen motiviert. Wie oben erwähnt, erhält man für BDDs ohne weitere Einschränkungen nur sehr schwache Schranken. Daher werden in der Forschung Methoden zum Beweis unterer Schranken an eingeschränkten BDD-Varianten studiert. Die wahrscheinlich wichtigste Methode zum Beweis unterer Schranken ist die Kommunikationskomplexität. Im vierten Teil der Vorlesung wollen wir Anwendungen von BDDs betrachten, die nicht durch den Hardwareentwurf motiviert sind. Wir werden sehen, wie große Graphen mit Hilfe von BDDs dargestellt werden können, und einen Algorithmus zur Flussmaximierung auf einem derart dargestellten Graphen vorstellen.

Literatur

Die Vorlesung folgt keinem Lehrbuch. Die Darstellung und Stoffauswahl ist am ehesten an die Monographie von Wegener (2000) angelehnt, die Vorlesung enthält aber auch Material, das nicht in diesem Buch enthalten ist. Das Buch von Wegener (2003) enthält neben Material zu BDDs als Berechnungsmodell auch Grundlagen zur Kommunikationskomplexität. Der Artikel von Drechsler und Sieling ist ein Überblicksartikel über BDDs, die Bücher von Hachtel und Somenzi und von Minato beschäftigen sich mit Schaltkreisentwurf und BDDs aus einer eher praktischen Sicht.

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

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

3. Meinel, Ch. und Theobald, T. (1998). Algorithms and Data Structures in VLSI-Design: OBDD – Foundations and Applications. Springer. Erhältlich unter http://eccc.hpi-web.de/eccc-local/ECCC-Books/Meinel-Theobald-book/obddbook.html

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

5. Long, D.E. (1998). The design of a cache-friendly BDD library. In Proc. of ICCAD, 639-645.

6. Somenzi, F. (2001). Efficient manipulation of decision diagrams.

7. Wegener, I. (2000). Branching programs and binary decision diagrams, theory and applications. SIAM Monographs on Discrete Mathematics and Applications. Int. Journal on Software Tools for Technology Transfer 3, 171–181.

8. Wegener, I. (2003). Komplexitätstheorie, Grenzen der Effizienz von Algorithmen. Springer.

2 Grundlagen zu OBDDs

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 (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 2.1: 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.

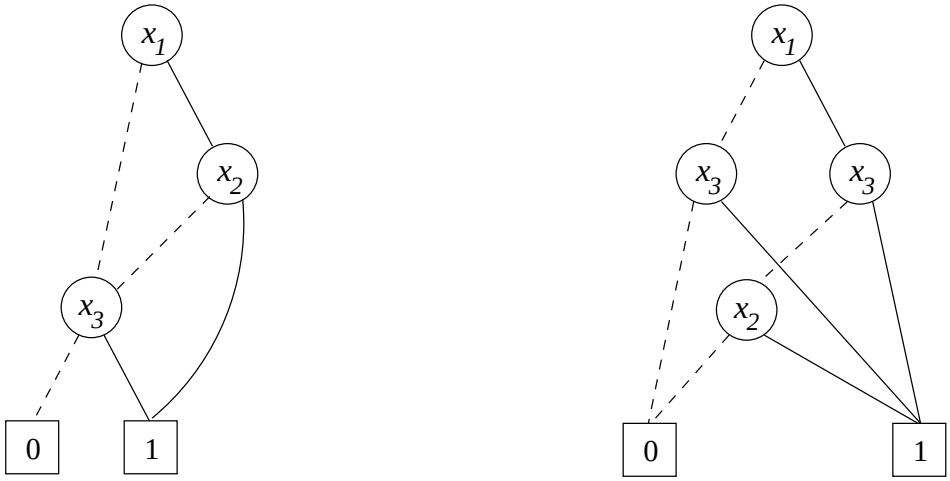


Abbildung 3: OBDDs für $f(x_1, x_2, x_3) = x_1x_2 \vee x_3$

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.

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. Der Zusatz „bis auf Isomorphie“ bedeutet hierbei, dass sich zwei OBDDs nur in den intern zur Darstellung verwendeten Knotennummern unterscheiden dürfen.

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* oder *Eliminationsregel*.

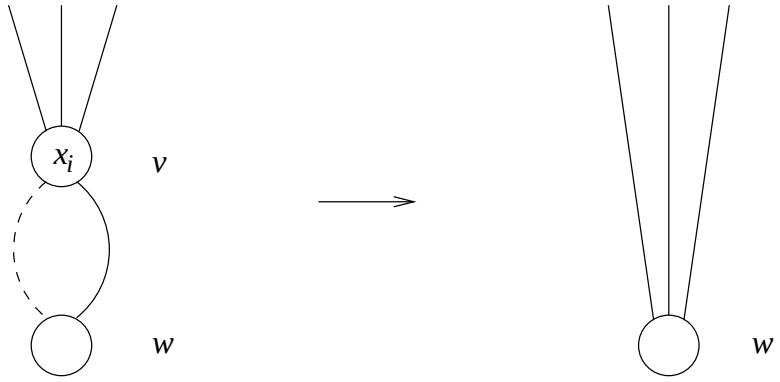


Abbildung 4: Deletion Rule

Die zweite Reduktionsregel, die *Merging Rule* oder *Verschmelzungsregel*, 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.

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

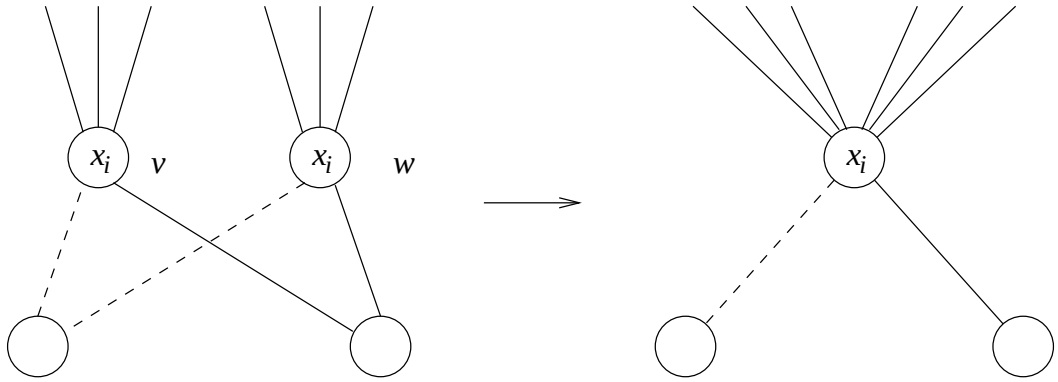


Abbildung 5: Merging Rule

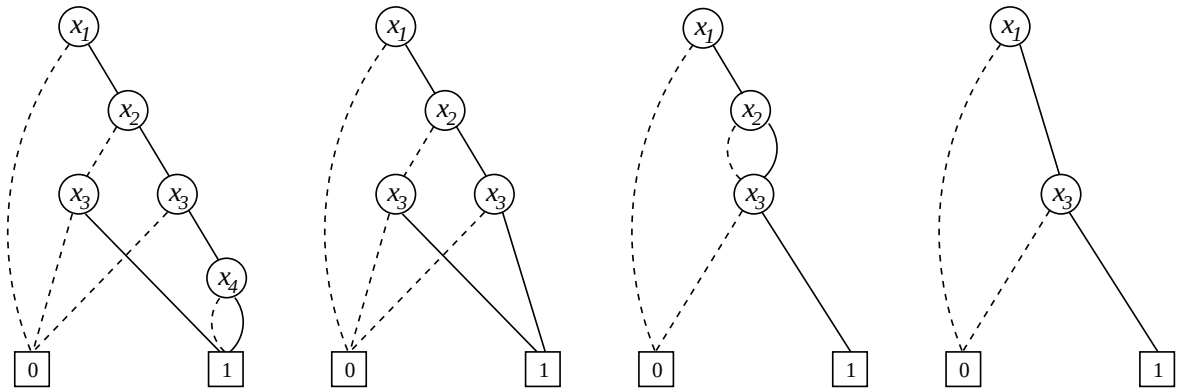


Abbildung 6: Beispiel für eine Reduktion

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.

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 $x_1, \ldots, x_n$. 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 $x_1, \ldots, x_{i-1}$ getestet werden. Der Pfad P wird durchlaufen, wenn die Variablen, die auf P getestet werden, passende Werte haben, wenn also $x_1 = c_1, \ldots, x_{i-1} = c_{i-1}$ für geeignete Konstanten $c_1, \ldots, c_{i-1}$ gilt. Am Knoten v und an Knoten unterhalb von v dürfen nur die Variablen $x_i, \ldots, x_n$ getestet werden. Daher wird an v die Funktion $f_{|x_1=c_1,\ldots,x_{i-1}=c_{i-1}}$ berechnet. Sollte es außer P noch einen Pfad Q geben, der von der Quelle zu v führt und der für $x_1 = d_1, \ldots, x_{i-1} = d_{i-1}$ durchlaufen wird, wird an v zugleich die Funktion $f_{|x_1=d_1,\ldots,x_{i-1}=d_{i-1}}$

berechnet. Da im OBDD an v und an Knoten unterhalb von v nur die Variablen $x_i, \ldots, x_n$ getestet werden dürfen, gilt $f_{|x_1=c_1,\ldots,x_{i-1}=c_{i-1}} = f_{|x_1=d_1,\ldots,x_{i-1}=d_{i-1}}$. An jedem mit x_i markierten Knoten wird also genau eine Subfunktion von f berechnet, die wir aus f erhalten können, indem wir $x_1, \ldots, x_{i-1}$ auf geeignete Weise konstantsetzen.

Wir können nun minimale OBDDs für jede boolesche Funktion und jede Variablenordnung beschreiben. Der folgende Satz ist zwar nur für die Variablenordnung $x_1, \ldots, x_n$ formuliert, gilt aber für alle Variablenordnungen, da man die Variablen umbenennen kann.

Satz 2.2: Sei S_i die Menge der Subfunktionen von f, die wir durch Konstantsetzen von $x_1, \ldots, x_{i-1}$ erhalten und die essentiell von x_i abhängen. Es gibt bis auf Isomorphie genau ein OBDD minimaler Größe für f und die Variablenordnung $x_1, \ldots, x_n$. Dieses OBDD enthält genau $|S_i|$ Knoten, die mit x_i markiert sind.

Beweis: Der Beweis dieses Struktursatzes wie auch spätere ähnliche Beweise bestehen aus den folgenden Schritten: (i) Konstruktion eines OBDDs mit den angegebenen Anzahlen von Knoten, (ii) Beweis, dass das konstruierte OBDD die Funktion f berechnet, (iii) Beweis, dass das OBDD minimale Größe hat, und (iv) Beweis, dass bei der Konstruktion des OBDDs keine Wahlmöglichkeiten waren.

Wir konstruieren nun 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 $f_{|x_1=c_1,\ldots,x_{i-1}=c_{i-1}}$ gibt, die essentiell von x_i abhängen, gibt es eine Zuweisung $c = (c_1, \ldots, c_{i-1})$ zu $x_1, \ldots, x_{i-1}$, für die $f_{|x_1=c_1,\ldots,x_{i-1}=c_{i-1}} \in S_i$ gilt, sodass 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 $f_{|x_1=c_1,\ldots,x_{i-1}=c_{i-1}}$ berechnet wird. Die ersten beiden Fälle führen zum Widerspruch, weil $f_{|x_1=c_1,\ldots,x_{i-1}=c_{i-1}}$ 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 $x_1, \ldots, x_n$ 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. □

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 so lange reduzieren, bis keine der Reduktionsregeln mehr anwendbar ist.

Lemma 2.3: Sei G ein OBDD für die Funktion f und die Variablenordnung π. Das OBDD 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 $x_1, \ldots, x_n$ ist.

Aus dem Beweis von Satz 2.2 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, sodass 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. □

Wir wollen jetzt zeigen, wie aus einem OBDD für die Funktion f und die Variablenordnung π, o.B.d.A. $x_1, \ldots, x_n$, das minimale OBDD berechnet werden kann. Es genügt zwar, so lange 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 2.3 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.

Zu jedem inneren Knoten v des OBDDs G gibt es als Attribute die 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 2.4:
Eingabe: Ein OBDD G für f mit der Ordnung $x_1, \ldots, x_n$.
Ausgabe: Das minimale OBDD G^* für f und die Ordnung $x_1, \ldots, x_n$.

- 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 ($i = 1, \ldots, n$).
- $id := 1$.
- 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 $v_2, \ldots, v_l$ durch die Zuweisungen

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

    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.

Die Rechenzeit von Algorithmus 2.4 wird durch das Sortieren dominiert. Daher ist es naheliegend, Bucketsort zu verwenden. Andererseits ist Bucketsort nicht direkt anwendbar, da es quadratisch viele mögliche Schlüssel gibt, aber lineare Rechenzeit und linearer Speicherplatz reichen sollen. Daher verwenden wir zwei Phasen. Der folgende Algorithmus realisiert kein Sortieren in dem Sinne, dass die Liste gemäß einer Ordnung umsortiert wird; es genügt uns, dass wir Knoten mit gleichem Paar von 0- und 1-Nachfolgern finden und verschmelzen können.

Wir verwenden als zusätzliche Datenstrukturen ein Array von linearen Listen (Buckets) $L[1, \ldots, |G|]$ (d.h., $L[i]$ ist eine lineare Liste, die Knotennummern aufnehmen kann), ein Array $R[1, \ldots, |G|]$, das Knotennummern aufnehmen kann, und eine Liste *L-ACTIVE*

Wir ersetzen in Algorithmus 2.4 die Schritte innerhalb der Schleife durch die folgenden Schritte.

- Anwendung der Deletion Rule wie in Algorithmus 2.4.
- Durchlaufe S_i. Für jedes $v \in S_i$ füge v in die Liste $L[v.null.id]$ ein. Falls v der erste Knoten in der Liste ist, füge zusätzlich $v.null.id$ in *L-ACTIVE* ein.

  Nur Knoten in derselben Liste haben denselben 0-Nachfolger, so dass nur solche Knoten verschmolzen werden können. Die Liste *L-ACTIVE* enthält die Indizes der nicht leeren Buckets in L.
- Für jedes $j \in$ *L-ACTIVE* führe die folgenden Schritte aus.
  - Für alle $v \in L[j]$:

    Falls $R[v.eins.id]$ leer ist, soll v im reduzierten OBDD enthalten sein. Also speichere v in $R[v.eins.id]$ und analog zu Algorithmus 2.4:

    $$id := id + 1; v.id := id; Z[v.id] := v;$$
    $$v.null := Z[v.null.id]; v.eins := Z[v.eins.id];$$

    Sonst wird v durch den Knoten w in $R[v.eins.id]$ repräsentiert, also

    $$v.id := w.id;$$
  - Um die Einträge in R zu löschen, durchlaufe $L[j]$ ein zweites Mal und lösche für jedes $v \in L[j]$ den Eintrag in $R[v.eins.id]$.
- Löschen von L und *L-ACTIVE*: Durchlaufe *L-ACTIVE* und lösche für jedes $j \in$ *L-ACTIVE* die Liste $L[j]$ sowie den Eintrag von j in *L-ACTIVE*.

Beim Löschen von L und R dürfen wir nicht einfach das gesamte Array durchlaufen, da eventuell viel weniger Einträge als $|G|$ zu löschen sind. Mit der Liste *L-ACTIVE* stellen wir sicher, dass wir für die Suche nach den r nicht leeren Buckets in R und das Löschen dieser Buckets mit Rechenzeit $O(r)$ auskommen. Analog hilft uns $L[i]$ beim Löschen von R. Da der Algorithmus die verwendeten Datenstrukturen selbst löscht, braucht diese für die nächste Iteration der Schleife in Algorithmus 2.4 nicht neu initialisiert zu werden. Da ansonsten der Aufwand pro Knoten konstant ist, hat der gesamte Algorithmus lineare Rechenzeit.

3 Die Abhängigkeit der OBDD-Größe von der Variablenordnung

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 $(v_0, \ldots, v_n)$ 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 $(v_0, \ldots, v_4)$. 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, sodass die Gesamtzahl der Knoten gleich $\sum_{i=1}^{n+1} i = O(n^2)$ ist.) Je nach Wertevektor sind noch Verschmelzungen von Knoten möglich, sodass 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.

Als nächstes Beispiel untersuchen wir die Funktion INDEX. Sei $n = 2^k$. Die Funktion INDEX_n hängt von den Variablen $x_0, \ldots, x_{n-1}$ und $a_0, \ldots, a_{k-1}$ ab. Die a-Variablen werden als Binärzahl $|a| := \sum_{i=0}^{k-1} a_i 2^i$ interpretiert, und der Funktionswert ist $x_{|a|}$.

Wir betrachten zunächst die Variablenordnung $a_0, \ldots, a_{k-1}, x_0, \ldots, x_{n-1}$. Dann gibt es ein OBDD linearer Größe für INDEX, das für $k = 3$ in Abbildung 8 gezeigt ist. Die a-Variablen sind in einem vollständigen Entscheidungsbaum angeordnet. Jedes Blatt wird nur für einen Wert $|a|$ erreicht, sodass dort einfach die Variable $x_{|a|}$ getestet werden kann. Man sieht auch leicht, dass man dieselbe OBDD-Größe für jede Variablenordnung erhält, bei der die a-Variablen vor den x-Variablen getestet werden.

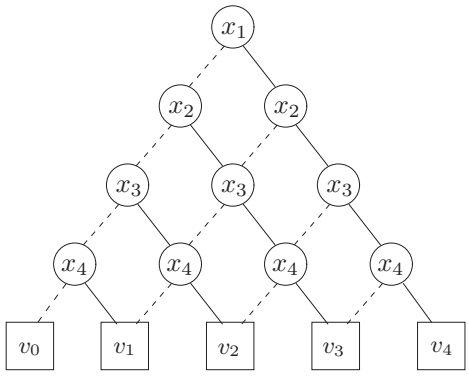


Abbildung 7: Ein OBDD für symmetrische Funktionen auf 4 Variablen

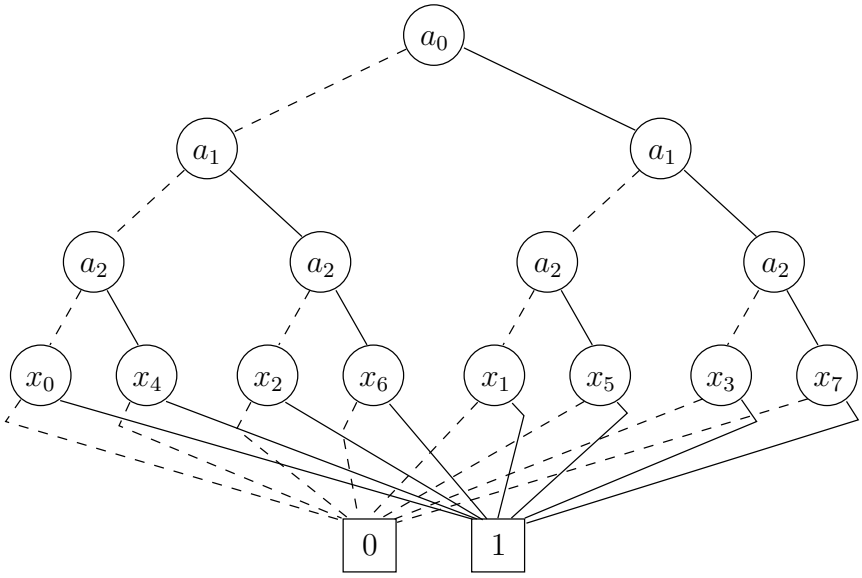


Abbildung 8: Ein OBDD für INDEX_8

Wir betrachten nun eine Variablenordnung, bei der l der x-Variablen zu Beginn getestet werden, und zeigen, dass das zugehörige OBDD mindestens $2^l - 2$ innere Knoten hat. Seien o.B.d.A. die zu Beginn getesteten Variablen die Variablen $x_0, \ldots, x_{l-1}$. Wir zeigen, dass die Subfunktionen von INDEX, die sich durch die 2^l verschiendenen Belegungen dieser Variablen ergeben, verschieden sind. Wenn wir zwei Belegungen betrachten, die sich in x_i unterscheiden, sind die resultierenden Subfunktionen verschieden, da wir die a-Variablen so belegen können, dass x_i ausgegeben werden muss. Also erhalten wir 2^l verschiedene Subfunktionen, von denen alle bis auf höchstens zwei nicht konstant sind, also von mindestens einer a-Variablen abhängen. (Die beiden konstanten Subfunktionen erhalten wir nur für $l = n$, wenn alle x-Variablen den Wert 0 bzw. 1 haben.) Alle diese Subfunktionen müssen an verschiedenen inneren Knoten dargestellt werden, wir erhalten also die untere Schranke $2^l - 2$. Insbesondere für $l = n$ erhalten wir die exponentielle untere Schranke $2^n - 2$ für die Anzahl der mit a-Variablen markierten Knoten.

Wir wollen nun untersuchen, für welchen Anteil an allen Variablenordnungen die OBDD-Größe für INDEX exponentiell ist; es könnte ja sein, dass der Fall exponentieller OBDD-Größe nur bei „wenigen“ Variablenordnungen auftritt. Dazu wählen wir zufällig gemäß Gleichverteilung eine Variablenordnung und schätzen die Wahrscheinlichkeit für exponentielle OBDD-Größe nach unten ab. Dies ist dann eine untere Schranke für den Anteil der Variablenordnungen mit exponentieller OBDD-Größe. Wir wählen $l = \lfloor \sqrt{n} \rfloor$. Wie groß ist die Wahrscheinlichkeit, dass bei zufälliger Wahl der Variablenordnung unter den ersten l Variablen nur x-Variablen sind? Bei der Wahl der ersten der l Positionen gibt es n x-Variablen und $k = \log n$ a-Variablen, also beträgt die Wahrscheinlichkeit für die Wahl einer x-Variablen $1 - k/(n + k)$. Falls für die erste Position eine x-Variable gewählt wurde, ist danach die Zahl der x-Variablen um Eins kleiner, die Wahrscheinlichkeit für die Wahl einer x-Variablen beträgt dann $1 - k/(n + k - 1)$, usw. Also erhalten wir für die Wahrscheinlichkeit, dass die

ersten l Variablen nur x-Variablen sind

$$\prod_{j=1}^{l}\left(1-\frac{k}{n+k-j+1}\right) \geq \left(1-\frac{k}{n+k-l}\right)^{l} \geq 1-\frac{lk}{n+k-l} = 1-O(\log n/\sqrt{n}).$$

Bei der letzten Abschätzung haben wir $(1-x)^i \geq 1-ix$ (für $0 \leq x \leq 1$) benutzt. Wir sehen, dass der Anteil der Variablenordnungen mit exponentieller OBDD-Größe $2^{\Omega(n^{1/2})}$ für n gegen unendlich gegen 1 konvergiert. INDEX ist also eine Funktion, bei der wir die Variablenordnung nicht zufällig wählen sollten, da dies mit hoher Wahrscheinlichkeit zu großen OBDDs führt.

Das Ergebnis über INDEX motiviert die folgende Definition. Dabei bezeichnen wir eine Funktion $g(n)$ als superpolynomiell, wenn für alle $c > 0$ gilt, dass $g(n) = \Omega(n^c)$. (Mit diesem Begriff werden auch Ausdrücke wie z.B. $n^{\log n}$ erfasst, die schneller als polynomiell, aber langsamer als exponentiell wachsen.)

Definition 3.1:

1. Eine Funktion $f = (f_n)$ heißt *nice* oder *gutartig*, falls alle Variablenordnungen zu polynomieller OBDD-Größe führen.

2. Eine Funktion $f = (f_n)$ heißt *ugly* oder *bösartig*, falls alle Variablenordnungen zu superpolynomieller OBDD-Größe führen.

3. Eine Funktion $f = (f_n)$ heißt *almost nice* oder *fast gutartig*, falls es eine Variablenordnung mit superpolynomieller OBDD-Größe gibt und der Anteil der Variablenordnungen mit polynomieller OBDD-Größe für n gegen unendlich gegen 1 konvergiert.

4. Eine Funktion $f = (f_n)$ heißt *almost ugly* oder *fast bösartig*, falls es eine Variablenordnung mit polynomieller OBDD-Größe gibt und der Anteil der Variablenordnungen mit superpolynomieller OBDD-Größe für n gegen unendlich gegen 1 konvergiert.

5. Eine Funktion $f = (f_n)$ heißt *ambiguous*, falls es ein Polynom $p(n)$ und eine superpolynomiell wachsende Funktion $q(n)$, sowie ein $\varepsilon > 0$ gibt, sodass sowohl der Anteil der Variablenordnungen mit OBDD-Größe höchstens $p(n)$, als auch der Anteil der Variablenordnungen mit OBDD-Größe mindestens $q(n)$ für hinreichend großes n mindestens ε beträgt.

Also sind alle symmetrischen Funktionen *nice*, während INDEX *almost ugly* ist. Bis jetzt ist noch keine Funktion bekannt, die *ambiguous* ist. Für *ugly* und *almost nice* erarbeiten wir im Folgenden Beispiele.

Die sogenannte *indirekte Adressierungsfunktion* ISA ist für $n = 2^k$ auf $n+k$ Variablen $x_0, \ldots, x_{n-1}, y_0, \ldots, y_{k-1}$ definiert. Sei $s \in \{0, \ldots, n-1\}$ der Wert der y-Variablen als Binärzahl interpretiert, und sei $b = \lfloor n/k \rfloor$. Falls $s \geq b$, ist die Ausgabe 0. Anderenfalls werden die x-Variablen in b Blöcke der Länge k zerlegt (nummeriert von 0 bis $b-1$). Sei $t \in \{0, \ldots, n-1\}$ der Wert des s-ten Blocks als Binärzahl interpretiert. Dann ist die Ausgabe x_t.

Wir wollen nun eine exponentielle untere Schranke für die Funktion ISA beweisen. Dazu gehen wir davon aus, dass ein OBDD G für ISA_n gegeben ist. In der Variablenordnung von G suchen wir die erste Stelle, an der $b-1$ x-Variablen getestet worden sind. Dann gibt es einen Block von x-Variablen, der komplett hinter dieser Position gelesen wird, da es b Blöcke gibt. Sei a die Nummer dieses Blocks. Wir belegen nun die y-Variablen so, dass $s = a$ gilt. Auf diese Weise entsteht eine Subfunktion, die darin besteht, dass die Variable ausgegeben wird, deren Adresse der Wert des gewählten Blocks ist. Wir beachten, dass dies nicht die Funktion INDEX ist, da auch Variablen aus dem gewählten Block ausgegeben werden dürfen. Dennoch führt dieselbe Argumentation wie bei INDEX zu der unteren Schranke $2^{\Omega(b)} = 2^{\Omega(n/\log n)}$. Da wir keine Annahme über die Variablenordnung gemacht haben, gilt diese untere Schranke für alle Variablenordnungen, d.h., wir haben bewiesen, dass ISA *ugly* ist.

Das einzige bekannte Beispiel für eine Funktion mit der Eigenschaft *almost nice* ist die Funktion *common knowledge direct storage access* CKDSA, die ebenfalls eine Variante der INDEX-Funktion ist. Sei wieder $n = 2^k$. Die Funktion ist auf n^2k+n Variablen definiert, die als n^2 Blöcke der Länge k sowie als ein weiterer Block der Länge n interpretiert werden. Die n^2 Blöcke werden wieder als Adressen aus $\{0,\ldots,n-1\}$ aufgefasst, der Block der Länge n als Datenblock. Falls alle diese Adressen denselben Wert haben, wird das adressierte Bit im Datenblock ausgegeben, anderenfalls 0.

Mit ähnlichen Argumenten wie bei INDEX erhält man die untere Schranke $2^{\Omega(n)}$, wenn der Datenblock vor allen Adressbits getestet wird. Es bleibt also zu zeigen, dass der Anteil der Variablenordnungen mit polynomieller OBDD-Größe gegen 1 konvergiert.

Wir zeigen zunächst, dass die OBDD-Größe durch $O(n^3 \log n)$ abgeschätzt werden kann, wenn in der Variablenordnung für jedes Bit der Adresse mindestens eine der zugehörigen n^2 Bits vor dem ersten Datenbit gelesen wird. Das OBDD hat das folgende Aussehen: Wenn zum ersten Mal das i-te Bit einer Adresse gelesen wird, wird dieser Wert „gespeichert“, d.h., in dieser Ebene verdoppelt sich die Breite des OBDDs, wobei die erste Hälfte der Knoten für den Wert 0 und zweite Hälfte für den Wert 1 des betrachteten Bits erreicht wird. Wenn das i-te Bit einer Adresse vorher schon gelesen wurde, genügt es, den Wert mit dem vorher gelesenen zu vergleichen und bei Ungleichheit zur 0-Senke zu gehen. Also wird das OBDD nicht breiter. Da sich die Breite nur $\log n$ Mal verdoppelt, kommen wir auf lineare Breite. Nachdem die Adresse bekannt ist, genügt es, in ähnlicher Weise die übrigen Adressbits auf den richtigen Wert und das adressierte Datenbit auf 1 zu testen. Da die Breite linear und die Anzahl der Variablen $O(n^2 \log n)$ ist, ergibt sich die behauptete obere Schranke für die Größe.

Es genügt nun zu zeigen, dass bei zufälliger Wahl der Variablenordnung diese die o.g. Eigenschaft mit großer Wahrscheinlichkeit hat. Von jedem Adressbit gibt es n^2 „Kopien“. Wir betrachten nun die zufällige Anordnung dieser n^2 Adressbits und der n Datenbits auf n^2+n Positionen. Uns interessiert die Wahrscheinlichkeit, dass an die erste Position eines der n Datenbits kommt. Da wir hierbei insgesamt n^2+n Bits betrachten, beträgt diese Wahrscheinlichkeit $n/(n^2+n) = 1/(n+1) \leq 1/n$. Damit beträgt die Wahrscheinlichkeit, dass für eine der $\log n$ Adresspositionen alle n^2 zugehörigen Bits hinter dem ersten Datenbit angeordnet werden, höchstens $(\log n)/n$, und die Wahrscheinlichkeit, bei zufälliger Wahl der Varia-

blenordnung polynomielle OBDD-Größe zu bekommen, beträgt mindestens $1 - (\log n)/n$, konvergiert also gegen 1.

4 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. Wir wollen im Folgenden drei verschiedene Varianten des Variablenordnungsproblems untersuchen.

Das Variablenordnungsproblem bei gegebenem Schaltkreis

Bei der ersten Variante 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-schweres 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. Es ist nicht schwer, Beispiele zu finden, bei denen diese Heuristik zu einer optimalen Variablenordnung führt, und andere Beispiele, bei denen diese Heuristik zu einer schlechten Variablenordnung führt.

Das Variablenordnungsproblem bei gegebener Wertetabelle

Bei der zweiten Variante des Variablenordnungsproblems ist die Funktion $f \in B_n$ durch ihre Wertetabelle gegeben. Die Aufgabe besteht darin, eine optimale Variablenordnung für f zu berechnen. Der im Folgenden vorgestellte Algorithmus basiert auf dynamischer Programmierung und stammt von Friedman und Supowit (1990). Der Struktursatz 2.2 für OBDDs wird in dem Algorithmus auf die folgende Weise ausgenutzt. Sei $I \subseteq \{1, \ldots, n\}$ und sei ein OBDD für f gegeben, in dem die Variablen mit Indizes aus I nach den Variablen mit Indizes in $\bar{I} := \{1, \ldots, n\} - I$ getestet werden. Sei i der Index der ersten I-Variablen. Dann hängt nach dem Struktursatz die Anzahl der x_i-Knoten nicht von der relativen Ordnung der übrigen I-Variablen und auch nicht von der relativen Ordnung der $\bar{I}$-Variablen ab.

Der folgende Algorithmus berechnet optimale Variablenordnungen für Teil-OBDDs G_I, wobei in diesem OBDD alle Subfunktionen dargestellt werden, die durch Konstantsetzen der Variablen in $\bar{I}$ entstehen. D.h., in G_I kommen nur Tests von I-Variablen vor und G_I bildet den „unteren Teil“ jedes OBDDs, in dem die $\bar{I}$-Variablen vor den I-Variablen getestet werden. Mit $\min(I)$ wird die minimale Größe von G_I bezeichnet (wobei das Minimum über alle Ordnungen der Variablen in I gebildet wird); π_I bezeichnet eine Variablenordnung der I-Variablen, für die das Minimum angenommen wird. Weiterhin speichert der Algorithmus Tabellen table(I), die für jede Belegung der Variablen in $\bar{I}$ einen Zeiger auf den Knoten in G_I enthält, an dem die jeweilige Subfunktion dargestellt wird.

Die verwendeten Datenstrukturen sind leicht für $I = \emptyset$ zu initialisieren; table$(\emptyset)$ ist die Wertetabelle der Funktion (wobei die Einträge 0 und 1 als Zeiger auf die jeweiligen Senken interpretiert werden), $\pi_\emptyset$ eine leere Permutation und $\min(\emptyset) = 0$. Der Algorithmus berechnet

aus den jeweils vorhandenen Informationen für Indexmengen der Größe $t-1$ die Informationen für Indexmengen der Größe t. Der folgende Algorithmus wird also nacheinander für $t = 1, \ldots, n$ ausgeführt. Eine optimale Variablenordnung ist dann $\pi_{\{1,\ldots,n\}}$.

for all $I^* \subseteq \{1, \ldots, n\}$ mit $|I^*| = t$
  for all $i \in I^*$
    $I := I^* - \{i\}$
    Erzeuge für alle Belegungen der Variablen in $\overline{I^*}$ einen x_i-Knoten und berechne mit Hilfe von table(I) die Nachfolger dieser Knoten.
    Wende die Reduktionsregeln auf die erzeugten Knoten an.
    $s(i) :=$ Anzahl der x_i-Knoten
    $s^*(i) := s(i) + \min(I)$
  $i^* := \text{argmin}_i\{s^*(i)\}$
  $\pi_{I^*} :=$ Ordnung beginnend mit i^*, gefolgt von $\pi_{I^*-\{i^*\}}$
  $\min(I^*) := s^*(i^*)$
  table(I^*) kann von den x_{i^*}-Knoten nach Anwendung der Reduktionsregeln abgelesen werden.

In der inneren Schleife werden alle Variablen aus I^* probeweise an die erste Position unter den I^*-Variablen geschoben und für die resultierende Variablenordnung wird die Anzahl der mit I^*-Variablen markierten Knoten bestimmt. Unter allen Möglichkeiten wird eine gewählt, die zu minimaler Größe führt. Die Anzahl der Knoten wird bestimmt, indem probeweise die x_i-Knoten mit Zeigern auf die Nachfolger erzeugt werden, die Reduktionsregeln angewendet werden, die Knoten gezählt werden und anschließend $\min(I)$ als Anzahl der Knoten in den darunterliegenden Ebenen addiert wird. Die Wahl des i, das zu minimaler Größe von G_{I^*} führt, wird dadurch realisiert, dass i in π_{I^*} an erster Position steht und die Parameter $\min(I^*)$ und table(I^*) für diese Variablenordnung gewählt werden.

Das Array table(I^*) enthält $2^{n-|I^*|}$ Einträge. In jeder Iteration der inneren Schleife werden $2^{n-|I^*|}$ Knoten probeweise erzeugt. Für die Reduktion kann der Linearzeitalgorithmus verwendet werden. Also kann eine Iteration der inneren Schleife in Zeit $O(2^{n-t})$ ausgeführt werden. Die Anzahl der Iterationen der inneren Schleife beträgt t, die der äußeren Schleife $\binom{n}{t}$. Also beträgt die Rechenzeit größenordnungsmäßig

$$\begin{aligned}\sum_{t=1}^{n}\binom{n}{t}t2^{n-t} &= n\sum_{t=1}^{n}\binom{n-1}{t-1}2^{n-t} \\ &= n\sum_{t=0}^{n-1}\binom{n-1}{t}1^t2^{n-t-1} = n3^{n-1}.\end{aligned}$$

Die erste Gleichung ergibt sich durch Rechnen mit Binomialkoeffizienten, bei der zweiten wird der Index der Summe verschoben und bei der dritten der binomische Lehrsatz angewendet.

Bei der Berechnung des Speicherbedarfs für die table-Datenstruktur (der Rest ist größenordnungsmäßig vernachlässigbar) beachten wir, dass wir bei der Bearbeitung von Indexmengen der Größe t nur auf Informationen über Indexmengen der Größe $t-1$ zurückgreifen, sodass die Informationen über die kleineren Indexmengen gelöscht werden können. Daher

genügt es, den Speicherbedarf für zwei aufeinanderfolgende Werte $t-1$ und t zu bestimmen, der sich größenordnungsmäßig durch das Maximum über den Speicherbedarf für alle t abschätzen lässt. Da $\binom{n}{t}$ Indexmengen betrachtet werden und der Speicherbedarf jeweils 2^{n-t} beträgt, genügt es, das Maximum für alle t von $s(t) = \binom{n}{t}2^{n-t}$ zu bestimmen. Durch einfaches Rechnen mit Binomialkoeffizienten erhalten wir

$$s(n) \leq s(n-1) \leq \cdots \leq s(\lfloor n/3 \rfloor) \geq s(\lfloor n/3 \rfloor - 1) \geq \cdots \geq s(1),$$

also ist das gesuchte Maximum gleich $s(\lfloor n/3 \rfloor)$. Dieses schätzen wir mit der Stirling-Formel $n! = (n/e)^n\sqrt{2\pi n}(1+o(1))$ und erhalten (für o.B.d.A. durch 3 teilbares n):

$$s(n/3) = \binom{n}{n/3}2^{2n/3} = \Theta\left(\frac{(n/e)^n\sqrt{2\pi n}}{(n/3e)^{n/3}(2n/3e)^{2n/3}\sqrt{4\pi^2 n^2}} \cdot 2^{2n/3}\right) = \Theta\left(3^n/\sqrt{n}\right).$$

Rechenzeit und Speicherplatz sind polynomiell in Bezug auf die Eingabelänge $N = 2^n$. Dies gilt natürlich nicht mehr, wenn die Eingabe in einer kompakteren Form, z.B. als OBDD, gegeben ist.

Das Variablenordnungsproblem bei gegebenem OBDD

Bei der dritten 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-schwer 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 Nichtapproximierbarkeitsergebnisses 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-schwer ist. Wir betrachten also das Problem

MinSBDD
Eingabe: Ein SBDD für $f_1, \ldots, f_m$.
Ausgabe: Ein SBDD minimaler Größe für $f_1, \ldots, f_m$.

Satz 4.1: 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., $U = \{u_1, \ldots, u_n\}$ ist eine Menge von Variablen und $C = \{C_1, \ldots, C_m\}$ 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 $f_1, \ldots, f_n, g_1, \ldots, g_m$ dar, die auf den Variablen $x_1, \ldots, x_n, y_1, \ldots, y_n$ 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 $f_1, \ldots, f_n, g_1, \ldots, g_m$.

Lemma 4.2: Falls (U, C) erfüllbar ist, gibt es ein SBDD für $f_1, \ldots, f_n, g_1, \ldots, g_m$ mit höchstens $2n + 2m$ inneren Knoten.

Lemma 4.3: Falls (U, C) nicht erfüllbar ist, hat jedes SBDD für $f_1, \ldots, f_n, g_1, \ldots, g_m$ 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 $f_1, \ldots, f_n, g_1, \ldots, g_m$ 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 $f_1, \ldots, f_n, g_1, \ldots, g_m$. 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-schwer ist, folgt die Behauptung.

Beweis von Lemma 4.2: Sei (U, C) erfüllbar und sei σ eine erfüllende Belegung. 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 $f_1, \ldots, f_n, g_1, \ldots, g_m$ 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 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 $2n + 2m$. □

Beweis von Lemma 4.3: Wir nehmen an, dass es ein SBDD für $f_1, \ldots, f_n, g_1, \ldots, g_m$ 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 $f_1, \ldots, f_n$ 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 im OBDD von jeder Funktion g_j einen Knoten geben, der mit einem Knoten aus dem 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 j-te Klausel von σ erfüllt wird. Da dies für alle Klauseln gilt, ist die oben konstruierte Variablenbelegung σ erfüllend. □

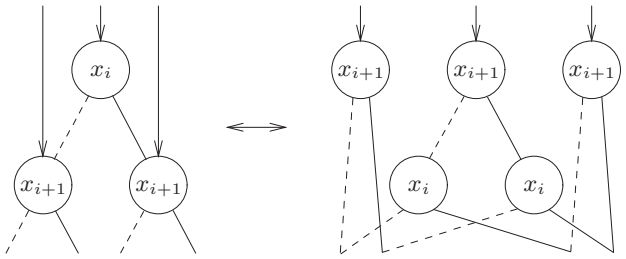


Abbildung 9: Realisierung der Operation SWAP(x_i, x_{i+1})

Die Operation SWAP, lokale Suche und Sifting

Wir diskutieren nun eine einfache elementare Operation zur Veränderung der Variablenordnung und darauf basierende Heuristiken. Komplexere Operationen zur Veränderung der Variablenordnung werden wir später behandeln.

Die einfachste Operation zum Verändern der Variablenordnung von OBDDs ist das Vertauschen benachbarter Variablen, was auch als Swap bezeichnet wird: Sei $x_1, \ldots, x_n$ die Variablenordnung des gegebenen OBDDs. SWAP(x_i, x_{i+1}) verändert die Variablenordnung in $x_1, \ldots, x_{i-1}, x_{i+1}, x_i, x_{i+2}, \ldots, x_n$. Es folgt direkt aus Satz 2.2, dass sich nur die Ebenen mit den Variablen x_i und x_{i+1} verändern. In Abbildung 9 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 so lange 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.

5 Das Variablenordnungsproblem für partiell symmetrische Funktionen

Wir wollen das Variablenordnungsproblem für eine spezielle Klasse von Funktionen, die *partiell symmetrischen Funktionen* untersuchen. Sei $f \in B_n$ über den Variablen $x_1, \ldots, x_n$ definiert. Wir definieren die Relation $\sim$ auf der Variablenmenge durch

$$x_i \sim x_j \Leftrightarrow f_{|x_i=0,x_j=1} = f_{|x_i=1,x_j=0} \text{ für } x_i \neq x_j,$$

und $x_i \sim x_i$ für alle x_i. Es ist leicht zu zeigen, dass $\sim$ eine Äquivalenzrelation ist. Seien $V_1, \ldots, V_k$ die Äquivalenzklassen von $\sim$. Aus der Definition von $\sim$ folgt, dass der Funktionswert von f nur von der Anzahl der Einsen in jeder Äquivalenzklasse abhängt, nicht aber von der Position der Einsen: Wenn wir zwei Belegungen der Variablen in V_i mit jeweils l Einsen und $|V_i| - l$ Nullen betrachten, entstehen aus diesen Belegungen dieselben Subfunktionen von f, da wir aufgrund der Definition von $\sim$ die Einsen beliebig unter den Variablen in V_i „verschieben“ dürfen.

Symmetrische Funktionen sind offensichtlich der Spezialfall von Funktionen, bei denen $\sim$ nur eine Äquivalenzklasse hat. Wir bezeichnen daher die Mengen $V_1, \ldots, V_k$ auch als Symmetriemengen. Symmetrische Funktionen können als Wertevektor dargestellt werden; analog können Funktionen mit den Symmetriemengen $V_1, \ldots, V_k$ durch eine k-dimensionale Matrix W der Größe $(|V_1|+1) \times \cdots \times (|V_k|+1)$ dargestellt werden; der Eintrag $W(i_1, \ldots, i_k)$ (mit $i_j \in \{0, \ldots, |V_j|\}$) gibt den Funktionswert für alle Eingaben mit i_j Einsen unter den Variablen in V_j an. W bezeichnen wir als Wertematrix.

Ein Beispiel für eine partiell symmetrische Funktion mit den zwei Symmetriemengen $\{x_1, \ldots, x_4\}$ und $\{y_1, \ldots, y_4\}$ ist durch die folgende Wertematrix gegeben. Die Interpretation der Tabelle sollte klar sein: Z.B. bedeutet der eingerahmte Wert, dass alle Eingaben, bei denen zwei x- und zwei y-Variablen gleich 1 sind, den Funktionswert 1 ergeben. In den weiteren Abbildungen von Wertematrizen lassen wir die Beschriftung der Zeilen und der Spalten weg.

		Anzahl der Einsen unter den y-Variablen				
		0	1	2	3	4
Anzahl der	0	1	1	0	1	0
Einsen	1	1	1	1	0	1
unter	2	1	1	**1**	1	0
den x-	3	1	1	1	1	1
Variablen	4	0	0	0	0	0

Jede Funktion ist diesem Sinne eine partiell symmetrische Funktion, da es n einelementige Symmetriemengen geben kann. Bei der folgenden Darstellung beschränken wir uns meist auf den Fall von zwei Symmetriemengen, da dann die Wertematrix zweidimensional und damit einfach darstellbar ist. Alle Betrachtungen lassen sich aber auch auf eine größere Anzahl von Symmetriemengen verallgemeinern; insbesondere der Algorithmus von Friedman

und Supowit lässt sich darüber auf eine neue Weise verstehen. Wir benutzen die partiell symmetrischen Funktionen mit zwei Symmetriemengen auch für die Untersuchung der Heuristik für das Variablenordnungsproblem, die darin besteht, zusammengehörende Variablen zusammen anzuordnen.

Wie können wir nun das Variablenordnungsproblem für partiell symmetrische Funktionen lösen? Wie kann die OBDD-Größe für eine spezielle Variablenordnung effizient berechnet werden? Um diese Fragen zu lösen, wenden wir den Struktursatz für OBDDs an.

Zur Vereinfachung der Darstellung betrachten wir nur den Fall, dass es die zwei Symmetriemengen $\{x_1, \ldots, x_n\}$ und $\{y_1, \ldots, y_m\}$ gibt. Die Wertematrix hat also die Größe $(n+1) \times (m+1)$. Wie realisieren wir das Ersetzen einer x-Variablen durch den Wert 0? Offensichtlich ist dann die Anzahl der Einsen unter den x-Variablen aus dem Bereich $0, \ldots, n-1$. D.h., die letzte Zeile der Wertematrix ist nicht mehr erreichbar, sodass wir sie streichen können. Auf diese Weise erhalten wir die Wertematrix für die Subfunktionen, bei denen eine x-Variable durch 0 ersetzt wurde; welche der x-Variablen dies ist, spielt aufgrund der partiellen Symmetrie keine Rolle. Analog können wir beim Ersetzen einer x-Variablen durch 1 die erste Zeile streichen. Bei den y-Variablen ist analog die erste bzw. letzte Spalte zu streichen. Wenn wir mehrere Variablen konstantsetzen, werden wiederum jeweils „erste“ oder „letzte“ Zeilen oder Spalten der verbliebenen Wertematrix gestrichen, sodass die Subfunktionen zusammenhängenden Teilmatrizen der ursprünglichen Wertematrix entsprechen. Wir erhalten also alle Subfunktionen von f, die durch Konstantsetzen von i x-Variablen und j y-Variablen entstehen, indem wir zusammenhängende Submatrizen der Wertematrix mit der Größe $(n+1-i) \times (m+1-j)$ betrachten.

Für $i = 2$ und $j = 3$ erhalten wir aus der Wertematrix oben die folgenden zusammenhängenden Wertematrizen bzw. Subfunktionen. Dabei haben wir mehrfach auftretende Wertematrizen/Subfunktionen nur einmal aufgeführt.

$$\begin{array}{cc} 1 & 1 \\ 1 & 1 \\ 1 & 1 \end{array} \qquad \begin{array}{cc} 1 & 0 \\ 1 & 1 \\ 1 & 1 \end{array} \qquad \begin{array}{cc} 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{array} \qquad \begin{array}{cc} 1 & 0 \\ 0 & 1 \\ 1 & 0 \end{array} \qquad \begin{array}{cc} 1 & 1 \\ 1 & 1 \\ 0 & 0 \end{array} \qquad \begin{array}{cc} 1 & 0 \\ 1 & 1 \\ 0 & 0 \end{array}$$

Die erste Matrix stellt offensichtlich die 1-Funktion dar. Die vorletzte Matrix stellt die Funktion dar, die den Wert 0 annimmt, wenn alle zwei verbliebenen x-Variablen den Wert 1 haben. Insbesondere hängt diese Funktion nicht von der letzten y-Variablen ab. Dies sehen wir daran, dass alle Zeilen konstant sind. Analog hängt eine durch eine Wertematrix dargestellte Funktion nicht von den x-Variablen ab, wenn alle Spalten konstant sind.

Wir fassen zusammen: Wir können Subfunktionen zählen, indem wir verschiedene Wertematrizen zählen. Wenn alle Zeilen einer Wertematrix konstant sind, hängt die Subfunktion nicht von den y-Variablen ab; wenn alle Spalten konstant sind, hängt sie nicht von den x-Variablen ab. Wir erhalten also die folgende Variante des Struktursatzes.

Satz 5.1: Sei f eine partiell symmetrische Funktion mit den Symmetriemengen V_1 und V_2 und der Wertematrix W_f. Sei eine Variablenordnung gegeben. Sei i die Anzahl der Variablen aus V_1 und j die Anzahl der Variablen aus V_2, die vor einer Variablen $x^* \in V_1$ getestet werden. Sei $T_1(r, s)$ die Anzahl der verschiedenen zusammenhängenden $r \times s$-Submatrizen von W_f, in denen nicht alle Spalten konstant sind. Dann enthält das reduzierte OBDD für f genau $T_1(|V_1| + 1 - i, |V_2| + 1 - j)$ Knoten, die mit x^* markiert sind.

Eine analoge Aussage gilt für die Variablen $y^* \in V_2$, wenn wir die Anzahl der zusammenhängenden Submatrizen zählen, bei denen nicht alle Zeilen konstant sind. Die Anzahl der zusammenhängenden $r \times s$-Submatrizen, in denen nicht alle Zeilen konstant sind, bezeichnen wir mit $T_2(r, s)$. Wir sehen also, dass für das Beispiel oben $T_1(3, 2) = 5$ und $T_2(3, 2) = 4$ gilt. In den beiden folgenden Tabellen sind alle T-Werte für das obige Beispiel aufgeführt.

$T_1(r,s)$	$s=$				
	5	4	3	2	1
$r=5$	1	2	3	4	4
4	2	4	6	6	6
3	3	5	5	5	4
2	4	4	4	4	2

$T_2(r,s)$	$s=$			
	5	4	3	2
$r=5$	1	2	3	3
4	2	4	5	5
3	3	4	4	4
2	3	3	3	3
1	3	3	3	2

In der ersten Tabelle fehlt die Zeile für $r = 1$, da natürlich in allen Matrizen mit einer Zeile alle Spalten konstant sind; analog fehlt in der zweiten Tabelle die Spalte mit $s = 1$.

Was haben wir bis jetzt erreicht? Wir können zu jeder Variablenordnung die Anzahl der Knoten auf jeder Ebene des reduzierten OBDDs bestimmen. Wenn z.B. auf Ebene $i + j + 1$ eine x-Variable getestet wird und zuvor i x-Variablen und j y-Variablen getestet wurden, beträgt die Anzahl der Knoten auf Ebene $i+j+1$ genau $T_1(|V_1|+1-i, |V_2|+1-j)$. Wir wollen nun alle Informationen in einem Hilfsgraphen darstellen, sodass wir die OBDD-Größe als Pfadlänge erhalten. Der Hilfsgraph ist ein gerichteter Gittergraph. Die Knotenmenge ist

$$U = \{(i, j) \mid i \in \{1, \ldots, |V_1| + 1\}, j \in \{1, \ldots, |V_2| + 1\}\};$$

die Kantenmenge ist

$$E = E_1 \cup E_2$$

mit

$$\begin{aligned} E_1 &= \{((i, j), (i - 1, j)) \mid i \in \{2, \ldots, |V_1| + 1, j \in \{1, \ldots, |V_2| + 1\}\}, \\ E_2 &= \{((i, j), (i, j - 1)) \mid i \in \{1, \ldots, |V_1| + 1, j \in \{2, \ldots, |V_2| + 1\}\}. \end{aligned}$$

Die Quelle des Graphen ist also $(|V_1| + 1, |V_2| + 1)$, die Senke ist $(1, 1)$. Die Kanten $((i, j), (i - 1, j)) \in E_1$ werden mit $T_1(i, j)$ markiert, die Kanten $((i, j), (i, j - 1)) \in E_2$ mit $T_2(i, j)$. Der Gittergraph für das Beispiel von oben ist in Abbildung 10 gezeigt.

Sei nun eine Variablenordnung π gegeben. Wir überlegen, wie wir anhand des Gittergraphen die OBDD-Größe bestimmen können. Sei z.B. die erste Variable von π eine x-Variable, z.B. x_1. Nach der Variante des Struktursatzes ergibt sich die Anzahl der x_1-Knoten als $T_1(|V_1| + 1, |V_2| + 1)$. Diese Anzahl können wir an der Kante ablesen, die die Quelle des Gittergraphen über eine E_1-Kante verlässt. Wir erreichen dann den Knoten $(|V_1|, |V_2| + 1)$. Wenn nun die nächste Variable eine y-Variable, z.B. y_2 ist, ergibt sich die Anzahl der y_2-Knoten als $T_2(|V_1|, |V_2| + 1)$, also als die Markierung der E_2-Kante, die den Knoten $(|V_1|, |V_2| + 1)$ verlässt. Allgemein erhalten wir zu einer Variablenordnung einen Pfad im Gittergraphen,

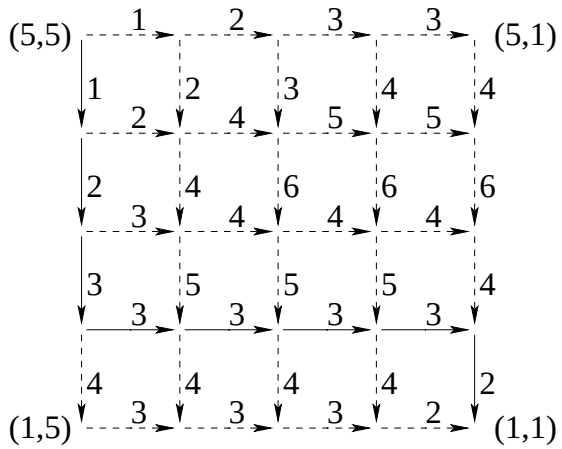


Abbildung 10: Der Gittergraph für die Beispielfunktion

indem wir an der Quelle starten, für jede x-Variable in der Variablenordnung über die E_1-Kante weiterlaufen und für jede y-Variable über die E_2-Kante. Der Pfad endet an der Senke $(1,1)$. Die Anzahl der inneren Knoten des OBDDs ist gleich der Länge des Pfades (bzgl. der Kantenmarkierungen).

Analog entspricht auch jedem Pfad eine Variablenordnung. D.h., wenn wir einen kürzesten Pfad von der Quelle $(|V_1|+1, |V_2|+1)$ zur Senke $(1,1)$ bestimmen (z.B. mit dynamischer Programmierung), erhalten wir auch eine Variablenordnung mit minimaler Knotenzahl. In dem Gittergraphen oben ist der kürzeste Pfad durch durchgezogene Kanten gekennzeichnet, während die Kanten außerhalb des kürzesten Pfades gestrichelt sind. Dies bedeutet, dass alle Variablenordnungen optimal sind, die mit drei x-Variablen beginnen, gefolgt von den vier y-Variablen und schließlich der letzten x-Variablen. Da der gekennzeichnete Pfad der einzige Pfad minimaler Länge ist (hiervon kann man sich leicht überzeugen, indem man für jeden Knoten die Länge des kürzesten Pfades zur Senke bestimmt), gibt es keine anderen optimalen Variablenordnungen.

Wenn wir partiell symmetrische Funktionen mit zwei Symmetriemengen (oder allgemeiner mit einer konstanten Anzahl von Symmetriemengen) betrachten, können wir alle zusammenhängenden Submatrizen in polynomieller Zeit aufzählen und damit die T-Werte und den Gittergraphen bestimmen. Auch die Berechnung kürzester Wege geht bekanntlich in polynomieller Zeit, sodass wir das Variablenordnungsproblem für diese Funktionen in polynomieller Zeit lösen können.

Was passiert bei Funktionen ohne Symmetrien? Die Werte„matrix“ ist dann n-dimensional und hat die Größe $2 \times \cdots \times 2$, ebenso der Gittergraph, dieser ist ein n-dimensionaler Hyperwürfel. Da es exponentiell viele zusammenhängende Submatrizen geben kann (z.B. der Größe $1 \times \cdots \times 1 \times 2$), erfordert auch das Aufzählen exponentielle Zeit (wobei natürlich nicht klar ist, ob das Zählen nicht auch effizienter geht), ebenso das Aufschreiben des Gittergraphen und die Suche nach einem kürzesten Weg. Andererseits arbeitet der Algorithmus von Friedman und Supowit auf eine ähnliche Weise. Jeder Knoten $(j_1, \ldots, j_n)$, $j_k \in \{1,2\}$, des Gittergraphen entspricht einer Indexmenge $I = \{i \mid j_i = 2\}$. Der Wert $\min(I)$ entspricht der Länge eines kürzesten Pfades von dem betrachten Knoten zur Senke $(1, \ldots, 1)$, also ei-

nem Zwischenergebnis der dynamischen Programmierung zur Berechnung eines kürzesten Weges.

Abschließend wollen wir noch eine Heuristik für das Variablenordnungsproblem diskutieren. Diese besagt, dass zusammengehörende Variablen auch zusammen angeordnet werden sollten. Ein Beispiel, das diese Heuristik motiviert, ist die Funktion $x_1x_2 \vee \cdots \vee x_{2n-1}x_{2n}$. Man überlegt leicht, dass es für diese Funktion ein OBDD mit $2n$ inneren Knoten gibt und dass diese Größe nur erreicht wird, wenn die zusammengehörenden Variablen x_{2i-1} und x_{2i} direkt hintereinander in der Variablenordnung stehen. Diese Variablenpaare bilden auch die Symmetriemengen. Offensichtlich gehören Variablen aus derselben Symmetriemenge zusammen. Variablenordnungen, die die Symmetriemengen zusammenlassen, bezeichnen wir als symmetrische Variablenordnungen. Ein Beispiel für eine Funktion, wo die Heuristik nicht zu einer optimalen Variablenordnung führt, haben wir oben bereits gesehen. Mit einem weiteren Beispiel wollen wir veranschaulichen, was die Ursache für diesen Effekt ist.

Sei $n = 3k + 1$. Die Funktion $f_n : \{0,1\}^{n+1} \to \{0,1\}$ ist auf einer x-Variablen und n y-Variablen folgendermaßen definiert:

$$f_n(x, y_1, \ldots, y_n) = \begin{cases} 1, & \text{falls } y_1 + \cdots + y_n = k, \\ x, & \text{falls } y_1 + \cdots + y_n = 2k + 1, \\ 0, & \text{sonst.} \end{cases}$$

Zur Veranschaulichung die Wertematrix für f_{10}:

$$\begin{array}{cccccccccccc} 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{array}$$

Die Anwendung der oben ausgeführten Methoden ist etwas mühselig, da wir auch geschlossene Formeln für die OBDD-Größe herleiten wollen; daher geben wir nur das Ergebnis an. Sei n o.B.d.A. gerade. Die mühseligen Rechnungen zeigen auch, dass die Variablenordnung $y_1, \ldots, y_{n/2}, x, y_{n/2+1}, \ldots, y_n$ optimal ist.

Variablenordnung	Anzahl innerer Knoten
$y_1, \ldots, y_n, x$	$\frac{1}{3}n^2 + \frac{4}{3}n + \frac{1}{3}$
$x, y_1, \ldots, y_n$	$\frac{1}{3}n^2 + \frac{4}{3}n + \frac{4}{3}$
$y_1, \ldots, y_{n/2}, x, y_{n/2+1}, \ldots, y_n$	$\frac{1}{4}n^2 + \frac{4}{3}n + \frac{2}{3}$

Hier unterscheidet sich die OBDD-Größe bei einer optimalen symmetrischen Variablenordnung von der OBDD-Größe bei einer optimalen Variablenordnung um einen Faktor, der gegen $4/3$ konvergiert. Es ist unbekannt, ob es eine nicht triviale obere Schranke für diesen Faktor gibt, der ja die Qualität der untersuchten Heuristik beschreibt. Die Ursache dafür, dass symmetrische Variablenordnungen schlechter sein können, sehen wir in Abbildung 11, das die OBDDs für f_{10} und die beiden symmetrischen Variablenordnungen sowie für $y_1, \ldots, y_6, x, y_7, \ldots, y_n$ zeigt. Die x-Knoten sind dabei schwarz gezeichnet, die y-Knoten weiß. Kanten ohne Endknoten führen zur 0-Senke; die Kanten, die einen Knoten auf der linken Seite verlassen, sind die 0-Kanten, die auf der rechten Seite die 1-Kanten. Wir sehen,

dass in dem Fall, dass die x-Variable ganz unten angeordnet ist, Verschmelzungen von ähnlichen OBDD-Teilen (die im Bild durch Punkte angedeutet sind) verhindert werden; wenn die x-Variable ganz oben angeordnet ist, muss der Wert der x-Variablen gespeichert werden, sodass ähnliche OBDD-Teile im oberen Bereich getrennt werden. Diese Nachteile vermeiden Variablenordnungen, wo die x-Variable in der Mitte getestet wird.

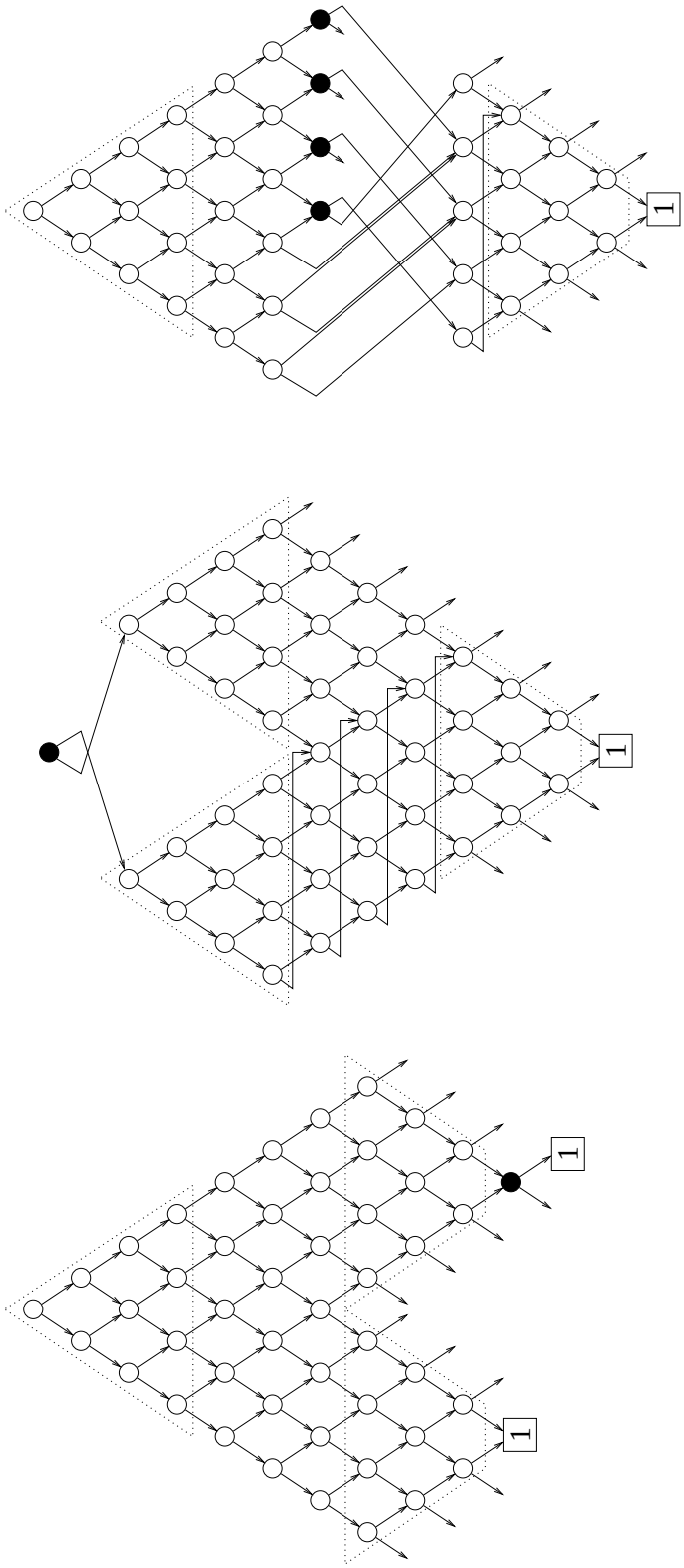


Abbildung 11: OBDDs für die Beispielfunktion f und verschiedene Variablenordnungen

6 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 wie viele 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 $a = (a_1, \ldots, a_n)$ erreicht wird, wird er auch für $a' = (a_1, \ldots, a_{i-1}, \bar{a}_i, a_{i+1}, \ldots, a_n)$ 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 $e_1, \ldots, e_k$ hat, müssen wir die Anzahl der Eingaben, für die $e_1, \ldots, e_k$ 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 einer Tiefensuche 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. $x_1, \ldots, x_n$, 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. 12 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. 12 die nicht erreichbaren Knoten entfernt worden sind, können noch die Knoten (v_3, v_6) und (v_4, v_7) verschmolzen werden.

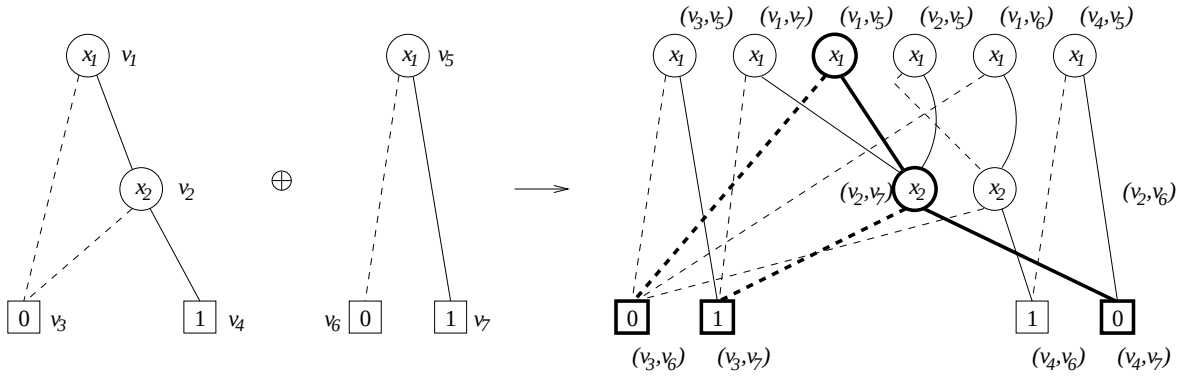


Abbildung 12: Beispiel für die Berechnung eines Produktgraphen

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 sogenannte 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 einen $NULL$-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 sogenannte 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 in einer breadth-first-search-Ordnung 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 $G_1, \ldots, G_m$ für die Funktionen $f_1, \ldots, f_m \in B_n$, eine Funktion $h \in B_m$.
Ausgabe: Ein OBDD für $h(f_1, \ldots, f_m)$.
Rechenzeit: $O(|G_1||G_2| \ldots |G_m|)$. **Speicherplatz:** $O(|G_1||G_2| \ldots |G_m|)$.

Der simultane Durchlauf durch m OBDDs ist genauso möglich wie der simultane Durchlauf durch zwei OBDDs. Dieser Durchlauf kann wiederum durch einen Produktgraphen von $G_1, \ldots, G_m$ simuliert werden. Dieser Produktgraph enthält für jedes m-Tupel $(v_1, \ldots, v_m)$ von Knoten aus $G_1, \ldots, G_m$ einen Knoten. Sei x_i die Variable, die von den Variablen, mit denen $v_1, \ldots, v_m$ markiert sind, als erste in der Variablenordnung steht. Dann wird am Knoten $(v_1, \ldots, v_m)$ die Variable x_i getestet. Die Nachfolger von $(v_1, \ldots, v_m)$ sind $(v_1^0, \ldots, v_m^0)$ bzw. $(v_1^1, \ldots, v_m^1)$, 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 $v_1, \ldots, v_m$ Senken sind, die mit $c_1, \ldots, c_m$ markiert sind, ist auch $(v_1, \ldots, v_m)$ eine Senke, die mit $h(c_1, \ldots, c_m)$ 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 einer Tiefensuche 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. 13 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. 13). 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.

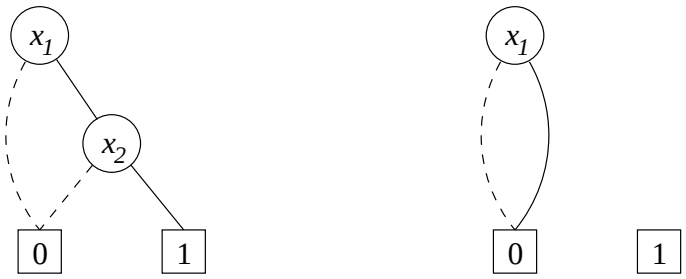


Abbildung 13: OBDDs für $f(x_1, x_2) = x_1 x_2$ und $f_{|x_2=0}$

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

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

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

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

Wir wenden daher den Synthesealgorithmus 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 Synthesealgorithmus 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 Synthesealgorithmus 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 2.2 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 Synthesealgorithmus 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 Synthesealgorithmus, 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 wir im Abschnitt 3 definiert haben. Dort haben wir auch gezeigt, dass das reduzierte OBDD für INDEX_n und die Variablenordnung $a_{k-1}, \ldots, a_0, x_{n-1}, \ldots, x_0$ genau $2n-1$ innere Knoten hat.

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

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

aus OBDDs für $\mathrm{INDEX}_n(x, a)$ und $\mathrm{INDEX}_n(x, b)$. 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 Satz 2.2 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

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

mit der Variablenordnung $s, a_{k-1}, \ldots, a_0, b_{k-1}, \ldots, b_0, x_{n-1}, \ldots, x_0$. 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 $\mathrm{INDEX}_n(x, a)$ oder für $\mathrm{INDEX}_n(x, b)$ 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 $g_n = \mathrm{INDEX}_n(x, c)$ für die Variablenordnung

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

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

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

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.

7 Die Implementierung von OBDD-Paketen

Mit OBDDs möchte man praktisch wichtige boolesche Funktionen möglichst effizient darstellen und möglichst effizient Berechnungen darauf ausführen. Insbesondere sollen Berechnungen möglich werden, die mit anderen Darstellungen von booleschen Funktionen wegen deren Bedarf an Rechenzeit oder Speicherplatz nicht möglich sind. Bisher haben wir die Operationen auf OBDDs aus einer eher theoretischen Sicht beschrieben, also angegeben, wie man diese Operationen algorithmisch realisieren kann, und in vielen Fällen Größenordnungen der Rechenzeiten angegeben. Wir sollten aber nicht vergessen, dass in praktischen Anwendungen viele dieser Operationen nacheinander ausgeführt werden und die berechneten Zwischenergebnisse dabei immer größer werden können. Ein Beispiel dafür ist die Berechnung eines OBDDs für eine Funktion, die durch einen Schaltkreis beschrieben wird. Dazu wird man für jeden Baustein des Schaltkreises eine Synthese ausführen. Während wir für einzelne Synteseoperationen obere Schranken für die Rechenzeit angeben konnten, ist dies für eine solche Folge von Syntheseoperationen nicht mehr möglich. Wir sollten uns auch daran erinnern, dass die Umformung eines Schaltkreises in ein OBDD ein NP-schweres Problem ist. OBDDs sind nur eine Heuristik, die in manchen, aber nicht in allen Fällen effizienter als beispielsweise Wertetabellen sind. Die Effizienz solcher Heuristiken kann man nur mit Experimenten bewerten.

Eine Erfahrung aus Experimenten besteht darin, dass das Hauptproblem bei OBDDs der Speicherplatz ist. Es ist klar, dass das Auslagern von Hauptspeicher sehr langsam ist und die Effizienz von Berechnungen drastisch verringern kann. In der Praxis ist es daher wichtig, dass auch „kleinere“ Verbesserungen insbesondere für den Speicherplatz realisiert werden, da dies entscheiden kann, ob eine Berechnung erfolgreich zu Ende geführt werden kann oder nicht.

Wir wollen daher im Folgenden einige Implementierungstricks für OBDDs diskutieren. Ein Beispiel für eine praktische Realisierung ist das CUDD-Paket, eine Implementierung von Algorithmen für OBDDs und andere BDD-Varianten von Fabio Somenzi. Das CUDD-Paket ist unter http://vlsi.colorado.edu/∼fabio erhältlich.

Gemeinsame Darstellung mehrerer Funktionen

Bei unserer Beschreibung der Operationen sind wir der Einfachheit halber meistens davon ausgegangen, dass die beteiligten Funktionen in separaten OBDDs dargestellt sind. Dafür gibt es keinen Grund. Ein OBDD kann mehrere Startknoten haben und somit mehrere Funktionen darstellen. Dass dies Speicherplatz spart, liegt auf der Hand, da mehrere Funktionen gemeinsame OBDD-Knoten haben können. Wir werden auch noch sehen, dass dies auch das mehrfache Ausführen von identischen Berechnung vermeidet. Wie bereits erwähnt, wird diese Variante von OBDDs manchmal auch als SBDD (shared BDD) bezeichnet.

Output-Inverter

Ein Output-Inverter ist eine zusätzliche Kantenmarkierung in einem OBDD. Jede Kante kann mit einem zusätzlichen Bit markiert werden. Die Auswertung der OBDDs erfolgt so, dass

wie bei gewöhnlichen OBDDs der Berechnungspfad durchlaufen wird und zusätzlich gezählt wird, ob eine gerade oder ungerade Anzahl von Kanten mit Output-Invertern durchlaufen wird. Wenn dies eine ungerade Anzahl ist, wird der Wert der erreichten Senke negiert.

Man überlegt sich leicht, dass die beschriebene Variante von OBDDs mit Output-Invertern nicht mehr bis auf Isomorphie eindeutig ist. Beispielsweise sind eine Kante zur 1-Senke und eine Kante mit Output-Inverter, die zur 0-Senke führt, äquivalent. Daher gibt es zusätzliche Regeln für die Verwendung von Output-Invertern, die die Eindeutigkeit sicherstellen:

1. Es gibt nur noch eine 0-Senke, aber keine 1-Senke mehr.
2. Output-Inverter sind nur auf 1-Kanten, nicht aber auf 0-Kanten erlaubt.
3. Wenn ein OBDD eine Funktion f darstellt, gibt es üblicherweise einen Zeiger zu dem Knoten, der f darstellt. Auch dieser Zeiger darf einen Output-Inverter enthalten.

Gegenüber gewöhnlichen OBDDs besteht der Vorteil der OBDDs mit Output-Invertern darin, dass an jedem Knoten v jetzt zwei Funktionen dargestellt werden können, nämlich eine Funktion f_v (falls v über eine Kanten ohne Output-Inverter erreicht wird) und die Negation $\bar{f_v}$ (falls v über eine Kante mit Output-Inverter erreicht wird). Die bestmögliche Größenersparnis ist demzufolge ein Faktor von zwei. Ein Beispiel dafür ist die Parity-Funktion auf n Variablen. Gewöhnliche OBDDs benötigen $2n - 1$ innere Knoten, während für OBDDs mit Output-Invertern n innere Knoten genügen. Ein weiterer Vorteil besteht darin, dass es mit Output-Invertern sehr einfach wird, für zwei Funktionen zu testen, ob die eine die Negation der anderen ist.

Man kann sich leicht überlegen, dass man ein beliebiges OBDD mit Output-Invertern so umformen kann, dass die obigen Eindeutigkeitsregeln beachtet werden. Eine elementare Umformung besteht darin, dass man für einen Knoten v die Output-Inverter auf allen eingehenden und ausgehenden Kanten invertieren kann, ohne die dargestellte Funktion zu verändern. Durch bottom-up-Anwendung dieser Regel kann man erreichen, dass die Eindeutigkeitsregeln erfüllt sind.

Es ist nun eine einfache, aber doch etwas mühselige Aufgabe, den Struktursatz und die bisher besprochenen Algorithmen auf die OBDDs mit Output-Invertern zu übertragen. Zur Implementierung wollen wir hier nur anmerken, dass in BDD-Paketen das Bit für den Output-Inverter nicht separat gespeichert wird, sondern dass das niederwertigste Bit des zugehörigen Zeigers verwendet wird. Da alle gängigen Rechner nur gerade Adressen verwenden, ist dies immer gleich 0 und kann demzufolge anderweitig, also auch für den Output-Inverter, verwendet werden. Neben dem Einsparen von Speicherplatz kann man so u.U. leichter erreichen, dass der Speicherplatz für einen Knoten (häufig 4 Bytes) ein Teiler der Größe einer Cache-Line ist. Voraussetzung ist natürlich eine Programmiersprache, die Bitmanipulationen dieser Art zulässt, beispielsweise C.

Die Computed-Table

Bei der Beschreibung der Synthese im Abschnitt 6 sind wir davon ausgegangen, dass die Computed-Table zu Beginn jeder Synthese initialisiert, also gelöscht wird. Wenn nun mehrere Synthesen mit derselben Operation ausgeführt werden und alle Funktionen zusammen

in einem OBDD gespeichert werden, gibt es dazu keinen Grund. Wenn beispielsweise bei der ersten $\vee$-Synthese ein Knoten für das Paar (v_1, v_2) erzeugt wurde, kann dieser erzeugte Knoten auch verwendet werden, wenn bei einer späteren $\vee$-Synthese wieder das Paar (v_1, v_2) erreicht wird. Demzufolge braucht man aber für Synthesen mit verschiedenen Operationen mehrere Computed-Tables oder man muss jeweils speichern, für welche Operation ein Eintrag in der Computed-Table erzeugt wurde.

Um Speicherplatz zu sparen, verwendet man für die Computed-Table üblicherweise eine Hashtabelle. Da zu Beginn einer Berechnung nicht klar ist, wie groß diese Tabelle sein muss, wird man ihre Größe in der Regel dynamisch anpassen. Man kann beispielsweise die Tabelle vergrößern, wenn zu viele Kollisionen auftreten. Ein Problem beim Hashing besteht darin, dass die verwendete Kollisionsstrategie zusätzlichen Aufwand, insbesondere an Speicher, benötigt. Als Alternative wurde daher vorgeschlagen, Hashing ohne eine Kollisionsstrategie zu verwenden, diese Variante einer Hashtabelle wird auch als *hash-based cache* bezeichnet. D.h., wenn für ein Paar (v_1, v_2) ein Knoten erzeugt wird und an der zugehörigen Position der Hashtabelle bereits ein Eintrag für das Paar (v_3, v_4) verhanden ist, wird dieser einfach gelöscht und dort das Paar (v_1, v_2) eingetragen. Die Konsequenz ist, dass bei einem späteren Erreichen von (v_3, v_4) wieder die zugehörigen Nachfolger erzeugt werden müssen und anschließend der neu erzeugte Knoten mit dem bereits vorhandenen Knoten verschmolzen wird. Insgesamt kann nicht mehr die polynomielle Laufzeit der Synthese garantiert werden, da auch die Nachfolger von dem Knoten für (v_3, v_4) und deren Nachfolger usw. nicht in der Computed-Table enthalten sein müssen (obwohl dies in der Praxis häufig der Fall sein wird). Allerdings gilt die Verwendung einer Computed-Table ohne Kollisionsstrategie als praktisch effizient.

Um die Computed-Table möglichst klein und effizient zu halten, ist es auch sinnvoll, die Einträge zu standardisieren. Wenn wir beispielsweise wieder die Operation $\vee$ betrachten und in der Computed-Table einen Eintrag für das Knotenpaar (v_1, v_2) haben, sollte dieser Eintrag auch gefunden werden, wenn später ein Knoten für das Paar (v_2, v_1) benötigt wird. Also benötigt man eine Ordnung auf den Knoten und kann dann zumindest für die symmetrischen Operationen die Knoten in einer bestimmten Reihenfolge in die Computed-Table eintragen. In diese Richtung geht auch die Verwendung des If-Then-Else-Operators. Man überlegt sich leicht, dass man alle gängigen binären Operatoren mit Hilfe des If-Then-Else-Operators darstellen kann. So ist $f \wedge g = ite(f, g, 0)$ und $f \oplus g = ite(f, \bar{g}, g)$. Es genügt demzufolge eine Computed-Table für den If-Then-Else-Operator.

Beim If-Then-Else-Operator sind viele Standardisierungen möglich. Beispielsweise gilt

$$ite(f, f, g) = ite(f, 1, g) = ite(g, 1, f) = ite(g, g, f)$$

oder

$$ite(f, g, \bar{f}) = ite(f, g, 1).$$

Um diese letzte Umformung realisieren zu können, muss man für zwei Funktionen effizient testen können, ob die eine die Negation der anderen ist. Dies ist wegen der Output-Inverter einfach möglich. Insgesamt muss man für die obigen Gleichungen (und einige weitere, die wir hier nicht aufführen wollen) festlegen, welche Form in der Computed-Table gespeichert wird.

Die Unique-Table

Anders als bei der Computed-Table dürfen bei der Unique-Table keine noch benötigten Einträge gelöscht werden, da sonst die Eindeutigkeit der OBDDs verloren geht. Also wird eine Kollisionsstrategie benötigt, wobei man sowohl offenes als auch geschlossenes Hashing verwenden kann. Für Knoten, die mit verschiedenen Variablen markiert sind, kann man auch verschiedene Hashtabellen vorsehen.

Beim Konstruieren von OBDDs wird es häufiger vorkommen, dass berechnete Zwischenergebnisse nicht mehr benötigt werden. Die entsprechenden Knoten können also gelöscht werden. Wenn ein Knoten gelöscht wird, werden dessen Nachfolger auch nicht mehr benötigt, falls es nicht noch andere Zeiger auf sie gibt. Um effizient feststellen zu können, ob ein Knoten noch benötigt wird, kann man in den Knoten einen Zähler für die Anzahl der auf ihn verweisenden Zeiger speichern, den sogenannten Referenzzähler. Wenn dieser bei einem Knoten der Wert 0 erreicht, kann der Knoten gelöscht werden, und die Referenzzähler der Nachfolger sind dementsprechend zu verringern. Es kann weiterhin Verweise von der Computed-Table auf gelöschte Knoten geben (die beim Referenzzähler nicht mitgezählt werden). Wenn ein solcher Knoten bei einem späteren Aufruf von Synthese wieder benötigt wird, kann man ihn einfach „wiederbeleben“, wenn er noch vorhanden ist. Somenzi (2001) berichtet auch, dass dies in der Praxis Vorteile bringt.

Um die OBDD-Knoten möglichst klein zu halten, werden manchmal nur 2 Bytes für den Referenzzähler verwendet. Wenn dieser im Laufe einer Rechnung seinen Maximalwert 65535 erreicht, wird er beim späteren Löschen von Verweisen nicht wieder verringert, sodass der zugehörige Knoten niemals mehr gelöscht wird. Dies ist natürlich ein Kompromiss zwischen den Zielen, die OBDD-Knoten möglichst klein zu halten und die Anzahl der gespeicherten OBDD-Knoten möglichst klein zu halten.

Aus Effizienzgründen wird der Speicherplatz nicht einzeln für die OBDD-Knoten angefordert, sondern es werden größere Blöcke angefordert, sodass das BDD-Paket die Speicherverwaltung innerhalb dieser Blöcke selbst organisieren muss. Wenn das OBDD bzw. die Unique-Table zu viele Knoten mit Referenzzähler 0 enthält, wird eine Garbage-Collection durchgeführt. Dazu wird die Computed-Table durchlaufen und Verweise auf Knoten mit Referenzzähler 0 werden gelöscht, in manchen Implementierungen auch die gesamte Computed-Table. Anschließend wird die Unique-Table durchlaufen und dort werden Verweise auf Knoten mit Referenzzähler 0 ebenfalls gelöscht. Die gelöschten Knoten werden in eine Liste mit freigegebenen Knoten eingehängt, sodass der Speicher wiederverwendet werden kann. In manchen BDD-Paketen werden bei der Garbage-Collection Knoten im Speicher verschoben, um die Speicherzugriffe bei den Operationen auf den OBDDs möglichst lokal zu halten und somit die Anzahl der Cache-Misses zu verringern.

Diskussion

Anhand der Beschreibung der Implementierungstricks sollte deutlich geworden sein, dass bei der Implementierung von OBDD-Paketen weniger objektorientierte Konzepte zum Einsatz gekommen sind, sondern dass vielmehr maschinennahe Tricks verwendet werden. Die

Ursache hierfür ist auch in der Art der Veröffentlichungen in der praktisch ausgerichteten Richtung im Bereich Schaltkreisverifikation zu sehen. Diese Veröffentlichungen enthalten fast ausnahmslos Tabellen mit Rechenzeiten auf sogenannten Benchmarks. Bei der Entscheidung über die Annahme dieser Veröffentlichungen spielt es auch eine wichtige Rolle, ob der verwendete Ansatz effizienter (in Bezug auf CPU-Sekunden) als frühere Ansätze ist oder ob es beispielsweise möglich ist, damit größere Schaltkreise als in früheren Arbeiten zu verifizieren. Auf der einen Seite sind Experimente erforderlich, um die Anwendbarkeit von Heuristiken zu überprüfen. Auf der anderen Seite führt dies auch dazu dass, Detailverbesserungen der in diesem Abschnitt beschriebenen Art wichtig werden.

Diese Vorgehensweise hat auch Konsequenzen für den Erfolg von weiteren BDD-Varianten. Da bereits gut optimierte Pakete für OBDDs zur Verfügung stehen, bilden diese auch den Maßstab für die Implementierung von neuen OBDD-Varianten. Demzufolge waren die OBDD-Varianten am erfolgreichsten, die sich am leichtesten durch Modifikation von vorhandenen BDD-Paketen implementieren ließen.

8 Operationen zur Veränderung der Variablenordnung

Neben der Operation Swap, die die Variablenordnung nur lokal verändert und die wir bereits weiter oben behandelt haben, gibt es noch Operationen, die auch nicht lokale Veränderungen der Variablenordnung realisieren.

Die Operation Exchange

Eingabe: Ein OBDD G für f mit der Variablenordnung $x_1, \ldots, x_n$, sowie i und j mit $1 \leq i < j \leq n$.
Ausgabe: Ein OBDD für f mit der Variablenordnung $x_1, \ldots, x_{i-1}, x_j, x_{i+1}, \ldots, x_{j-1}, x_i, x_{j+1}, \ldots, x_n$.
Rechenzeit: $O(|G|^2)$. **Speicherplatz:** $O(|G|^2)$.

Wir bezeichnen mit $f_{|x_i \leftrightarrow \bar{x}_i}$ die Funktion, die durch Negation der Variablen x_i entsteht. Auf OBDDs lässt sich dies leicht realisieren, indem an den x_i-Knoten die 0- und 1-Nachfolger vertauscht werden.

Wir berechnen aus G ein OBDD G' für $f_{|x_i \leftrightarrow \bar{x}_i, x_j \leftrightarrow \bar{x}_j}$. Weiterhin berechnen wir ein OBDD für $h(x_1, \ldots, x_n) = x_i \oplus x_j$. Anschließend berechnen wir ein OBDD für $ite(h, f_{|x_i \leftrightarrow \bar{x}_i, x_j \leftrightarrow \bar{x}_j}, f)$. In diesem OBDD ist einfach die Bedeutung von x_i und x_j vertauscht, also ist es das Resultat der Exchange-Operation. Da das OBDD für h konstante Größe hat, ergeben sich die Schranken für Rechenzeit und Speicherplatz aus den Schranken für die Synthese mit ite.

Die Operation Jump-up

Eingabe: Ein OBDD G für f mit der Variablenordnung $x_1, \ldots, x_n$, sowie i und j mit $1 \leq j < i \leq n$.
Ausgabe: Ein OBDD für f mit der Variablenordnung, in der x_i an Position j springt und die Variablen $x_j, \ldots, x_{i-1}$ eine Position nach hinten verschoben werden.
Rechenzeit: $O(|G|)$. **Speicherplatz:** $O(|G|)$.

Seien $v_1, \ldots, v_l$ die Knoten, die mit einer der Variablen $x_j, \ldots, x_n$ markiert sind und mindestens einen Vorgänger haben, der mit einer der Variablen $x_1, \ldots, x_{j-1}$ markiert ist, und seien $f_1, \ldots, f_l$ die an diesen Knoten berechneten Funktionen. Durch Kopieren erhalten wir zwei SBDDs für $f_1, \ldots, f_l$ und erzeugen hieraus ein SBDD G_0 für $f_{1|x_i=0}, \ldots, f_{l|x_i=0}$ und ein SBDD G_1 für $f_{1|x_i=1}, \ldots, f_{l|x_i=1}$. Diese beiden SBDD enthalten keine x_i-Knoten. In G ersetzen wir die Knoten $v_1, \ldots, v_l$ durch x_i-Knoten, wobei der c-Nachfolger des x_i-Knotens, der v_r ersetzt, auf den Knoten von G_c zeigt, der $f_{r|x_i=c}$ berechnet.

Auf diese Weise realisieren wir eine Synthese für $ite(x_i, f_{|x_i=1}, f_{|x_i=0})$ direkt, damit offensichtlich wird, dass die konstruierten OBDDs lineare Größe in der Eingabe haben und lineare Rechenzeit ausreicht.

Die Operation Jump-down

Eingabe: Ein OBDD G für f mit der Variablenordnung $x_1, \ldots, x_n$, sowie i und j mit $1 \leq i < j \leq n$.
Ausgabe: Ein OBDD für f mit der Variablenordnung, in der x_i an Position j springt und die Variablen $x_{i+1}, \ldots, x_j$ eine Position nach vorne verschoben werden.
Rechenzeit: $O(|G|^2)$. **Speicherplatz:** $O(|G|^2)$.

Wir fügen eine neue Variable y an die $(j+1)$-te Position der Variablenordnung ein, sodass alle Variablen ab x_{j+1} um eine Position nach hinten verschoben werden. Dann erzeugen wir ein OBDD für die Funktion $h(x_1, \ldots, x_n, y) = y$. Mit Ersetzung durch Konstanten berechnen wir OBDDs für $f_{|x_i=0}$ und $f_{|x_i=1}$ und mit der ite-Synthese ein OBDD für $ite(y, f_{|x_i=1}, f_{|x_i=0})$. In diesem OBDD kommt x_i nicht vor, daher kann x_i gelöscht werden und anschließend y in x_i umbenannt werden.

Wir sehen, dass sich bei Jump-down die Größe quadrieren kann, während bei Jump-up die Größe nur linear wachsen kann. An einem Beispiel wollen wir zeigen, dass das Quadrieren der Größe auch wirklich vorkommen kann. Die betrachtete Funktion ist

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

mit der Variablenordnung $s, a_{k-1}, \ldots, a_0, b_{k-1}, \ldots, b_0, x_{n-1}, \ldots, x_0$. Wir haben bereits bei dem worst-case Beispiel für Quantifizierung erwähnt, dass die OBDD-Größe linear ist. Wenn nun die Variable s an die letzte Position springt, sehen wir, dass wir durch Konstantsetzen der a- und b-Variablen auf die n^2 verschiedenen Weisen die n^2 verschiedenen Subfunktionen $\bar{s}x_i \vee sx_j$ erhalten, also dass das OBDD mit der neuen Variablenordnung quadratische Größe hat.

Die Ursache für das verschiedene Verhalten der beiden Jump-Operationen liegt darin, dass wir bei Jump-up die verschobene Variable nur „speichern“ müssen, was die Breite höchstens verdoppelt. Bei einem Jump-down dagegen müssen wir simultan beide Berechnungspfade verfolgen, die bei den beiden Belegungen der verschobenen Variablen durchlaufen werden. Dafür gibt es unter Umständen quadratisch viele Möglichkeiten.

Mit derselben Funktion f_n können wir zeigen, dass auch bei der Operation Exchange ein Quadrieren der OBDD-Größe möglich ist. Wir betrachten dazu die Funktion f_n und ein OBDD mit der Variablenordnung $s, a_{k-1}, \ldots, a_0, b_{k-1}, \ldots, b_0, x_{n-1}, \ldots, x_0, t$, d.h., gegenüber der Variablenordnung oben gibt es noch die redundante Variable t. Beim Vertauschen von s und t ergibt sich dann dasselbe OBDD wie beim Jump-down von s.

Umordnen von OBDDs

Eingabe: Ein reduziertes OBDD G für $f \in B_n$ mit der Variablenordnung π.
Ausgabe: Das reduzierte OBDD H für f mit der Variablenordnung $x_1, \ldots, x_n$.
Rechenzeit: $O(|G||H| \log |H|)$. **Speicherplatz:** $O(|G| + |H|)$.

Es sollte klar sein, dass die vorgegebene Variablenordnung für H keine Einschränkung ist, da wir die Variablen umbenennen können. Wir beschreiben zunächst eine einfachere Variante

des Algorithmus mit Speicherplatzkomplexität $O(|G||H|)$. O.B.d.A. sei f von allen Variablen essentiell abhängig. Als Datenstruktur verwenden wir Listen $L(1), \ldots, L(n)$ von Paaren von Knoten und partiellen Variablenbelegungen. Wir erzeugen zunächst die zwei Senken für H. Die inneren Knoten von H werden schichtweise von oben nach unten konstruiert, wobei wir darauf achten, dass keine Knoten in H eingefügt werden, die später durch Reduktionsregeln wieder entfernt werden müssen. In der Liste $L(i)$ speichern wir die x_i-Knoten, für die wir noch entscheiden müssen, ob auf sie die Verschmelzungsregel anwendbar ist; wir werden dafür sorgen, dass keine Knoten in $L(i)$ eingefügt werden, auf die die Eliminationsregel anwendbar ist. Zu jedem x_i-Knoten v in $L(i)$ wird eine Belegung $c_1, \ldots, c_{i-1}$ der Variablen $x_1, \ldots, x_{i-1}$ gespeichert, sodass die an v berechnete Funktion gleich $f_{|x_1=c_1,\ldots,x_{i-1}=c_{i-1}}$ ist.

Zu Beginn wird der Startknoten von f erzeugt. Da f von allen Variablen essentiell abhängt, wird der Startknoten mit x_1 markiert und in $L(1)$ eingefügt; die zugehörige Belegung ist die leere Belegung.

Anschließend werden nacheinander die Listen $L(1), \ldots, L(n)$ bearbeitet. Für $L(i)$ werden dabei die folgenden Schritte ausgeführt:

1. In $L(i) = \{v_1, \ldots, v_r\}$ werden Knoten gesucht, die verschmolzen werden können. Da noch keine Nachfolger für diese Knoten berechnet wurden, können wir die Verschmelzungsregel nicht einfach anwenden, sondern benötigen eine andere Methode. Zu jedem Knoten v_j ist eine Belegung σ_j der vorher zu testenden Variablen gespeichert, sodass v_j erreicht wird. Also können wir aus G OBDDs für $f_{|\sigma_j}$ mit der Variablenordnung π berechnen. Wir stellen alle diese Funktionen in einem SBDD dar und reduzieren dies. Wenn zwei Funktionen $f_{|\sigma_j}$ und $f_{|\sigma_k}$ übereinstimmen, werden sie im SBDD an demselben Knoten dargestellt. Dann können v_j und v_k verschmolzen werden. Die verbliebenen Knoten werden in H benötigt, da an ihnen verschiedene Funktionen aus S_i (siehe Struktursatz) dargestellt werden. Die Verschmelzungen führen wir wie beim Reduktionsalgorithmus mit Hilfe der Knotennummern durch.

2. Für jeden Knoten $v_j \in L(i)$ werden die beiden Nachfolger erzeugt. Für den c-Nachfolger ($c \in \{0, 1\}$) wird die erste Variable x_k in der Variablenordnung $x_1, \ldots, x_n$ gesucht, von der die Funktion $f_{v_j|x_i=c}$ essentiell abhängt, und der Knoten wird in $L(k)$ gespeichert. Die zugehörige Variablenbelegung ergibt sich aus der Belegung für v_j, aus $x_i = c$ und einer beliebigen Belegung der Variablen $x_{i+1}, \ldots, x_{k-1}$. Falls $f_{v_j|x_i=c}$ die konstante Funktion d ist, wird kein Knoten erzeugt, sondern der Zeiger für den c-Nachfolger von v_j wird auf die d-Senke gesetzt. Alle diese Tests für $f_{v_j|x_i=c}$ können in Zeit $O(|G|)$ mit Hilfe des in Schritt 1 berechneten OBDDs für f_{v_j} ausgeführt werden.

Für die Senken ist offensichtlich, dass an ihnen die konstanten Funktionen berechnet werden. Mit Induktion über eine umgekehrte topologische Ordnung der Knoten kann man zeigen, dass an jedem Knoten die Subfunktion von f berechnet wird, die durch die zugehörige partielle Variablenbelegung beschrieben wird. Für einen x_i-Knoten v mit der partiellen Variablenbelegung σ ist $f_{|\sigma,x_i=c}$ entweder eine konstante Funktion, dann zeigt der Nachfolger auf die entsprechende Senke, oder eine Funktion, für die ein Knoten w mit der partiellen Belegung $\sigma, x_i = c$ in $L(j)$ eingefügt wurde. Wenn w mit anderen Knoten verschmolzen wurde,

kann sich die zugeordnete partielle Belegung verändern, nicht aber die dort dargestellte Subfunktion. Also wird für $c \in \{0,1\}$ am c-Nachfolger von v die Funktion $f_{|\sigma,x_i=c}$ berechnet, was für die inneren Knoten nach Induktionsvoraussetzung folgt und für die Senken klar ist. Damit wird an v die Funktion $f_{|\sigma}$ dargestellt.

Für die Abschätzung von Rechenzeit und Speicherplatz beachten wir, dass für jeden inneren Knoten in H höchstens zwei Knoten in die Listen $L(1), \ldots, L(n)$ eingefügt werden. Alle Listen zusammen enthalten also $O(|H|)$ viele Knoten. Das SBDD, das für $L(j)$ erzeugt wird, enthält höchstens $|L(j)||G|$ Knoten. Da f von allen Variablen essentiell abhängt, können die Ersetzungen durch Konstanten in linearer Zeit ausgeführt werden. Die Rechenzeit der Reduktion ist ebenfalls linear, also $O(|L(j)||G|)$. Insgesamt ergibt sich die Schranke $O(|G||H|)$ für Rechenzeit und Speicherplatz.

Abschließend diskutieren wir, wie wir die Speicherplatzkomplexität verbessern können. Dazu wollen wir vermeiden, die OBDDs für die Funktionen $f_{|\sigma_j}$ gleichzeitig zu speichern. Um dennoch die Verschmelzungsregel anwenden zu können, definieren wir eine Ordnung auf den OBDDs für die Funktionen $f_{|\sigma_j}$, und die Subfunktionen werden gemäß dieser Ordnung in einem AVL-Baum gespeichert, wobei die Subfunktionen durch die partiellen Variablenbelegungen repräsentiert werden. Dagegen werden die zugehörigen OBDDs nicht gespeichert, sondern bei jedem Zugriff (z.B. bei Vergleichen für die AVL-Baum-Operationen) neu berechnet, was in linearer Zeit möglich ist. Wir geben nun die verwendete Ordnung an: Zu jedem OBDD wird eine Codierung der Größe $O(|G|)$ berechnet, indem das OBDD mit DFS durchlaufen wird (jeweils 0-Nachfolger vor 1-Nachfolger) und zu jedem Knoten seine DFS-Nummer, seine Markierung und die DFS-Nummern der Nachfolger angegeben werden. Die Ordnung ist dann die lexikographische Ordnung auf den Codierungen. Diese Codierungen können in linearer Zeit berechnet und verglichen werden.

Für die Reduktion von $L(i)$ genügen dann $O(|L(i)|)$ AVL-Baum-Operationen; für jede davon sind $O(\log |L(i)|)$ Vergleiche von OBDDs ausreichend, die ihrerseits inklusive der Neuberechnung der jeweiligen OBDDs in Zeit $O(|G|)$ möglich sind. Insgesamt erhält man die Rechenzeit $O(\sum_i |L(i)|(\log |L(i)|)|G|) = O(|G||H| \log |H|)$, wobei wir ausnutzen, dass alle Listen zusammen höchstens $O(|H|)$ Knoten enthalten. Da nur $O(|H|)$ Knoten und nur konstant viele OBDDs, die aus G durch Ersetzen durch Konstanten entstehen, gespeichert werden, erhalten wir die Speicherplatzschranke $O(|G| + |H|)$.

9 Zero-suppressed BDDs

Wir starten mit der Definition von Zero-suppressed BDDs (ZBDDs). Anschließend werden wir die Änderung gegenüber OBDDs motivieren und die veränderten Eigenschaften von ZBDDs diskutieren.

Die den ZBDDs zugrundeliegenden Graphen sind die OBDDs, die nun allerdings auf eine andere Weise ausgewertet werden. Die Berechnung für eine Eingabe a startet an der Quelle und folgt demselben Berechnungspfad wie in einem OBDD. Es wird der Funktionswert 1 berechnet, wenn der Berechnungspfad an einer 1-Senke endet **und** alle Variablen, die auf dem Berechnungspfad *nicht* getestet werden, den Wert 0 haben.

Als Beispiel für ein ZBDD betrachten wir den Graphen auf der linken Seite von Abbildung 14. Wir sehen zunächst, dass es sich von einem OBDD nicht im Aussehen, sondern nur in der Art der Funktionsauswertung unterscheidet. Für die Eingabe $x_1 = 0, x_2 = 1, x_3 = 1$ berechnet das ZBDD den Wert 0; als OBDD würde es den Wert 1 berechnen.

Die ZBDDs wurden eingeführt, um die charakteristischen Funktionen von Mengen von Mengen effizienter darzustellen. Sei $S = \{T_1, \ldots, T_l\}$ mit $T_i \subseteq \{1, \ldots, n\}$. Dann wird jede Menge $T_i = \{i_1, \ldots, i_k\}$ durch ein Monom m_i repräsentiert, das die Variablen $x_{i_1}, \ldots, x_{i_k}$ positiv und die übrigen negiert enthält. Schließlich wird S durch die Disjunktion dieser Monome dargestellt. Wenn nun die einzelnen Mengen T_i „wenige“ Variablen enthalten, enthalten die OBDDs viele Tests von Variablen auf 0, die in dem entsprechenden ZBDD wegfallen können.

Wir wollen nun den Einfluss der Reduktionsregeln auf ZBDDs untersuchen. Das linke BDD in Abbildung 14 ergibt sich aus dem mittleren durch Anwendung der Eliminationsregel. Wenn wir die beiden Graphen als OBDDs betrachten, stellen sie offensichtlich dieselbe Funktion dar. Wenn wir dagegen das mittlere BDD als ZBDD auffassen, ergibt sich für die Eingabe $x_1 = 0, x_2 = 1, x_3 = 1$ der Funktionswert 1. Wir schließen daraus, dass wir auf ZBDDs die Eliminationsregel für OBDDs *nicht* anwenden dürfen. Wenn wir auf der Kante zwischen dem x_1- und x_3-Knoten einen x_2-Knoten einfügen wollen, muss der 1-Nachfolger zur 0-Senke zeigen; dies simuliert den Test, den die ZBDDs auf ausgelassenen Variablen ausführen. Das resultierende ZBDD ist auf der rechten Seite von Abbildung 14 gezeigt. Es ist nun naheliegend, eine neue Reduktionsregel einzuführen. Mit der Eliminationsregel für

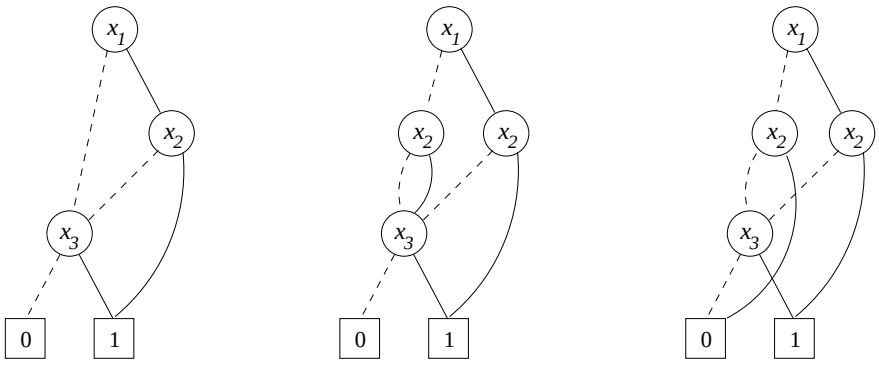


Abbildung 14: Beispiele für ZBDDs und OBDDs

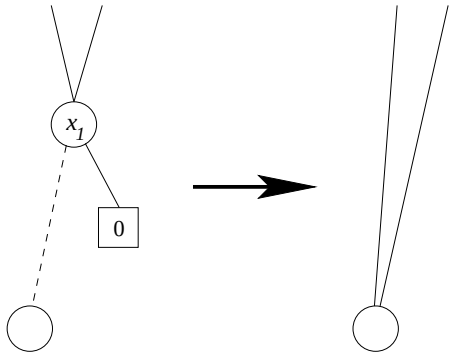


Abbildung 15: Die Eliminationsregel für ZBDDs

ZBDDs können alle Knoten gelöscht werden, deren 1-Nachfolger die 0-Senke ist. Alle eingehenden Kanten werden auf den 0-Nachfolger des gelöschten Knotens umgelenkt. Dies ist in Abbildung 15 gezeigt. Dagegen sollte klar sein, dass die Verschmelzungsregel die Menge der auf den Berechnungspfaden getesteten Variablen nicht ändert, sodass sie gleichermaßen für OBDDs und ZBDDs anwendbar ist.

Wir wollen nun einen Struktursatz für ZBDDs entwerfen. Dazu überlegen wir, welche Funktionen an den Knoten eines ZBDDs dargestellt werden. Wenn wir im obigen ZBDD die Auswertung am mit x_3 markierten Knoten starten, werden die Tests von x_1 und x_2 ausgelassen, so dass implizit die Konjunktion mit $\bar{x}_1$ und $\bar{x}_2$ berechnet wird. Allgemein werden bei der Variablenordnung $x_1, \ldots, x_n$ an den x_i-Knoten Funktionen der Form $\bar{x}_1 \wedge \cdots \wedge \bar{x}_{i-1} f_{|x_1=c_1,\ldots,x_{i-1}=c_{i-1}}$ berechnet. In OBDDs konnten die x_i-Knoten entfallen, an denen Funktionen berechnet werden, die von x_i nicht essentiell abhängen, also auf die die Eliminationsregel anwendbar ist. Analog können in ZBDDs die x_i-Knoten entfallen, auf die die Eliminationsregel für ZBDDs anwendbar ist. Dies sind die Knoten, an denen eine Funktion g dargestellt wird, sodass $g_{|x_i=1} = 0$ gilt. Insgesamt erhalten wir den Struktursatz für ZBDDs.

Satz 9.1: Sei $f \in B_n$ und sei

$$S_i := \{g = \bar{x}_1 \wedge \cdots \wedge \bar{x}_{i-1} \wedge f_{|x_1=c_1,\ldots,x_{i-1}=c_{i-1}} \mid g_{|x_i=1} \neq 0, c_1, \ldots, c_{i-1} \in \{0,1\}\}.$$

Das minimale ZBDD für f und die Variablenordnung $x_1, \ldots, x_n$ ist bis auf Isomorphie eindeutig und enthält genau $|S_i|$ innere Knoten, die mit x_i markiert sind. Weiterhin kann das minimale ZBDD für f und die Variablenordnung $x_1, \ldots, x_n$ aus jedem ZBDD für f und die Variablenordnung $x_1, \ldots, x_n$ berechnet werden, indem die Reduktionsregeln so lange wie möglich angewendet werden.

Da der Beweis gegenüber dem für OBDDs keine neuen Ideen enthält, wollen wir hier darauf verzichten. Auch die Übertragung der Reduktionsalgorithmen ist einfach.

Eine wesentliche Frage für den Einsatz von ZBDDs ist die Frage nach der Größenersparnis gegenüber OBDDs. Wir sehen, dass an den Knoten von OBDDs und ZBDDs verschiedene Funktionen dargestellt werden. Um die Beziehung zwischen den Größen herzuleiten, definieren wir noch eine besondere Variante von OBDDs und ZBDDs.

Definition 9.2: Ein OBDD oder ZBDD heißt *vollständig* (oder geschichtet), falls auf jedem Pfad von der Quelle zu einer Senke jede Variable *genau* einmal getestet wird. Falls außerdem die Verschmelzungsregel nicht anwendbar ist, nennen wir den Graphen quasireduziert.

Da sich die Semantik von OBDDs und ZBDDs nur durch die Behandlung der Variablen unterscheidet, die auf den Berechnungspfaden ausgelassen werden, können wir ein vollständiges OBDD auch als ZBDD interpretieren und umgekehrt, wobei beide dieselbe Funktion darstellen. Die Frage nach den Größenunterschieden können wir daher lösen, indem wir die Größenunterschiede zwischen reduzierten und quasireduzierten OBDDs bzw. ZBDDs untersuchen.

Um ein reduziertes OBDD vollständig zu machen, fügen wir für jeden x_i-Knoten v_i eine Kette von x_1-, ..., x_{i-1}-Knoten $v_1, \ldots, v_{i-1}$ ein, sodass beide Nachfolger von v_j auf v_{j+1} zeigen. Analog wird für jede Senke eine entsprechende Kette von x_1-, ..., x_n-Knoten erzeugt. Dann werden alle eingehenden Kanten zu v_i, die an einem mit x_k markierten Knoten starten, zu v_{k+1} „umgebogen“. Dies entspricht der umgekehrten Anwendung der Eliminationsregel für OBDDs. Abschließend werden alle nicht erreichbaren Knoten entfernt. Man sieht, dass sich die Größe um einen Faktor von höchstens $n+1$ vergrößert (wobei die Senken bei der Größe mitgezählt werden), da für jeden Knoten höchstens n Knoten hinzugefügt werden. Man überlegt auch leicht, dass der Faktor $n+1$ optimal ist, er ergibt sich bei einem OBDD, das nur aus einer Senke besteht.

In ähnlicher Weise können wir aus einem reduzierten ZBDD ein quasireduziertes ZBDD machen; lediglich die Kette von den Knoten $v_1, \ldots, v_{i-1}$ wird anders konstruiert: Alle 1-Nachfolger zeigen zur 0-Senke; nur die 0-Nachfolger zeigen zum nächsten Knoten. Dies entspricht der umgekehrten Anwendung der Eliminationsregel für ZBDDs.

Wenn wir also ein OBDD in ein ZBDD umformen wollen, wenden wir auf die eben beschriebene Art die inverse Eliminationsregel für OBDDs an und anschließend die Eliminationsregel für ZBDDs. Analoges gilt für die Umformung von ZBDDs in OBDDs. Ein Sonderfall ist hierbei die Funktion $\bar{x}_1 \wedge \cdots \wedge \bar{x}_n$, deren ZBDD nur aus einer 1-Senke besteht. Bei der Umformung muss dann zusätzlich eine 0-Senke erzeugt werden. Wenn also $\mathrm{OBDD}(f,\pi)$ und $\mathrm{ZBDD}(f,\pi)$ die Größe von reduzierten OBDDs bzw. ZBDDs für f und π bezeichnen, gilt:

$$\begin{aligned} \mathrm{OBDD}(f,\pi) &\leq (n+1)\mathrm{ZBDD}(f,\pi), \qquad \text{falls } f \neq \bar{x}_1 \wedge \cdots \wedge \bar{x}_n, \\ \mathrm{ZBDD}(f,\pi) &\leq (n+1)\mathrm{OBDD}(f,\pi). \end{aligned}$$

Bei der Funktion $\bar{x}_1 \wedge \cdots \wedge \bar{x}_n$ wird ein ZBDD, das nur aus einer 1-Senke besteht, in ein OBDD mit $n+2$ Knoten umgeformt. Insbesondere haben Funktionen mit exponentieller OBDD-Größe auch exponentielle ZBDD-Größe und auch die Eigenschaften nice, ugly, usw. stimmen für OBDDs und ZBDDs überein. Der Faktor $n+1$ erscheint recht klein, da wir bisher immer an exponentiellen unteren Schranken interessiert waren. In praktischen Anwendungen kann er allerdings den Unterschied zwischen realisierbar und nicht mehr realisierbar ausmachen.

Von den Operationen auf ZBDDs behandeln wir nur die Synthese und das Ersetzen durch Konstanten, da die anderen Operationen ähnlich zu den Operationen für OBDDs ausgeführt

werden können. Zunächst wollen wir motivieren, warum die Synthese anders als bei OBDDs ausgeführt werden muss. Wir betrachten dazu die Operation NAND. Das ZBDD, das nur aus der 0-Senke besteht, berechnet die Nullfunktion. Wir wenden nun NAND auf zwei Kopien der Nullfunktion an. Im OBDD-Fall erhalten wir wieder ein OBDD mit konstanter Größe (das nur aus der 1-Senke besteht). Für ZBDDs benötigt die Einsfunktion $n+1$ Knoten. Also können wir das ZBDD für das NAND von zwei Kopien der Nullfunktion nicht einfach durch Konstruktion eines Produktgraphen erhalten.

Es hat sich herausgestellt, dass eine modifizierte Produktgraphkonstruktion für sogenannte 0-erhaltende Operatoren möglich ist; dies sind Operatoren $\otimes$, für die $0 \otimes 0 = 0$ gilt. Wir diskutieren anschließend, wie wir die Synthese für Operatoren ohne diese Eigenschaft realisieren können.

Sei also $\otimes$ ein 0-erhaltender Operator und seien die ZBDDs $G_1 = (V_1, E_1)$ und $G_2 = (V_2, E_2)$ für f_1 und f_2 mit der Variablenordnung $x_1, \ldots, x_n$ gegeben. Wir konstruieren wie bei den OBDDs den Produktgraphen, allerdings mit einer Ausnahme: Seien $v \in V_1$ und $w \in V_2$, wobei v mit x_i und w mit x_j, o.B.d.A. $j > i$, markiert ist und v die Nachfolger v_0 und v_1 hat. Der 0-Nachfolger von (v, w) ist dann wie vorher (v_0, w), der 1-Nachfolger ist aber $(v_1, 0\text{-Senke})$. Mit diesem Algorithmus kann analog zu den OBDDs der Produktgraph berechnet werden. Wir wollen nun erklären, wie sich die modifizierte Produktgraphkonstruktion ergibt.

Sei a eine Eingabe mit $f_1(a) = 1$ oder $f_2(a) = 1$, o.B.d.A. $f_1(a) = 1$. In G_1 werden dann auf dem Berechnungspfad für a alle Variablen mit dem Wert 1 getestet und es wird die 1-Senke erreicht. Wenn auch $f_2(a) = 1$, gilt für G_2 dasselbe und damit auch für den Produktgraphen. Wenn $f_2(a) = 0$, wird in G_2 die 0-Senke erreicht oder der Test einer Variablen mit dem Wert 1 wird ausgelassen; im letzteren Fall darf auch eine 1-Senke erreicht werden, obwohl der Funktionswert 0 ist. Aufgrund der Modifikation wird in jedem Fall im Produktgraphen ein Paar $(1\text{-Senke}, 0\text{-Senke})$ erreicht, und damit der korrekte Wert berechnet.

Sei nun $f_1(a) = f_2(a) = 0$. Dann wird in jedem der beiden ZBDDs die 0-Senke erreicht oder ein Test einer Variablen mit dem Wert 1 ausgelassen. Durch die Modifikation ist sichergestellt, dass im Produktgraphen das Paar $(0\text{-Senke}, 0\text{-Senke})$ erreicht wird. Dies ist eine 0-Senke, da $\otimes$ 0-erhaltend ist.

Es bleibt die Synthese mit nicht 0-erhaltenden Operatoren $\otimes$. Dann ist aber die Negation von $\otimes$ 0-erhaltend, sodass wir erst die Synthese mit der Negation von $\otimes$ ausführen und anschließend die dargestellte Funktion negieren können. Eine Möglichkeit für die Ausführung der Negation besteht darin, das ZBDD durch Anwenden der inversen Eliminationsregel in ein vollständiges OBDD umzuformen, die dargestellte Funktion (durch Vertauschen der 0- und 1-Senke) zu negieren, das entstandenen OBDD wieder als ZBDD zu interpretieren und zu reduzieren. Dabei kann sich die Größe höchstens um den Faktor $n+1$ vergrößern; am Beispiel der Nullfunktion haben wir bereits gesehen, dass dieser Faktor auch möglich ist. Wir erhalten also:

Satz 9.3: Sei $h = f \otimes g$. Falls $\otimes$ 0-erhaltend ist, gilt

$$\text{ZBDD}(h, \pi) \leq \text{ZBDD}(f, \pi)\text{ZBDD}(g, \pi),$$

anderenfalls

$$\text{ZBDD}(h, \pi) \leq \text{ZBDD}(f, \pi)\text{ZBDD}(g, \pi)(n + 1) + 1.$$

Der Summand $+1$ ist für den Fall $h = \overline{\bar{x}_1 \wedge \cdots \wedge \bar{x}_n}$ vorgesehen, da dann beim Negieren eine 0-Senke erzeugt werden muss.

Als letzte Operation bleibt das Ersetzen durch Konstanten. O.B.d.A. sei $x_1, \ldots, x_n$ die Variablenordnung und sei x_i die Variable, die durch c ersetzt werden soll. Am einfachsten ist dies möglich, wenn auf jedem Berechnungspfad ein x_i-Knoten vorhanden ist, dann genügt es, den Zeiger von den $\bar{c}$-Nachfolgern auf den jeweiligen c-Nachfolger umzulenken. Wenn es Berechnungspfade ohne x_i-Knoten gibt, wollen wir wieder die inverse Eliminationsregel anwenden: Für jeden Knoten v, der nach der x_i-Ebene getestet wird und für den es einen Zeiger von Knoten oberhalb der x_i-Ebene gibt, fügen wir einen x_i-Knoten ein, dessen 1-Nachfolger die 0-Senke ist und dessen 0-Nachfolger v ist, und setzen die Zeiger von oberhalb der x_i-Ebene auf den eingefügten Knoten um.

Wenn nun $c = 1$ ist, zeigen anschließend beide Nachfolger der zusätzlich eingefügten Knoten auf die 0-Senke, sodass diese Knoten wieder entfernt werden können. In diesem Fall vergrößert sich das ZBDD also nicht. Dagegen ist es im Fall $c = 0$ möglich, dass sich das ZBDD vergrößert. Falls oberhalb der x_i-Ebene s Knoten und unterhalb t Knoten vorhanden sind, ist die Anzahl der eingefügten Knoten, die nicht wieder entfernt werden, durch $\min\{s + 1, t - 1\}$ beschränkt; die Beschränkung durch $s + 1$ ergibt sich daraus, dass es höchstens $s + 1$ Kanten geben kann, die die x_i-Ebene kreuzen, der Term $t - 1$ ergibt sich, da der für die 0-Senke eingefügte x_i-Knoten auf jeden Fall wieder gelöscht wird. Also ist $s + t \leq \text{ZBDD}(f, \pi)$ und die Anzahl hinzugefügter Knoten ist durch $\min\{s + 1, t - 1\} \leq (s + t)/2 \leq \text{ZBDD}(f, \pi)/2$ beschränkt. Also ist die ZBDD-Größe nach Ersetzen einer Variablen durch 0 durch $(3/2)\text{ZBDD}(f, \pi)$ beschränkt.

Abschließend wollen wir zeigen, dass auch diese Schranke scharf ist. Dazu betrachten wir das ZBDD in Abbildung 16, wobei die Variablenordnung $x_1, \ldots, x_m, y, z_1, \ldots, z_m$ ist. Man sieht leicht, dass bei der Ersetzung von y durch 0 insgesamt $(m + 1)$ y-Knoten einzufügen sind, die nicht eliminiert werden können.

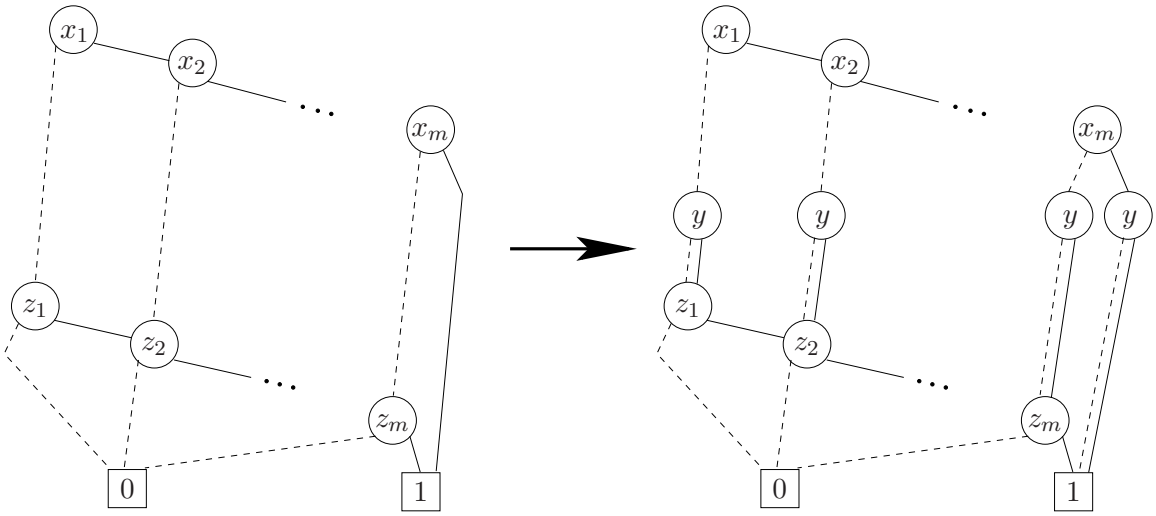


Abbildung 16: Ein Beispiel für ein ZBDD, bei dem die Ersetzung von y durch 0 zu einer Vergrößerung um den Faktor $3/2$ führt

10 Ordered Functional Decision Diagrams

In OBDDs werden an den inneren Knoten die dargestellten Funktionen gemäß der Shannon-Zerlegung

$$f = \bar{x}_i f_{|x_i=0} \vee x_i f_{|x_i=1}$$

zerlegt. Neben der Shannon-Zerlegung gibt es auch die sogenannte Reed-Muller-Zerlegung (auch positive Davio-Zerlegung genannt):

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

Die Korrektheit der Zerlegung ist leicht nachzurechnen. Die Shannon- und die Reed-Muller-Zerlegung haben als weitere gemeinsame Eigenschaft, dass die Zerlegung eindeutig ist, also dass man für die Resultate $f_{|x_i=0}$ und $f_{|x_i=1}$ bzw. $f_{|x_i=0}$ und $f_{|x_i=0} \oplus f_{|x_i=1}$ keine Wahlmöglichkeiten hat. Wir werden später auch eine Zerlegung kennenlernen, wo das nicht der Fall ist.

Die Idee der Ordered Functional Decision Diagrams (OFDDs) besteht nun darin, an dem 0-Nachfolger des Knotens für f die Funktion $f_{|x_i=0}$ und am 1-Nachfolger die Funktion $f_{|x_i=0} \oplus f_{|x_i=1}$ darzustellen. Ein Beispiel dafür zeigt Abbildung 17, wobei wieder an den Knoten die dort berechneten Funktionen angegeben sind. Die Funktion am 1-Nachfolger der Quelle ergibt sich als $(x_1x_2 \vee x_3)_{|x_1=0} \oplus (x_1x_2 \vee x_3)_{|x_1=1}$.

Syntaktisch sehen OFDDs also genauso wie OBDDs aus. Die Auswertung ist allerdings anders, wir müssen die Reed-Muller-Zerlegung auswerten: Wenn ein x_i-Knoten betrachtet wird und $x_i = 0$ gilt, ist der Funktionswert der am 0-Nachfolger berechnete Wert; wenn $x_i = 1$ gilt, ist der Funktionswert das Parity der Werte, die am 0- und am 1-Nachfolger berechnet werden. Wir sollten daher die OFDDs bottom-up auswerten und für jeden Knoten den berechneten Funktionswert bestimmen und speichern. Dies ist offensichtlich in linearer Zeit $O(|G|)$ möglich, aber dennoch aufwendiger als bei OBDDs.

Wenn wir das OFDD aus Abbildung 17 für $x_1 = x_2 = x_3 = 1$ auswerten wollen, durchlaufen wir die Knoten von unten nach oben. Am linken x_3-Knoten wird eine 1 berechnet (als Parity von 0 und 1), am rechten x_3-Knoten eine 0 (als Parity von 1 und 1). Damit wird auch am x_2-Knoten eine 0 berechnet und an der Quelle eine 1. Also ist der Funktionswert 1.

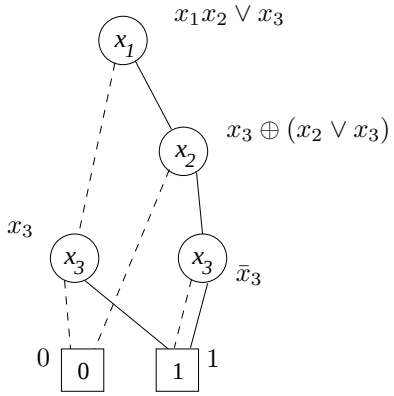


Abbildung 17: Ein Beispiel für ein OFBDD

Aus der Reed-Muller-Zerlegung wollen wir herleiten, welche Reduktionsregeln wir auf OFDDs anwenden dürfen. Bei der Verschmelzungsregel ist klar, dass nur Knoten verschmolzen werden, die dieselbe Funktion darstellen und dass auch nach der Verschmelzung dieselbe Funktion dargestellt wird. Für die Eliminationsregel überlegen wir, dass die an einem x_i-Knoten dargestellte Funktion f genau dann nicht von x_i essentiell abhängt, wenn in der Zerlegung $f_{|x_i=0} \oplus x_i(f_{|x_i=0} \oplus f_{|x_i=1})$ der Term in der Klammer gleich Null ist. Dies bedeutet, dass die 1-Kante zur 0-Senke zeigt. In diesem Fall wird an dem Knoten die Funktion $f_{|x_i=0}$ dargestellt, sodass wir die eingehenden Kanten zum 0-Nachfolger umsetzen dürfen. Die Eliminationsregel für OFDDs ist also dieselbe wie für ZBDDs. Hier aber eine Warnung: Die von einem ZBDD berechnete Funktion hängt von einer Variablen x_i auch dann essentiell ab, wenn es keine x_i-Knoten gibt, da das ZBDD die ausgelassenen Variablen implizit auf 0 testet. In einem OFDD ohne x_i-Knoten hängt die dargestellte Funktion dagegen nicht von x_i essentiell ab.

Eine weitere Variante der Auswertung ist die Folgende: Wir starten die Berechnung an der Quelle. Wenn ein x_i-Knoten mit $x_i = 0$ erreicht wird, wird die ausgehende 0-Kante aktiv und die Berechnung am 0-Nachfolger fortgesetzt. Wenn dagegen $x_i = 1$ ist, werden beide ausgehenden Kanten aktiv und die Berechnung an beiden Nachfolgern fortgesetzt. Insgesamt wird eine 1 berechnet, wenn die 1-Senke über ungerade viele aktive Pfade erreicht wird. Diese Art der Auswertung führt i.A. zu exponentieller Rechenzeit, ist aber für das Verständnis des sogenannten τ-Operators nützlich, den wir gleich untersuchen wollen.

Die Äquivalenz zu der o.g. Auswertungsregel erhält man mit Induktion über eine umgekehrte topologische Ordnung der Knoten. Offensichtlich ergeben beide Auswertungsregeln an den Senken dasselbe Ergebnis. Betrachte nun einen x_i-Knoten v mit $x_i = 0$. Nach der ersten Regel ist der an v berechnete Funktionswert gleich dem Funktionswert, der am 0-Nachfolger v_0 berechnet wird. Dieser ist nach Induktionsvoraussetzung genau dann 1, wenn die Anzahl aktiver Pfade von v_0 zur 1-Senke ungerade ist, was wiederum äquivalent dazu ist, dass die Anzahl aktiver Pfade von v zur 1-Senke ungerade ist. Also wird auch an v der korrekte Funktionswert berechnet.

Der zweite Fall ist $x_i = 1$. Nach der ersten Regel ist das Parity der Funktionswerte an den Nachfolgern v_0 und v_1 zu bilden. Nach Induktionsvoraussetzung wird an v_i genau dann eine 1 berechnet, wenn die Anzahl aktiver Pfade von v_i zur 1-Senke ungerade ist. Also wird auch an v genau dann eine 1 berechnet, wenn die Anzahl aktiver Pfade von v zur 1-Senke ungerade ist. Insgesamt liefern also beide Arten der Auswertung denselben Funktionswert.

Wir definieren jetzt den τ-Operator $\tau : B_n \to B_n$ durch

$$(\tau f)(a) := \bigoplus_{b|b\leq a} f(b).$$

Hierbei ist das $\leq$ koordinatenweise zu verstehen, also gilt genau dann
$(a_1, \ldots, a_n) \leq (b_1, \ldots, b_n)$, wenn $\forall i : a_i \leq b_i$.

Die Bedeutung des τ-Operators für das Verständnis von OFDDs ergibt sich aus dem folgenden Lemma.

Lemma 10.1: Sei G ein vollständiges OBDD für f. Dann stellt G als OFDD interpretiert die Funktion τf dar.

Beweis: Sei a eine Eingabe. Wenn wir G als OFDD interpretieren, sind für a genau die Pfade aktiv, die im Sinne der OBDD-Auswertung für die Eingaben b mit $b \leq a$ aktiv sind. Der Funktionswert ist das Parity der Werte der über diese Pfade erreichten Senken und dies ist auch genau das Resultat von $(\tau f)(a)$. □

Anmerkung: Die Aussage gilt nicht notwendigerweise für unvollständige OBDDs bzw. OFDDs. Ein Beispiel, das dies zeigt, ist das OBDD/OFDD, das nur aus einer 1-Senke besteht. Offensichtlich repräsentiert dieser Graph sowohl als OBDD als auch als OFDD die 1-Funktion. Dagegen ist $(\tau 1)(0, \ldots, 0, 1) = 0$.

Wir zeigen nun einfache Eigenschaften den τ-Operators.

Lemma 10.2:

1. Für alle Funktionen $f, g \in B_n$ gilt $\tau(f \oplus g) = \tau f \oplus \tau g$.
2. Für alle Funktionen $f \in B_n$ gilt $\tau\tau f = f$.
3. τ ist bijektiv auf B_n.

Beweis: Die erste Eigenschaft folgt direkt daraus, dass $\oplus$ assoziativ ist. Für den Nachweis der zweiten Eigenschaft setzen wir zweimal die Definition ein und nutzen die erste Aussage aus:

$$(\tau\tau f)(a) = \tau\left(\bigoplus_{b|b\leq a} f(b)\right) = \bigoplus_{b|b\leq a} (\tau f)(b) = \bigoplus_{b|b\leq a} \bigoplus_{c|c\leq b} f(c) = \bigoplus_{b,c|c\leq b\leq a} f(c).$$

Wenn sich a und c an i Positionen unterscheiden, ist die Anzahl der b mit $c \leq b \leq a$ gleich 2^i. In diesem Fall wird $f(c)$ 2^i-mal modulo 2 addiert, sodass für $i \geq 1$ das Resultat gleich 0 ist. Also genügt es, den Fall $c = a$ zu betrachten, und dann ergibt der letzte Ausdruck der abgesetzten Formel $f(a)$. Schließlich ist τ selbstinvers und auf B_n vollständig definiert, also auch bijektiv. □

Da τ sein eigenes Inverses ist, folgt sofort die umgekehrte Aussage des ersten Lemmas.

Folgerung 10.3: Sei G ein vollständiges OFDD für f. Dann stellt G als OBDD interpretiert die Funktion τf dar.

Da wir auf OFDDs dieselben Reduktionsregeln wie auf ZBDDs anwenden dürfen, gilt für unvollständige OFDDs/ZBDDs die folgende Aussage.

Folgerung 10.4: Sei G ein OFDD (ZBDD) für f. Dann stellt G als ZBDD (OFDD) interpretiert die Funktion τf dar.

Aus den Betrachtungen zum τ-Operator erhalten wir auch einen Struktursatz für OFDDs. Sei f auf den Variablen $x_1, \ldots, x_n$ definiert und sei $S \subseteq \{1, \ldots, i-1\}$. Dann bezeichnet $f_{i,S}$ die Funktion, die sich als Parity aller Funktionen $f_{|x_1=a_1,\ldots,x_{i-1}=a_{i-1}}$ mit $a_j = 0$ für $j \notin S$ und $a_j \in \{0,1\}$ für $j \in S$ ergibt.

Satz 10.5: Es gibt bis auf Isomorphie genau ein minimales OFDD für die Funktion f und die Variablenordnung $x_1, \ldots, x_n$. Dieses enthält für jede Funktion $f_{i,S}$, $S \subseteq \{1, \ldots, i-1\}$, die von x_i essentiell abhängt, genau einen x_i-Knoten. Das minimale OFDD für f und $x_1, \ldots, x_n$ erhalten wir aus jedem OFDD für f und $x_1, \ldots, x_n$, indem wir die Reduktionsregeln (Verschmelzungsregel, Eliminationsregel für ZBDDs/OFDDs) so lange anwenden, wie dies möglich ist.

Die Struktur des Beweises folgt wieder dem entsprechenden Satz für OBDDs, sodass wir hier darauf verzichten wollen.

Die Begriff nice, ugly, usw. können wir auch für OFDDs definieren. Es sollte klar sein, dass daraus, dass eine Funktion f diese Eigenschaft für OBDDs hat, folgt, dass die Funktion τf dieselbe Eigenschaft für OFDDs hat. Wir wollen dennoch einige Beispielfunktionen betrachten.

Lemma 10.6: Wenn f symmetrisch ist, ist auch τf symmetrisch, und die Größe von OFDDs für f ist $O(n^2)$.

Beweis: Wenn die Eingaben a und a' genau j Einsen enthalten, stimmt die Anzahl der Eingaben b und b' mit jeweils i Einsen und $b \leq a$ bzw. $b' \leq a'$ überein. Hieraus folgt

$$\tau f(a) = \bigoplus_{b|b \leq a} f(b) = \bigoplus_{b'|b' \leq a'} f(b') = \tau f(a'),$$

also ist τf symmetrisch. Wir haben gesehen, dass vollständige OBDDs für symmetrische Funktionen die Größe $O(n^2)$ haben, also gilt dies auch für vollständige OFDDs. □

Wir betrachten nun zwei Funktionen auf ungerichteten Graphen mit n Knoten. Die Eingabe für diese Funktionen besteht aus $N = \binom{n}{2}$ Variablen, die für jede Kante angeben, ob sie vorhanden ist oder nicht. Die Funktion $1cl_{n,3}$ entscheidet, ob der Graph aus genau einem Dreieck besteht (und keine weiteren Kanten hat). Die Funktion $\oplus cl_{n,3}$ entscheidet, ob es im Graphen eine ungerade Anzahl von Dreiecken gibt.

Es ist leicht zu zeigen, dass die Funktion $1cl_{n,3}$ für jede Variablenordnung (vollständige) OBDDs der Größe $O(N^2) = O(n^4)$ hat. Für die Funktion $\oplus cl_{n,3}$ kann man die untere Schranke $2^{\Omega(N)}$ für die OBDD-Größe zeigen. Der Beweis der unteren Schranke $2^{\Omega(n)} = 2^{\Omega(N^{1/2})}$ ist viel einfacher und enthält gegenüber dem entsprechenden Beweis für ISA keine wesentlichen neuen Ideen (Übungsaufgabe). Um Schranken für die OFDD-Größen dieser Funktionen zu erhalten, genügt uns nun die folgende einfache Aussage.

Lemma 10.7: $\tau(1cl_{n,3}) = \oplus cl_{n,3}$.

Beweis: $1cl_{n,3}$ entscheidet, ob ein Graph aus genau einem Dreieck besteht. Sei nun a eine Eingabe, also ein Graph. Bei der Auswertung von $\tau(1cl_{n,3})(a) = \bigoplus_{b|b\leq a} 1cl_{n,3}(b)$, betrachten wir alle Graphen b, die wir aus a erhalten können, indem wir auf alle möglichen Weisen Kanten aus a entfernen. Die Funktion $1cl_{n,3}$ berechnet nur auf Graphen mit 3 Kanten eine 1, und dies auch nur, wenn diese Kanten ein Dreieck bilden. Für jedes Dreieck in a erhalten wir durch Entfernen von Kanten genau einen solchen Graphen. Also gibt $\tau(1cl_{n,3})(a)$ an, ob der Graph a eine ungerade Anzahl von Dreiecken enthält. □

Folgerung 10.8:

1. Die OFDD-Größe von $1cl_{n,3}$ ist $2^{\Omega(N)}$.
2. Die OFDD-Größe von $\oplus cl_{n,3}$ ist $O(N^2)$.

Also gibt es Funktionen mit polynomieller OBDD-Größe und exponentieller OFDD-Größe und auch Funktionen mit polynomieller OFDD-Größe und exponentieller OBDD-Größe.

Die OFDDs wurden als alternative Datenstruktur für boolesche Funktionen vorgeschlagen. Daher wollen wir die Komplexität der wichtigsten Operationen untersuchen.

Ohne Beweis erwähnen wir nur, dass das Variablenordnungsproblem und das Problem Erfüllbarkeit-Anzahl NP-schwer sind. Dagegen ist das Problem Erfüllbarkeit leicht zu lösen: Wegen des Struktursatzes ist eine durch ein reduziertes OFDD dargestellte Funktion erfüllbar, wenn es nicht nur aus der 0-Senke besteht. Als weitere Operationen betrachten wir die Synthese und das Ersetzen durch Konstanten.

Wir beginnen mit der Synthese für die Operation $\oplus$. Hierfür kann die Produktgraphkonstruktion von den OBDDs übertragen werden. Seien u und v x_i-Knoten aus G_f bzw. G_g, an denen die Funktionen f_u bzw. g_v berechnet werden. Dann soll am Knoten (u, v) die Funktion $f_u \oplus g_v$ berechnet werden. Seien nun $f_{u,0} = f_{u|x_i=0}$ und $f_{u,1} = f_{u|x_i=0} \oplus f_{u|x_i=1}$ die an den Nachfolgern von u berechneten Funktionen und seien $g_{v,0}$ und $g_{v,1}$ analog definiert. Es gilt

$$f_u \oplus g_v = (f_{u,0} \oplus x_i f_{u,1}) \oplus (g_{v,0} \oplus x_i g_{v,1}) = (f_{u,0} \oplus g_{v,0}) \oplus x_i(f_{u,1} \oplus g_{v,1}).$$

D.h., wir erhalten auch den c-Nachfolger von (u, v) als Paar der c-Nachfolger von u und v. Auch die übrigen Fälle des Synthesealgorithmus für OBDDs lassen sich leicht übertragen. Insgesamt erhalten wir, dass ein OFDD für $f \oplus g$ leicht in Zeit $O(|G_f||G_g|)$ berechnet werden kann und auch die Größe $O(|G_f||G_g|)$ hat.

Anders ist die Situation bei der Synthese mit der Operation $\wedge$. Als Beispiel betrachten wir die Funktionen f und g, die folgendermaßen definiert sind. Die Funktion f ist die Funktion $\oplus cl_{n,3}$, die, wie oben gezeigt, OFDDs der Größe $O(N^2)$ hat. Die Funktion g berechnet eine 1, wenn es höchstens 3 Einsen in der Eingabe der Länge N gibt. Dann ist g symmetrisch und hat damit OFDDs der Größe $O(N^2)$. Man sieht nun leicht, dass $f \wedge g$ gleich der Funktion $1cl_{n,3}$ ist, also nur OFDDs exponentieller Größe hat. Also kann bei der $\wedge$-Synthese die Größe exponentiell wachsen. Woran liegt das?

Wir versuchen, die $\wedge$-Synthese der an den Knoten u und v berechneten Funktionen auf die an den Nachfolgern berechneten Funktionen zurückzuführen:

$$\begin{aligned} f_u \wedge g_v &= (f_{u,0} \oplus x_i f_{u,1}) \wedge (g_{v,0} \oplus x_i g_{v,1}) \\ &= (f_{u,0} g_{v,0}) \oplus x_i (f_{u,0} g_{v,1} \oplus f_{u,1} g_{v,0} \oplus f_{u,1} g_{v,1}) \end{aligned}$$

Die Berechnung von OFDDs für $(f_{u,0}g_{v,0})$ und $(f_{u,1}g_{v,1})$ können wir wie oben mit der Produktgraphkonstruktion durchführen. Allerdings müssen wir auch OFDDs für $(f_{u,0}g_{v,1})$ und $(f_{u,1}g_{v,0})$ bestimmen, was ebenfalls mit der Produktgraphkonstruktion möglich ist, und wir müssen die drei Funktion mit $\oplus$ verknüpfen. Für die daraus resultierende Funktion gibt es möglicherweise keinen Knoten im Produktgraphen; ein solcher Knoten kann aber mit der $\oplus$-Synthese berechnet werden, wobei aber wiederum viele Knoten entstehen können, die im Produktgraphen nicht enthalten sind. Da diese Extra-Aufrufe der Synthese u.U. auf jeder Rekursionsebene durchgeführt werden, kann man die Rechenzeit nicht mehr durch ein Polynom beschränken; dass wirklich exponentielle Rechenzeit möglich ist, zeigt das Beispiel von oben.

Das Ersetzen von x_i durch die Konstante 0 ist wieder einfach; es genügt, die Zeiger, die auf x_i-Knoten zeigen, auf den 0-Nachfolger umzusetzen. Für das Ersetzen von x_i durch die Konstante 1 wollen wir die Zeiger auf x_i-Knoten v auf Knoten umsetzen, die die Funktion $f_{v|x_i=1}$ berechnen; diese gibt es aber nicht unbedingt im OFDD. Da es Knoten für $f_{v|x_i=0}$ und für $f_{v|x_i=0} \oplus f_{v|x_i=1}$ gibt, können wir solche Knoten mit der $\oplus$-Synthese berechnen. Dabei kann sich aber die Größe quadrieren. Wir zeigen nun, dass wir bei iteriertem Ersetzen von Variablen durch die Konstante 1 sogar eine exponentielle Vergrößerung erhalten können.

Sei $N = \binom{n}{2}$ und $M = \lceil \log \binom{n}{3} \rceil$. Die betrachtete Funktion ist auf $N + M$ Variablen $x_1, \ldots, x_N$ und $y_1, \ldots, y_M$ definiert. Wir beschreiben die Funktion durch ihr OFDD. Das OFDD beginnt mit einem vollständigen Baum der Tiefe M, in dem die y-Variablen getestet werden. Die Blätter dieses Baums assoziieren wir mit den $\binom{n}{3}$ dreielementigen Mengen $\{i, j, k\} \subseteq \{1, \ldots, n\}$, wobei eventuell Blätter übrig bleiben. An dem Blatt für $\{i, j, k\}$ testen wir, ob der durch die x-Variablen beschriebene Graph genau die Kanten $\{i, j\}$, $\{i, k\}$ und $\{j, k\}$ (und sonst keine weiteren) hat. Dies ist eine UND-Verknüpfung von Literalen, die mit OFDDs in Größe $O(N)$ (linear in der Anzahl der Variablen) zu realisieren ist. Die eventuell übrigen Blätter des Baumes sind 0-Senken. Die OFDD-Größe ist also $O(n^3 N) = O(n^5) = O((N + M)^{5/2})$.

Wir ersetzen nun alle y-Variablen durch 1. Eine Funktion an den Blättern des Baumes kann nur dann eine 1 berechnen, wenn der Graph aus genau einem Dreieck und keinen weiteren Kanten besteht. Damit kann höchstens eine dieser Funktionen eine 1 berechnen. Durch die Ersetzung aller y-Variablen durch 1 erhalten wir das Parity aller dieser Funktionen, das mit dem ODER übereinstimmt. Also entsteht durch die Konstantsetzung ein OFDD für die Funktion $1cl_{n,3}$, das exponentielle Größe hat.

Eine Variante von OFDDs sind die sogenannten OKFDDs (Ordered Kronecker Functional Decision Diagrams). Hier wird neben der Variablenordnung für jede Variable x_i angegeben, ob die Zerlegung an den x_i-Knoten die Shannon-Zerlegung, die Reed-Muller-Zerlegung (positive Davio-Zerlegung) oder die negative Davio-Zerlegung ist. Dabei ist die negative Davio-Zerlegung die Zerlegung $f = f_{|x_i=1} \oplus \bar{x}_i (f_{|x_i=0} \oplus f_{|x_i=1})$. Die einzige neue Idee hierbei

besteht darin, dass man für jede Variable die Art der Zerlegung angeben kann. Dies führt natürlich zu dem Problem zu entscheiden, welcher Zerlegungstyp für die einzelnen Variablen der bessere ist. Da man mehr Wahlmöglichkeiten hat, ist die Annahme naheliegend und auch leicht zu beweisen, dass man mit OKFDDs mehr Funktionen kompakt darstellen kann als mit OBDDs und OFDDs. Andererseits kombinieren sich natürlich auch die Nachteile der OFDDs und OBDDs.

11 Parity-OBDDs

Parity-OBDDs ($\oplus$OBDDs) sind eine Erweiterung von OBDDs und OFDDs, die helfen, einige der Nachteile von OFDDs zu vermeiden. Wir beginnen mit der Definition.

Ein $\oplus$OBDD ist ein gerichteter azyklischer Graph, der aus inneren Knoten und einer 1-Senke besteht. Jeder innere Knoten ist mit einer Variablen markiert und hat eine beliebige Anzahl von 0- und 1-Nachfolgern. Weiterhin gibt es eine Quelle und die Variablenordnungsbedingung wie bei OBDDs. Für eine Eingabe $(a_1, \ldots, a_n)$ wird ein $\oplus$OBDD folgendermaßen ausgewertet: An den x_i-Knoten werden die ausgehenden a_i-Kanten aktiv. Das $\oplus$OBDD berechnet den Funktionswert 1, wenn die Anzahl der Pfade von der Quelle zur 1-Senke, die nur aus aktiven Kanten bestehen, ungerade ist.

Da wir nur die Pfade zur 1-Senke zählen und nicht verlangen, dass jeder Knoten ausgehende 0- und 1-Kanten hat, ist eine 0-Senke überflüssig. Ein Beispiel für ein $\oplus$OBDD ist in Abbildung 18 gezeigt.

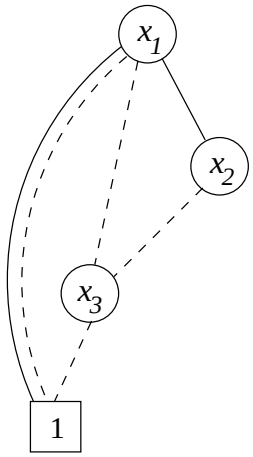


Abbildung 18: Ein Beispiel für ein Parity-OBDD

Für die Eingabe, die nur aus Nullen besteht, werden genau die gestrichelten Kanten aktiv. Da es zwei Pfade von der Quelle zur 1-Senke gibt, die nur aus gestrichelten Kanten bestehen, wird eine Null berechnet.

Da das Zählen von Pfaden i.A. zu exponentieller Rechenzeit führt, bietet es sich an, wie bei den OFDDs die Auswertung bottom-up vorzunehmen und zu den einzelnen Knoten den Wert der dort berechneten Funktion zu speichern. An der 1-Senke wird der Wert 1 berechnet. Der Wert an einem x_i-Knoten v ergibt sich als das Parity der Werte an den a_i-Nachfolgern des Knotens. Die Korrektheit ist einfach einzusehen: Die Anzahl der Pfade von v zur 1-Senke ist ungerade, wenn es ungerade viele Pfade von allen a_i-Nachfolgern zusammen zur 1-Senke gibt. Für die Eingabe, die nur aus Nullen besteht, wird dann in dem Beispiel am x_3-Knoten eine 1 berechnet, am x_2-Knoten ebenfalls eine 1 und am x_1-Knoten eine 0.

Die beschriebene Auswertung entspricht der folgenden Zerlegung der Funktion: Wenn der x_i-Knoten v die 0-Nachfolger $u_1, \ldots, u_l$ und die 1-Nachfolger $w_1, \ldots, w_r$ hat, wird an v die Funktion

$$f_v = \bar{x}_i(f_{u_1} \oplus \cdots \oplus f_{u_l}) \vee x_i(f_{w_1} \oplus \cdots \oplus f_{w_r})$$

berechnet. Anders als die Shannon-Zerlegung oder die Reed-Muller-Zerlegung ist diese Zerlegung i.A. nicht eindeutig. Dies zeigt sich auch daran, dass die Shannon-Zerlegung und die Reed-Muller-Zerlegung als Spezialfälle enthalten sind; die Shannon-Zerlegung ist der Spezialfall $l = r = 1$, die Reed-Muller-Zerlegung der Spezialfall $l = 1$ und $r = 2$ mit der weiteren Einschränkung, dass $f_{u_1} = f_{w_1}$. Wir können hieraus schließen, dass wir sowohl OBDDs als auch OFDDs in $\oplus$OBDDs umwandeln können, ohne die Anzahl der Knoten zu vergrößern. Für OBDDs ist dies einfach einzusehen, es genügt die 0-Senke und alle eingehenden Kanten wegzulassen. Das OBDD berechnet den Funktionswert 1, wenn es einen Pfad von der Quelle zur 0-Senke gibt. Da es auch nicht mehr als einen solchen Pfad geben kann, ist dies äquivalent dazu, dass es ungerade viele solche Pfade gibt.

In einem OFDD ersetzen wir jeden inneren Knoten v mit den Nachfolgern v_0 und v_1 durch einen Knoten v mit dem 0-Nachfolger v_0 und den 1-Nachfolgern v_0 und v_1. Weiterhin lassen wir die 0-Senke und ihre eingehenden Kanten weg. Wir erhalten dann ein $\oplus$OBDD für dieselbe Funktion, da wir OFDDs auch über das Zählen von Pfaden von der Quelle zur 1-Senke auswerten können, wobei eine Eingabe mit $a_i = 1$ für x_i-Knoten sowohl die ausgehende 0-Kante als auch die ausgehende 1-Kante aktiviert. Also berechnet das konstruierte $\oplus$OBDD dieselbe Funktion wie das gegebene OFDD.

Mit der o.g. Zerlegung können wir für das obige Beispiel die dargestellte Funktion berechnen. Am x_3-Knoten wird $\bar{x}_3$ berechnet und am x_2-Knoten $\bar{x}_2\bar{x}_3$. Wir erhalten für den x_1-Knoten die Funktion

$$\begin{aligned} \bar{x}_1(\bar{x}_3 \oplus 1) \vee x_1(\bar{x}_2\bar{x}_3 \oplus 1) &= \bar{x}_1 x_3 \vee x_1(x_2 \vee x_3) \\ &= \bar{x}_1 x_3 \vee x_1 x_2 \vee x_1 x_3 \\ &= x_1 x_2 \vee x_3. \end{aligned}$$

Wir haben bereits früher ein OBDD für diese Funktion und dieselbe Variablenordnung gesehen, das ebenfalls nur 3 innere Knoten hat und anders aussah. Wir können daraus schließen, dass minimale $\oplus$OBDDs für eine feste Funktion und eine feste Variablenordnung i.A. nicht bis auf Isomorphie eindeutig sind, wobei hier minimal bezüglich der Anzahl der Knoten gemeint ist.

Ein weiteres Beispiel, das den Unterschied zu OBDDs deutlich macht, ist die Funktion INDEX. Wir benutzen hier eine Variablenordnung, in der die Datenvariablen $x_0, \ldots, x_{n-1}$ *vor* den Adressvariablen $a_0, \ldots, a_{k-1}$ angeordnet sind. Wir haben bereits gezeigt, dass OBDDs für derartige Variablenordnungen exponentielle Größe haben. Ein $\oplus$OBDD mit der Größe $O(n \log n)$ ist in Abbildung 19 gezeigt. Dieses $\oplus$OBDD ist nicht knotenminimal, allerdings wird so die Arbeitsweise vielleicht klarer. Der einzige Knoten, wo es mehr als eine ausgehende 0- oder 1-Kante gibt, ist die Quelle. Insgesamt gibt es die folgenden n Möglichkeiten, die 1-Senke von der Quelle zu erreichen: Die 0-te Möglichkeit besteht darin, den linken Pfad zu durchlaufen, auf dem nur x_0 und die Adressvariablen getestet werden, die i-te Möglichkeit ($1 \leq i \leq n-1$) darin, die Variable x_i zu testen und anschließend die Adressvariablen. Die i-te Möglichkeit führt genau dann zu einem Pfad aus aktiven Kanten zur 1-Senke, wenn $x_i = 1$ und die Adressvariablen den Wert i codieren. Dies sind genau die Fälle, wo eine 1 berechnet werden muss. Da jeweils nur einer dieser Fälle eintreten kann (die Adressvariablen können nicht zugleich zwei verschiedene Werte codieren), gibt es genau einen und damit ungerade viele akzeptierende Pfade. Also wird INDEX berechnet.

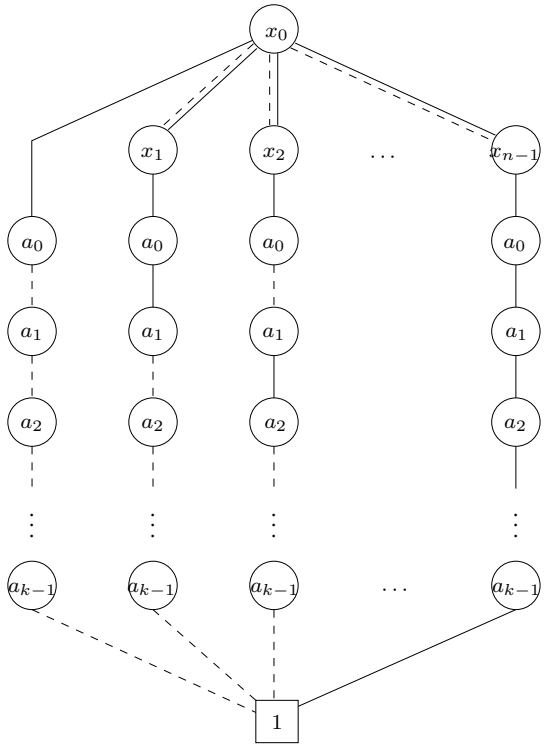


Abbildung 19: Ein Parity-OBDD für INDEX_n

Eine andere Sichtweise besteht darin, dass das $\oplus$OBDD an der Quelle den durch die Adressvariablen codierten Wert *rät* und anschließend *verifiziert*, ob die zugehörige Datenvariable den Wert 1 hat und der geratene Wert für die Adressvariablen richtig war. Wir haben hier bewusst die Ausdrucksweise aus GTI für nichtdeterministische Berechnungsmodelle gewählt, um die Ähnlichkeit dazu deutlich zu machen. Die einzige Besonderheit besteht hier darin, dass von den zu ratenden Werten nur *genau einer* richtig ist, sodass die Aussagen, dass (mindestens) ein akzeptierender Pfad existiert und dass ungerade viele akzeptierende Pfade existieren, äquivalent sind. Diesen Zusammenhang mit dem Nichtdeterminismus werden wir in Abschnitt 12 noch weiter untersuchen.

Da die Anzahl der ausgehenden Kanten für jeden Knoten nicht notwendigerweise konstant ist, wird der Speicherbedarf von $\oplus$OBDDs durch die Anzahl der Kanten bestimmt. Die Frage nach der Komplexität der Minimierung der Zahl der Kanten in $\oplus$OBDDs ist bislang noch offen und zwar sowohl für den Fall einer fest vorgegebenen Variablenordnung als auch für die Situation, dass auch die Variablenordnung gewählt werden muss. Ebenso ist die Komplexität der Minimierung der Knotenzahl durch Wahl der Variablenordnung noch offen. Wir werden daher nur die Minimierung der Knotenzahl bei vorgegebener Variablenordnung behandeln.

Wie gesagt, ist die Komplexität des Variablenordnungsproblems offen. Trotz der Ähnlichkeit zu den OBDDs sind aber Unterschiede bekannt. Beispielsweise ändert sich weder die Anzahl der Knoten noch die Anzahl der Kanten von vollständigen $\oplus$OBDDs, wenn man die Reihenfolge der Variablen umdreht; der Beweis ist eine einfache Übungsaufgabe. Bei OBDDs gilt dies nicht; ein Gegenbeispiel ist die INDEX-Funktion. Weiterhin ist bekannt, dass es für $\oplus$OBDDs keine Funktionen mit den Eigenschaften almost nice und ambiguous gibt.

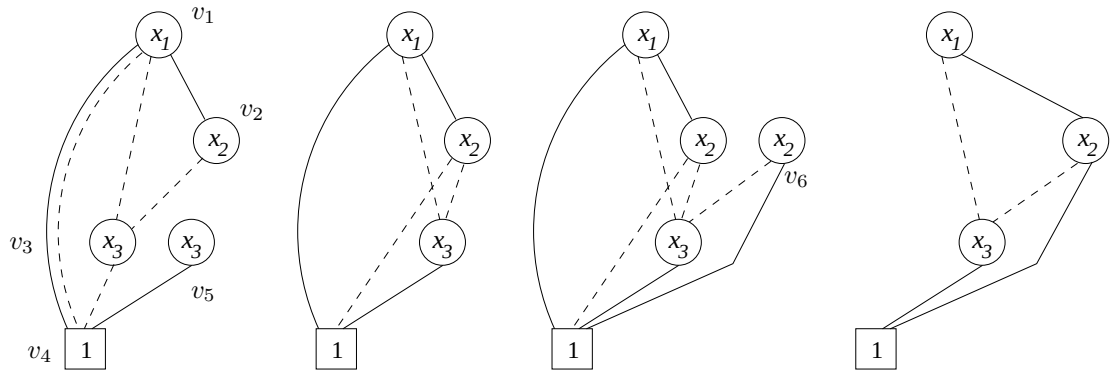


Abbildung 20: Beispiele für Hinzufügen und Entfernen von Linearkombinationen

Wir wollen nun zwei elementare Operationen für das Verändern von $\oplus$OBDDs kennenlernen, die wir später dazu benutzen, um die Anzahl der Knoten in einem $\oplus$OBDD zu minimieren. Seien $v_1, \ldots, v_r$ x_i-Knoten eines $\oplus$OBDDs, und sei $x_1, \ldots, x_n$ die Variablenordnung. Unter dem *Hinzufügen von Linearkombinationen* verstehen wir, einen x_i-Knoten v für die Funktion $f_{v_1} \oplus \cdots \oplus f_{v_r}$ zu erzeugen. Dazu markieren wir v mit x_i, und die c-Nachfolger von v sind die Knoten, die ungerade häufig c-Nachfolger von den Knoten $v_1, \ldots, v_r$ sind. Dann ist einfach einzusehen, dass an v die gewünschte Funktion $f_{v_1} \oplus \cdots \oplus f_{v_r}$ berechnet wird. Falls unter den Markierungen von $v_1, \ldots, v_r$ auch Variablen x_l mit $l > i$ vorkommen, können wir zu einem solchen Knoten v_j einen Dummy-Knoten v'_j, der mit x_i markiert ist und v_j als 0- und 1-Nachfolger hat, erzeugen. Dann können wir, wie zuvor beschrieben, den Knoten v für die Linearkombination erzeugen, wobei wir v'_j statt v_j berücksichtigen. Anschließend kann v'_j wieder gelöscht werden.

Die zweite Operation ist das *Entfernen von Linearkombinationen*, also die inverse Operation. Seien $v, v_1, \ldots, v_r$ Knoten, wobei v mit einer Variablen markiert ist, die nicht nach den Variablen von $v_1, \ldots, v_r$ in der Variablenordnung steht. Weiterhin gelte $f_v = f_{v_1} \oplus \cdots \oplus f_{v_r}$. Wir wollen nun v aus dem $\oplus$-OBDD entfernen. Dazu ersetzen wir jede Kante, die zu v führt, durch insgesamt r Kanten zu $v_1, \ldots, v_r$. Wenn dabei die Situation auftritt, dass es von einem Knoten w nun zwei Kanten zu v_i gibt, können wir natürlich beide Kanten entfernen. Wieder ändert sich die dargestellte Funktion nicht.

Als Beispiel für diese Operationen (siehe Abbildung 20) starten wir mit dem $\oplus$OBDD von oben. Wir fügen zunächst einen x_3-Knoten v_5 für die Funktion $f_{v_5} = f_{v_3} \oplus f_{v_4}$ ein und entfernen dann den Knoten v_3, der die Funktion $f_{v_3} = f_{v_4} \oplus f_{v_5}$ darstellt. Im zweiten Schritt fügen wir einen x_2-Knoten v_6 für $f_{v_6} = f_{v_2} \oplus f_{v_4}$ ein und entfernen anschließend den Knoten für $f_{v_2} = f_{v_6} \oplus f_{v_4}$. Wir sehen, dass wir nun das $\oplus$OBDD erhalten, das wir ebenfalls auf natürliche Weise aus dem von früher bekannten OBDD für $x_1x_2 \vee x_3$ erhalten würden.

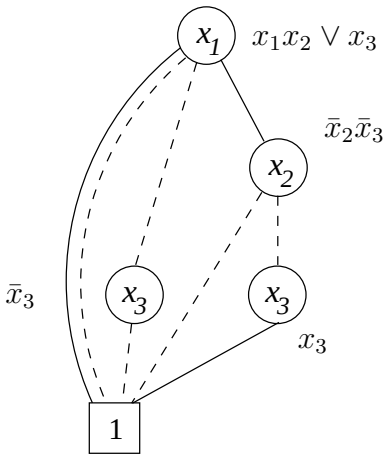
Wir geben nun einen Struktursatz für (knoten)minimale $\oplus$OBDDs an und beweisen ihn. Da minimale $\oplus$OBDDs nicht eindeutig sind, wird dieser Struktursatz „anders“ als die bisherigen Struktursätze aussehen, aber zugleich auch erklären, warum minimale $\oplus$OBDDs nicht eindeutig sind. Im Folgenden seien die Funktion f und die Variablenordnung $x_1, \ldots, x_n$ fest. Wir können jede Funktion $f : \{0,1\}^n \to \{0,1\}$ als Vektor aus dem Vektorraum $(\mathbb{Z}_2)^{2^n}$ auffassen, indem wir einfach die Wertetabelle als Vektor interpretieren. Sei $1 \leq k \leq n+1$.

Der Vektorraum V_k^f ist der Vektorraum, der von den Subfunktionen $f_{|x_1=c_1,\ldots,x_{i-1}=c_{i-1}}$ mit $k \leq i \leq n+1$ aufgespannt wird. Weiterhin gibt es für jedes $\oplus$OBDD G für f den Vektorraum V_k^G. Dieser Vektorraum wird von den Funktionen aufgespannt, die in G an der 1-Senke sowie den Knoten berechnet werden, die mit $x_k, \ldots, x_n$ markiert sind.

Zur Veranschaulichung betrachten wir als Beispiel wieder die Funktion $f(x_1, x_2, x_3) = x_1x_2 \vee x_3$. In der folgenden Tabelle sind Erzeugendensysteme für die einzelnen Vektorräume V_k^f angegeben.

$x_1x_2x_3$	V_1^f				V_2^f			V_3^f		V_4^f
000	0	0	0	1	0	0	1	0	1	1
001	1	1	1	1	1	1	1	1	1	1
010	0	1	0	1	1	0	1	0	1	1
011	1	1	1	1	1	1	1	1	1	1
100	0	0	0	1	0	0	1	0	1	1
101	1	1	1	1	1	1	1	1	1	1
110	1	1	0	1	1	0	1	0	1	1
111	1	1	1	1	1	1	1	1	1	1

Die zweite Tabelle enthält Erzeugendensysteme für die Vektorräume V_k^G des angegebenen $\oplus$OBDDs.



$x_1x_2x_3$	V_1^G					V_2^G				V_3^G			V_4^G
000	0	0	0	1	1	0	0	1	1	0	1	1	1
001	1	1	1	0	1	1	1	0	1	1	0	1	1
010	0	1	0	1	1	1	0	1	1	0	1	1	1
011	1	1	1	0	1	1	1	0	1	1	0	1	1
100	0	0	0	1	1	0	0	1	1	0	1	1	1
101	1	1	1	0	1	1	1	0	1	1	0	1	1
110	1	1	0	1	1	1	0	1	1	0	1	1	1
111	1	1	1	0	1	1	1	0	1	1	0	1	1

Man kann sich davon überzeugen, dass in diesem Beispiel die einzelnen Vektorräume V_i^G und V_i^f übereinstimmen, wobei allerdings bei V_i^G linear abhängige Funktionen an den Knoten berechnet werden. I.A. müssen V_i^G und V_i^f auch nicht übereinstimmen, es gilt nur eine Teilmengenbeziehung. Dies führt nun zu den folgenden Aussagen.

Lemma 11.1: Sei G ein $\oplus$OBDD für f. Dann gilt für alle $k \in \{1, \ldots, n+1\}$: $V_k^f \subseteq V_k^G$.

Satz 11.2: Ein $\oplus$OBDD für f und die Variablenordnung $x_1, \ldots, x_n$ mit minimaler Knotenzahl enthält genau $\dim(V_k^f) - \dim(V_{k+1}^f)$ Knoten, die mit x_k markiert sind. Insgesamt enthält es $\dim(V_f^1)$ Knoten (einschließlich der 1-Senke).

Die Knoten eines minimalen $\oplus$OBDDs entsprechen den Elementen einer Basis des Vektorraums V_f^1. Aus der linearen Algebra wissen wir, dass Basen immer dieselbe Anzahl von Elementen enthalten, aber i.A. nicht eindeutig bestimmt sind. Dasselbe gilt hier: Die minimale Anzahl von Knoten ist unabhängig von der Wahl der Basis gleich, bei verschiedenen Basen erhält man aber verschiedene $\oplus$OBDDs. Das abwechselnde Hinzufügen und Entfernen von Linearkombinationen im Beispiel oben können wir auch als Basiswechsel auffassen.

Beweis des Lemmas: Sei $g \in V_k^f$. Dann gibt es Subfunktionen $g_1, \ldots, g_l$ von f, sodass $g = g_1 \oplus \cdots \oplus g_l$ ist und die Subfunktionen g_i durch Konstantsetzen von $x_1, \ldots, x_j$, $j \geq k-1$, aus f entstehen. Wenn wir in G die partiellen Berechnungspfade für die Konstantsetzung für g_i verfolgen, erhalten wir Knoten $v_1, \ldots, v_r$, sodass $g_i = f_{v_1} \oplus \cdots \oplus f_{v_r} \in V_k^G$. Damit ist auch das Parity aller g_i, also g in V_k^G enthalten und die behauptete Inklusion folgt. □

Beweis des Satzes: Wir gehen ähnlich zum Beweis des Struktursatzes für OBDDs vor und konstruieren zunächst ein $\oplus$OBDD. Dann zeigen wir, dass es die behaupteten Eigenschaften hat und f berechnet.

Sei o.B.d.A. f ungleich der Nullfunktion (in diesem Fall hat das leere $\oplus$OBDD die geforderten Eigenschaften). Wir konstruieren das $\oplus$OBDD induktiv bottom-up. Zunächst erzeugen wir eine 1-Senke. Der zugehörige Basisvektor ist der Vektor, der nur aus Einsen besteht und der den Vektorraum V_{n+1}^f aufspannt. Seien nun die mit $x_{i+1}, \ldots, x_n$ markierten Knoten, sowie die 1-Senke, mit den zugehörigen Basisvektoren erzeugt. Wir benutzen nun den Basisergänzungssatz, um diese Basis von V_{i+1}^f zu einer Basis von $V_i^f \supseteq V_{i+1}^f$ zu ergänzen (falls es möglich ist, den Vektor zu f zu wählen, tun wir dies auf jeden Fall, da an der Quelle f und nicht irgendeine Linearkombination dargestellt werden soll). Für jeden gewählten Basisvektor erzeugen wir einen x_i-Knoten.

Wir konstruieren nun die Kanten des $\oplus$OBDDs. Sei v ein x_i-Knoten, an dem die Funktion $g \in V_i^f - V_{i+1}^f$ berechnet werden soll. Sei $g = f_1 \oplus \cdots \oplus f_l$, wobei $f_1, \ldots, f_l$ Subfunktionen von f sind, die durch Konstantsetzen von $x_1, \ldots, x_{r-1}$ mit $r \geq i$ entstehen (g ist nicht notwendigerweise eine solche Subfunktion). Dann ist $g_{|x_i=c} = f_{1|x_i=c} \oplus \cdots \oplus f_{l|x_i=c}$. Nach Induktionsvoraussetzung sind alle Funktionen $f_{1|x_i=c}, \ldots, f_{l|x_i=c}$ und damit auch $g_{|x_i=c}$ in V_{i+1}^f enthalten. D.h., für $g_{|x_i=c}$ gibt es gewählte Basisvektoren, deren Parity gleich $g_{|x_i=c}$ ist. Die c-Kanten von v zeigen dann auf die Knoten, die diesen Basisvektoren entsprechen.

Offensichtlich werden bei Anwendung des Basisergänzungssatzes $\dim(V_i^f) - \dim(V_{i+1}^f)$ mit x_i markierte Knoten erzeugt. Insgesamt werden also

$$1 + (\dim(V_n) - \dim(V_{n+1}^f)) + \cdots + (\dim(V_1^f) - \dim(V_2^f)) = \dim(V_1^f)$$

Knoten erzeugt. Es bleibt zu zeigen, dass das konstruierte $\oplus$OBDD die Funktion f darstellt. Dies geht wie üblich dadurch, dass wir mit Induktion über eine umgekehrte topologische Ordnung der Knoten zeigen, dass die an dem jeweiligen Knoten berechnete Funktion durch den zugehörigen Basisvektor dargestellt wird. Für die Senke ist das offensichtlich und für die x_i-Knoten wurden die ausgehenden c-Kanten genau so gewählt, dass das Parity der an den c-Nachfolgern berechneten Funktionen gleich der Subfunktion ist, die durch Ersetzen von x_i durch c entsteht. □

Wir kommen nun zum Algorithmus zur Minimierung der Knotenzahl bei vorgegebener Variablenordnung, o.B.d.A. $x_1, \ldots, x_n$. Dieser Algorithmus wird manchmal auch als Reduktionsalgorithmus bezeichnet. Eigentlich ist klar, was zu tun ist: Wir müssen lineare Abhängigkeiten im $\oplus$OBDD beseitigen, sodass die an den Knoten berechneten Funktionen den Vektorraum V_1^f aufspannen. Allerdings sind Tricks nötig, um dies in polynomieller Zeit auszuführen, da die Vektoren, die die Subfunktionen darstellen, exponentielle Länge in der Zahl der Variablen haben und daher nicht explizit gespeichert werden können.

Der Algorithmus besteht aus zwei Phasen, einer ersten, wo bottom-up gearbeitet wird und einer zweiten, wo top-down gearbeitet wird. Wir gehen immer davon aus, dass es keine von der Quelle aus nicht erreichbaren Knoten gibt.

Bottom-up-Phase: In dieser Phase wird sichergestellt, dass die an den Knoten berechneten Funktionen linear unabhängig sind.

Offensichtlich bildet der Vektor, der nur aus Einsen besteht und mit der 1-Senke assoziiert ist, eine linear unabhängige Menge. Im Folgenden gehen wir davon aus, dass die Menge der Vektoren, die zu den mit x_{i+1}-, ..., x_n-Knoten sowie zur 1-Senke gehören, linear unabhängig ist. Wir wollen nun erreichen, dass auch die Menge der Vektoren, die zu den x_i-, ..., x_n-Knoten sowie zur 1-Senke gehören, linear unabhängig ist. Dann können wir dieses Verfahren n-mal iterieren.

Seien $v_1, \ldots, v_r$ die Knoten, die mit $x_{i+1}, \ldots, x_n$ markiert sind, sowie die 1-Senke. Wir arbeiten nun in einem $2r$-dimensionalen Vektorraum. Dem Knoten v_j ordnen wir den Vektor zu, der aus zwei Kopien des j-ten Einheitsvektor der Dimension r besteht. Sei nun v ein x_i-Knoten. Auch zu v erzeugen wir einen $2r$-dimensionalen Vektor. Die ersten r Koordinaten des Vektors erhalten wir als das Parity der ersten Hälfte der Vektoren zu den 0-Nachfolgern von v, die zweite Hälfte der Koordinaten als das Parity der zweiten Hälfte der Vektoren zu den 1-Nachfolgern von v. Mit Hilfe dieser $2r$-dimensionalen Vektoren und Gauss-Elimination können wir nun die linearen Abhängigkeiten bestimmen. Wir haben bereits oben beschrieben, wie wir die linearen Abhängigkeiten beseitigen können. Auf diese Weise erreichen wir, dass auch die Subfunktionen, die an den mit $x_i, \ldots, x_n$ markierten Knoten sowie der 1-Senke berechnet werden, linear unabhängig sind.

Abschließend wollen wir dafür sorgen, dass die Funktion f an der Quelle dargestellt wird und nicht als Linearkombination von mehreren an Knoten berechneten Funktionen. Wenn es Knoten $v_1, \ldots, v_r$ gibt, sodass $f = f_{v_1} \oplus \cdots \oplus f_{v_r}$ gilt, suchen wir unter den Markierungen von $v_1, \ldots, v_r$ einen Knoten mit der Variable mit dem kleinsten Index x_j. Sei dies o.B.d.A. v_1. Dann wird mit der Operation Hinzufügen von Linearkombinationen ein x_j-Knoten für f erzeugt, und anschließend v_1 entfernt. Der erzeugte Knoten ist dann die Quelle.

Top-down-Phase: Wir benutzen wieder G, um das bis jetzt berechnete $\oplus$OBDD zu bezeichnen. Bis jetzt ist es allerdings noch möglich, dass V_k^G ein größerer Vektorraum als V_k^f ist. Daher gibt es die zweite Phase.

Wir wollen induktiv erreichen, dass an den mit $x_1, \ldots, x_j$ markierten Knoten nur linear unabhängige Funktionen aus V_1^f dargestellt werden. Sei o.B.d.A. die Quelle mit x_1 markiert. Dann gilt diese Forderung für $j = 1$. Sei nun j beliebig, sodass an den mit $x_1, \ldots, x_j$ markierten Knoten nur Funktionen aus V_1^f dargestellt werden. Für jeden Knoten v, der mit x_i,

$i \leq j$, markiert ist und der einen c-Nachfolger hat, der mit x_k, $k \geq j+1$, markiert ist, erzeugen wir mit der Operation Hinzufügen von Linearkombinationen einen x_{j+1}-Knoten für $f_{v|x_i=c}$ ($c \in \{0,1\}$) und lassen die c-Kante von v auf diesen neuen Knoten zeigen. Die „alten" x_{j+1}-Knoten sind nun nicht mehr erreichbar und können gelöscht werden. Die an den neuen x_{j+1}-Knoten berechneten Funktion sind wie gefordert in V_1^f enthalten, da sie Subfunktionen von Funktionen aus V_1^f sind (nämlich von den Funktionen, die an mit $x_1, \ldots, x_j$ markierten Knoten berechnet werden). Falls es unter den neu erzeugten x_{j+1}-Knoten noch lineare Abhängigkeiten gibt, können diese wie in der ersten Phase entfernt werden.

Damit haben wir erreicht, dass an den Knoten des $\oplus$OBDDs nur Funktionen aus V_1^f berechnet werden, die außerdem linear unabhängig sind. Also gibt es $\dim(V_1^f)$ Knoten, d.h., das $\oplus$OBDD ist knotenminimal für die vorgegebene Variablenordnung. Die Rechenzeit wird durch die $2n$ Gausseliminationen dominiert, die sich auf einfache Weise in Rechenzeit $O(|G|^3)$ realisieren lassen, wobei $|G|$ die Knotenanzahl bezeichnet. (Mit in der Praxis ineffizienten Algorithmen kann man auch einen Exponenten von ungefähr 2,38 erreichen.) Also beträgt die Rechenzeit $O(n|G|^3)$. Es ist naheliegend und man kann auch mit einer Reduktion zeigen, dass die Bestimmung von linearen Abhängigkeiten für die Knotenminimierung von $\oplus$OBDDs notwendig ist, sodass man keine essentiell besseren Rechenzeiten erwarten kann.

Ein Beispiel für die Minimierung eines $\oplus$OBDDs ist in Abb. 21 auf Seite 77 gezeigt, wobei die Schritte, in denen sich das $\oplus$OBDD nicht verändert, weggelassen sind. Zum Nachvollziehen des Beispiels ist es hilfreich, die fehlenden Schritte zu ergänzen und die Vektoren anzugeben, die die an den einzelnen Knoten dargestellten Funktionen repräsentieren. Das gegebene $\oplus$OBDD ist oben links gezeigt. In der ersten Phase wird nur einer der x_4-Knoten eliminiert, das Resultat befindet sich rechts daneben. In diesem $\oplus$OBDD gibt es aber noch zwei x_3-Knoten. In der zweiten Phase wird ein neuer x_2-Knoten eingefügt, sodass der vorhandene x_2-Knoten unerreichbar wird, also entfernt werden kann (3. Abbildung). Dadurch wird auch einer der x_3-Knoten unerreichbar und kann entfernt werden, sodass das in der letzten Abbildung gezeigte Resultat entsteht.

Wir kommen jetzt zu den übrigen Operationen für boolesche Funktionen. Die Synthese mit der Operation $\oplus$ ist besonders einfach: Seien o.B.d.A. die Quellen der gegebenen $\oplus$OBDDs mit derselben Variablen markiert. Wir verschmelzen die Quellen der gegebenen $\oplus$OBDDs und die Senken der gegebenen $\oplus$OBDDs. Dann ist offensichtlich, dass es im neuen $\oplus$OBDD eine ungerade Anzahl von Berechnungspfaden gibt, wenn es in genau einem der beiden gegebenen $\oplus$OBDDs eine ungerade Anzahl von Berechnungspfaden gibt.

Die Synthese mit dem Operator $\wedge$ geht wieder mit einer Produktgraphkonstruktion. Seien u und v x_i-Knoten aus den beiden gegebenen $\oplus$OBDDs. Seien $u_{0,1}, \ldots, u_{0,r(0)}$ die 0-Nachfolger von u und seien die 1-Nachfolger von u und die Nachfolger von v analog bezeichnet. Wir nutzen nun aus, dass

$$\begin{aligned} f_u \wedge f_v &= (\bar{x}_i(f_{u_{0,1}} \oplus \cdots \oplus f_{u_{0,r(0)}}) \oplus x_i(f_{u_{1,1}} \oplus \cdots \oplus f_{u_{1,r(1)}})) \\ &\quad \wedge (\bar{x}_i(f_{v_{0,1}} \oplus \cdots \oplus f_{v_{0,s(0)}}) \oplus x_i(f_{v_{1,1}} \oplus \cdots \oplus f_{v_{1,s(1)}})) \\ &= \bar{x}_i \bigoplus_{1 \leq i \leq r(0), 1 \leq j \leq s(0)} (f_{u_{0,i}} \wedge f_{v_{0,j}}) \oplus x_i \bigoplus_{1 \leq i \leq r(1), 1 \leq j \leq s(1)} (f_{u_{1,i}} \wedge f_{v_{1,j}}). \end{aligned}$$

D.h., in der Produktgraphkonstruktion sind die c-Nachfolger von (u, v) alle Paare von c-Nachfolgern von u und von c-Nachfolgern von v. Ansonsten lässt sich die Produktgraphkonstruktion leicht übertragen und die Größe der Ausgabe und die Rechenzeit ist durch $O(|G_1||G_2|)$ beschränkt, wobei $|G_1|$ und $|G_2|$ die Knotenzahlen der gegebenen $\oplus$OBDDs bezeichnen.

Auf diese Weise können wir auch die $\wedge$-Synthese von $\oplus$OBDDs realisieren, die wir durch Umformung aus OFDDs G_1 und G_2 erhalten haben. Das Resultat ist ein $\oplus$OBDD der Größe $O(|G_1||G_2|)$, also tritt hier keine exponentielle Vergrößerung ein. Wenn wir allerdings versuchen würden, aus dem Resultat wieder ein OFDD zu berechnen, könnte hierbei die exponentielle Vergrößerung eintreten.

Wir beachten, dass in der Rechnung ausgenutzt wird, dass $\oplus$ und $\wedge$ distributiv sind. Also können wir diesen Ansatz nicht direkt auf $\vee$ übertragen. Da aber die Negation einfach zu realisieren ist ($\oplus$-Synthese mit der 1-Funktion) können wir mit Hilfe der de-Morgan-Regeln die $\vee$-Synthese realisieren: $f \vee g = ((f \oplus 1) \wedge (g \oplus 1)) \oplus 1$. Wir erhalten dieselben Schranken für Rechenzeit und Speicherplatz wie bei der $\wedge$-Synthese.

Die übrigen Operationen erfordern nun keine wesentlichen neuen Ideen mehr. Der Test auf Erfüllbarkeit ist der Test, ob nach der Knotenminimierung etwas anderes als das leere $\oplus$OBDD übrig bleibt. Erfüllbarkeit-Anzahl ist NP-schwer, da diese Operation schon für den Spezialfall der OFDDs NP-schwer ist. Ersetzen durch Konstanten ist wie bei OBDDs durch Umsetzen der Zeiger auf x_i-Knoten auf alle c-Nachfolger möglich. Schließlich kann der Äquivalenztest durch eine Parity-Synthese und einen anschließenden Test auf Nichterfüllbarkeit realisiert werden.

Gegen den Einsatz von OBDD-Varianten mit einer variablen Zahl ausgehenden Kanten aus jedem Knoten spricht, dass es schwierig ist, die Darstellung effizient zu implementieren. Bei den $\oplus$OBDDs kommt noch die relativ ineffiziente Minimierung hinzu. Auf der anderen Seite zeigen uns die $\oplus$OBDDs, worin die Einschränkung der OFDDs besteht, und liefern auch Ideen für Algorithmen auf OFDDs. So ist überhaupt nicht klar, wie der Test, ob für zwei als OFDDs dargestellte Funktionen f und g gilt, dass $f \leq g$ ist, effizient realisiert werden kann. Mit Hilfe von $\oplus$OBDDs geht dies ganz einfach: Da $f \leq g \Leftrightarrow f \wedge \bar{g} = 0$ ist, genügt es, die OFDDs in $\oplus$OBDDs umzuformen, diese mit Synthese zu verknüpfen und auf Nichterfüllbarkeit zu testen.

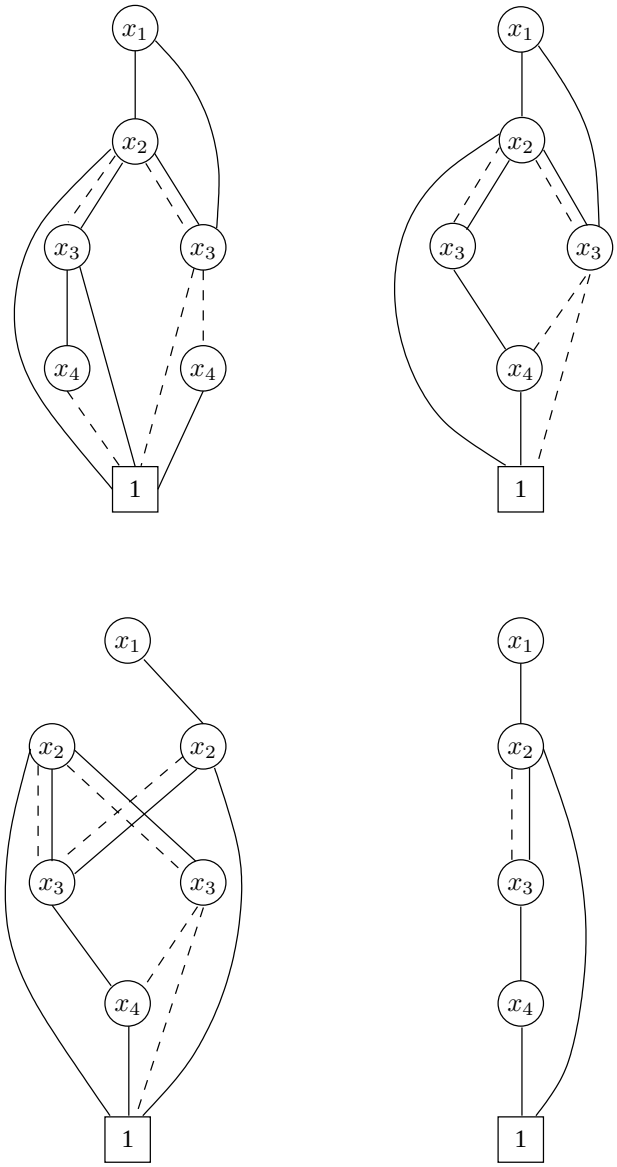


Abbildung 21: Ein Beispiel für den Algorithmus zur Minimierung der Knotenanzahl

12 Überblick über nichtdeterministische und randomisierte BDDs

Wir wollen die Idee der $\oplus$OBDDs, mehrere Berechnungspfade für jede Eingabe zu ermöglichen, verallgemeinern und systematischer behandeln. Dazu überlegen wir zuerst, dass die Interpretation der $\oplus$OBDDs, nämlich eine 1 zu berechnen, wenn die Anzahl akzeptierender Berechnungspfade *ungerade* ist, recht willkürlich ist und eigentlich nur durch die Anwendbarkeit von Methoden aus der linearen Algebra motiviert ist. Im Folgenden zählen wir einige weitere Möglichkeiten auf, wie wir von einer Anzahl akzeptierender Berechnungspfade zu einem Funktionswert kommen können.

Um mit der Literatur konsistent zu bleiben und technische Schwierigkeiten zu umgehen, definieren wir eine neue BDD-Variante. Hierbei betrachten wir wieder gerichtete azyklische Graphen mit einer Quelle und 0- und 1-Senken. Es gibt jetzt zwei Typen von inneren Knoten: die Berechnungsknoten, die mit einer Variablen markiert sind und eine ausgehende 0- und eine ausgehende 1-Kante haben, und *unmarkierte Knoten* mit zwei ausgehenden Kanten. Somit gibt es auf natürliche Weise für jede Eingabe a eventuell mehrere Berechnungspfade, die mit a konsistent sind, da wir an den unmarkierten Knoten zwei Möglichkeiten haben, einen Berechnungspfad fortzusetzen. In diesem Abschnitt machen wir keine weiteren Einschränkungen an die betrachteten BDDs wie etwa die Variablenordnungsbedingung.

Für die Berechnung des Funktionswertes aus der Anzahl der akzeptierenden Berechnungspfade betrachten wir die folgenden Möglichkeiten:

1. ODER über die Ergebnisse aller Berechnungspfade: Das BDD berechnet eine 1, wenn es mindestens einen akzeptierenden Berechnungspfad gibt.
2. UND über die Ergebnisse aller Berechnungspfade: Das BDD berechnet eine 1, wenn alle Berechnungspfade akzeptierend sind.
3. Mehrheitsentscheidung: Das BDD akzeptiert, wenn die Anzahl akzeptierender Berechnungspfade größer als die Anzahl verwerfender Berechnungspfade ist.

Wir überlegen nun, dass der Akzeptanzmodus ODER über die Ergebnisse aller Berechnungspfade dem Konzept des Nichtdeterminismus aus GTI entspricht; diese Sichtweise haben wir bereits bei der Konstruktion von $\oplus$OBDDs für die Index-Funktion behandelt. Man kann sich einfach vorstellen, dass das BDD an den unmarkierten Knoten die Kante „rät“, an der die Berechnung fortgesetzt wird. Die Verifikation, ob richtig geraten wurde, besteht darin zu testen, ob der Berechnungspfad an der 1-Senke endet. Diese Analogie zwischen dem Akzeptanzmodus ODER und dem Nichtdeterminismus bei Turingmaschinen kann auch durch eine beweisbare Aussage ausgedrückt werden. Sei $\mathcal{NL}$ die Menge aller Funktionen, die von nichtuniformen nichtdeterministischen Turingmaschinen mit logarithmischer Platzbeschränkung berechnet werden können.

Satz 12.1: Die Menge aller Funktionen, die von BDDs mit unmarkierten Knoten, polynomieller Größe und dem Akzeptanzmodus ODER berechnet werden können, ist gleich $\mathcal{NL}$.

Gegenüber dem Beweis der analogen Aussage über deterministische Turingmaschinen und $\mathcal{L}$ sind zum Beweis dieses Satzes keine neuen Ideen nötig, sodass wir hier darauf verzichten wollen. Aufgrund des Zusammenhangs mit nichtdeterministischen Turingmaschinen bezeichnen wir BDDs mit unmarkierten Knoten und dem Akzeptanzmodus ODER als nichtdeterministische BDDs. Die Einschränkung auf nichtdeterministische OBDDs geht dann auf die erwartete Weise.

Da nichtdeterministische Turingmaschinen in Bezug auf akzeptieren und verwerfen nicht symmetrisch sind, betrachtet man für die Komplemente von Sprachen, die von nichtdeterministischen Turingmaschinen akzeptiert werden können, ein spezielles Maschinenmodell, die so genannten co-nichtdeterministischen Turingmaschinen. Diese Turingmaschinen haben wie die gewöhnlichen nichtdeterministischen Turingmaschinen eine Übergangsrelation statt einer Übergangsfunktion, und sie akzeptieren, wenn die Berechnung für *alle* Berechnungspfade akzeptierend ist. Es ist wieder leicht zu zeigen, dass die Menge aller Funktionen, die von BDDs mit unmarkierten Knoten, polynomieller Größe und dem Akzeptanzmodus UND berechnet werden können, gleich der Menge der Sprachen ist, die von nichtuniformen co-nichtdeterministischen Turingmaschinen auf logarithmischem Platz berechnet werden können. Die zugehörige Komplexitätsklasse heißt co-$\mathcal{NL}$, das BDD-Modell co-nichtdeterministische BDDs.

Wir merken an, dass der Satz von Immerman und Szelepcsényi (siehe Vorlesung Komplexitätstheorie) auch für nichtuniforme platzbeschränkte Turingmaschinen gilt und damit aussagt, dass $\mathcal{NL} =$ co-$\mathcal{NL}$ ist.

Wenn wir schließlich die Akzeptanz von BDDs von einer Mehrheitsentscheidung abhängig machen, können wir auch an den unmarkierten Knoten auswürfeln, über welche Kante die Berechnung fortgesetzt wird. Ein solches BDD berechnet genau dann eine 1, wenn die Akzeptanzwahrscheinlichkeit mindestens $1/2$ beträgt. Der Zusammenhang zu probabilistischen Turingmaschinen ist über diese Interpretation naheliegend. Diese Variante von BDDs wird randomisierte BDDs genannt. Analog zu den probabilistischen Turingmaschinen kann man auch andere Akzeptanzmodi definieren, etwa einseitigen Fehler (d.h., falls $f(x) = 1$, muss mit einer Wahrscheinlichkeit von mindestens $1/2$ akzeptiert werden, falls $f(x) = 0$, muss mit Sicherheit (mit Wahrscheinlichkeit 1) verworfen werden).

Eine Variante von nichtdeterministischen OBDDs, die auch in der Praxis als Datenstruktur für boolesche Funktionen eingesetzt wird, sind die sogenannten Partitioned BDDs (PBDDs). Die Eigenschaften dieser Variante werden wir daher untersuchen. Dagegen ist bekannt, dass für randomisierte OBDDs der Erfüllbarkeitstest und der Äquivalenztest NP-schwer sind, sodass sich diese Variante wohl nicht als Datenstruktur eignet. Allerdings können mit Hilfe der von den OBDDs bekannten Algorithmen auch Erkenntnisse über randomisierte OBDDs gewonnen werden, und es ist nicht schwer zu zeigen, dass randomisierte OBDDs polynomieller Größe mehr Funktionen darstellen können als deterministische OBDDs. Schließlich wollen wir in Abschnitt 18 Methoden für den Beweis unterer Schranken für nichtdeterministische OBDDs kennenlernen.

13 Partitioned BDDs

Uneingeschränkte nichtdeterministische OBDDs sind als Datenstruktur für boolesche Funktionen weniger geeignet, da der Äquivalenztest NP-schwer ist. Dies ist einfach einzusehen. Das Problem SAT besteht darin, für eine Konjunktion von Klauseln zu testen, ob die dargestellte Funktion ungleich der Nullfunktion ist. Dies ist offensichtlich äquivalent zu dem Test, ob das Komplement der gegebenen Konjunktion von Klauseln ungleich der Einsfunktion ist. Die Komplementbildung bei einer Konjunktion von Klauseln ergibt mit den de-Morgan-Regeln eine Disjunktion von Monomen, die sich in linearer Größe mit nichtdeterministischen OBDDs darstellen lässt. Weiterhin hat die Einsfunktion nichtdeterministische OBDDs konstanter Größe. Ein polynomieller Äquivalenztest für nichtdeterministische OBDDs würde also P $=$ NP implizieren.

Bei Partitioned BDDs (PBDDs) wird daher der Nichtdeterminismus eingeschränkt, indem nur *am Anfang einer Auswertung* nichtdeterministisch entschieden werden darf, in welchem von insgesamt k gegebenen (deterministischen) OBDDs $G_1, \ldots, G_k$ die Auswertung ausgeführt wird. Seien $f_1, \ldots, f_k$ die Funktionen, die von $G_1, \ldots, G_k$ dargestellt werden. Dann stellt das PBDD $(G_1, \ldots, G_k)$ die Funktion $f_1 \vee \cdots \vee f_k$ dar. Weiterhin gibt es noch eine Erweiterung und eine Einschränkung. Die Erweiterung besteht darin, dass die OBDDs $G_1, \ldots, G_k$ *verschiedene* Variablenordnungen haben dürfen. Wenn wir nun zwei PBDDs $(G_1, \ldots, G_k)$ und $(H_1, \ldots, H_k)$ z.B. mit Synthese verknüpfen wollen, ist die Forderung naheliegend, dass G_i und H_i jeweils dieselbe Variablenordnung haben müssen (was auch erfordert, dass die Anzahl der Teile für beide PBDDs gleich ist). Wir benötigen aber noch mehr. Wenn z.B. für denselben Input x nur G_1 und H_2 eine 1 berechnen und G_1 und H_2 verschiedene Variablenordnungen haben, wird die Synthese schwierig zu realisieren, da ein simultaner Durchlauf durch die OBDDs nur bei gleicher Variablenordnung funktioniert. Diese Überlegungen führen zu der folgenden Definition von PBDDs.

Definition 13.1: Ein Vektor $w_1, \ldots, w_k$ von Funktionen aus B_n heißt Vektor von *Windowfunktionen über* $\{0,1\}^n$, wenn $w_1 \vee \cdots \vee w_k = 1$. Diese Windowfunktionen heißen *disjunkt*, wenn für alle i, j mit $i \neq j$ gilt, dass $w_i \wedge w_j = 0$.

Ein (k, w, π)-PBDD für einen Vektor $w = (w_1, \ldots, w_k)$ von Windowfunktionen und einen Vektor $\pi = (\pi_1, \ldots, \pi_k)$ von Variablenordnungen besteht aus k deterministischen OBDDs $G_1, \ldots, G_k$ mit den Variablenordnungen $\pi_1, \ldots, \pi_k$. Es stellt die Funktion $f \in B_n$ dar, wenn jedes G_i die Funktion $f_i = f \wedge w_i$ darstellt.

Da $G_1, \ldots, G_k$ verschiedene Variablenordnungen haben können, gehen wir immer davon aus, dass sie keine Knoten gemeinsam haben. Die Disjunktion der von den einzelnen OBDDs dargestellten Funktionen ist

$$f_1 \vee \cdots \vee f_k = (f \wedge w_1) \vee \cdots \vee (f \wedge w_k) = f \wedge (w_1 \vee \cdots \vee w_k) = f.$$

Wir können ein PBDD als nichtdeterministisches BDD auffassen, das mit einem Baum von $k - 1$ unmarkierten Knoten beginnt, dessen Blätter die Quellen von $G_1, \ldots, G_k$ sind. Da dieser Baum immer gleich aussieht und für die folgenden Betrachtungen nicht hilfreich ist,

haben wir ihn aus der Definition weggelassen und werden ihn auch im Folgenden nicht weiter beachten.

Um uns davon zu überzeugen, dass man auf diese Weise mehr Funktionen als mit gewöhnlichen deterministischen OBDDs darstellen kann, betrachten wir die Funktion ISA_n für $n = 2^l$. Hierbei sind wieder $y_0, \ldots, y_{l-1}$ Adressvariablen und die $x_0, \ldots, x_{n-1}$ zugleich Adress- und Datenvariablen. Die Funktion w_i, $1 \leq i \leq n$, nimmt den Wert 1 an, wenn die adressierte x-Variable die Variable x_{i-1} ist. Als gemeinsame Variablenordnung für alle OBDDs wählen wir $y_0, \ldots, y_{l-1}, x_0, \ldots, x_{n-1}$. Das OBDD G_i berechnet eine 1, wenn die adressierte x-Variable die Variable x_{i-1} ist und diese den Wert 1 annimmt. Die Arbeitsweise von G_i ist einfach einzusehen. Zunächst werden in einem vollständigen Entscheidungsbaum die y-Variablen und an den Blättern dieses Baumes, wiederum in vollständigen Entscheidungsbäumen die Variablen des adressierten x-Blocks sowie x_{i-1} getestet. Anschließend ist bekannt, ob x_{i-1} adressiert wird und diese Variable den Wert 1 hat. Die Größe von G_i kann leicht mit $O(n^2/\log n)$ abgeschätzt werden. In diesem Beispiel nutzen wir nicht aus, dass die einzelnen OBDDs verschiedene Variablenordnungen haben dürfen.

Wir wollen nun ein weiteres PBDD für ISA angeben, das sogar mit nur zwei Teilen auskommt. Sei x_r die erste Variable des x-Blocks, der durch die y-Variablen adressiert wird. Sei x_s die auszugebende x-Variable. Als Windowfunktion für den ersten Teil wählen wir die Funktion, die testet, ob $s \geq r$ ist. Als Windowfunktion für den zweiten Teil wählen wir das Komplement dieser Funktion. (Diese Funktion berechnet auch dann eine 1, wenn $|y| \geq \lfloor n/\log n \rfloor$, also wenn die y-Variablen keinen x-Block adressieren.) Die Disjunktion dieser Windowfunktionen ist offensichtlich 1. Als Variablenordnung für den ersten Teil wählen wir $y_0, \ldots, y_{\log n-1}, x_0, \ldots, x_{n-1}$, als Variablenordnung für den zweiten Teil $y_0, \ldots, y_{\log n-1}, x_{n-1}, \ldots, x_0$. Es ist nun einfach, die OBDDs für die einzelnen Teile zu beschreiben. Beide OBDDs beginnen aus einem vollständigen Baum der y-Variablen, um $|y|$ zu bestimmen. Falls $|y| \geq \lfloor n/\log n \rfloor$, wird eine 0 ausgegeben. Anderenfalls werden die x-Variablen des adressierten Blocks ebenfalls in einem vollständigen binären Baum getestet. An den Blättern dieses Baumes ist bekannt, welche Variable auszugeben ist. Das erste OBDD gibt den Wert dieser Variablen nur aus, wenn sie nicht vor dem adressierten Block liegt, anderenfalls gibt es den Wert 0 aus. Wenn die auszugebende Variable im adressierten Block liegt, wurde sie in dem Baum getestet und gespeichert, sodass sie nun ausgegeben werden kann. Wenn die auszugebende Variable nach dem adressierten Block liegt, darf sie noch getestet werden. Analog darf in dem zweiten OBDD die auszugebende Variable noch getestet werden, wenn sie vor dem adressierten Block liegt, da in diesem OBDD die x-Variablen in der umgekehrten Reihenfolge gelesen werden. Es ist nun einfach, die Größe der beiden OBDDs durch $O(n^2/\log n)$ abzuschätzen, da ihre Größe durch $\lfloor n/\log n \rfloor + 1$ vollständige Bäume der Größe n dominiert wird.

Mit PBDDs können wir weiterhin eine gegebene Disjunktion von Monomen in linearer Größe darstellen; allerdings besteht hier die Schwierigkeit darin, geeignete Window-Funktionen zu finden, deren Disjunktion 1 ist. Diese Window-Funktionen brauchen wir insbesondere für die Darstellung der 1-Funktion, die in dem Beweis, dass das Erfüllbarkeitsproblems NP-schwer ist, benötigt wird. Aus diesem Grund kann man diesen Beweis nicht einfach auf PBDDs übertragen. Wir beachten, dass die Darstellung der 1-Funktion aus OBDDs für die Funktionen $1 \wedge w_1, \ldots, 1 \wedge w_k$, also aus OBDDs für die Windowfunktionen bestehen.

Wir wollen nun einige Eigenschaften der PBDDs untersuchen.

Lemma 13.2: Minimale (k, w, π)-PBDDs sind eine bis auf Isomorphie eindeutige Darstellung von booleschen Funktionen. Die Reduktion ist in linearer Zeit möglich.

Beweis: Durch die Vorgabe von k und w sind die an den einzelnen OBDDs darzustellenden Funktionen $f \wedge w_i$ eindeutig festgelegt. Durch die Vorgabe von π sind auch die minimalen OBDDs für diese Funktionen bis auf Isomorphie eindeutig. Diese OBDDs können separat reduziert werden, wobei wir den Linearzeitalgorithmus für die Reduktion anwenden. □

Also sind auch der Äquivalenztest und der Erfüllbarkeitstest in linearer Zeit möglich. Wir diskutieren nun weitere Operationen.

Für die Auswertung genügt es, die einzelnen OBDDs auszuwerten und das ODER über die Resultate zu bilden. Die Rechenzeit beträgt dann $O(kn)$.

Für die Synthese der (k, w, π)-PBDDs $G = (G_1, \ldots, G_k)$ und $H = (H_1, \ldots, H_k)$ mit einem 0-erhaltenden Operator $\otimes$ genügt es, die Teile G_i und H_i einzeln zu verknüpfen. Dies sieht man leicht ein, da für 0-erhaltende Operatoren gilt, dass

$$(g \otimes h) \wedge w_i \quad = \quad (g \wedge w_i) \otimes (h \wedge w_i).$$

Dies lässt sich leicht durch Einsetzen von 0 und 1 für w_i verifizieren. Also ist die Synthese für 0-erhaltende Operatoren in Zeit $O(|G||H|)$ möglich.

Ein (k, w, π)-PBDD G für $\bar{f}$ kann aus einem (k, w, π)-PBDD für f mit der $\oplus$-Synthese mit der Einsfunktion berechnet werden. Wie oben bemerkt hat die Einsfunktion nicht unbedingt eine Darstellung konstanter Größe, da ja die Funktionen $w_1, \ldots, w_k$ darzustellen sind. Sei G_1 eine Darstellung der 1-Funktion. Dann ist die Negation in Rechenzeit $O(|G||G_1|)$ möglich.

Nun können wir (wie bei den ZBDDs) die Synthese für einen nicht 0-erhaltenden Operator $\otimes$ in die Synthese für die Negation von $\otimes$ und eine anschließende Negation des Resultats zerlegen. Die Rechenzeit und die Ausgabegröße sind dann durch $O(|G||H||G_1|)$ beschränkt.

Für disjunkte Windowfunktionen können die Operationen Erfüllbarkeit-Anzahl und Erfüllbarkeit-Alle realisiert werden, indem der entsprechende Algorithmus für OBDDs auf den einzelnen Teilen aufgerufen wird. Bei Erfüllbarkeit-Anzahl sind die Resultate zu addieren. Die Rechenzeit ist dann linear. Bei Erfüllbarkeit-Alle sind die ausgegebenen Listen von erfüllenden Belegungen aneinanderzuhängen. Die Rechenzeit ist wie bei OBDDs linear in der Ein- und Ausgabegröße. Bei der Operation Erfüllbarkeit-Anzahl ist allerdings nicht klar, wie sie für nicht disjunkte Windowfunktionen einfach realisiert werden kann, bei Erfüllbarkeit-Alle hat man bei nicht disjunkten Windowfunktionen das Problem, Duplikate aus den verschiedenen Listen von erfüllenden Belegungen zu entfernen.

Erstaunlicherweise bereiten die Operation Ersetzen durch Konstanten und die daraus abgeleiteten Operationen Probleme. Wenn wir z.B. x_1 durch 0 ersetzen wollen, können wir nicht einfach die 1-Kanten von den x_1-Knoten auf den 0-Nachfolger umsetzen, da i.A. $(f_{|x_1=0} \wedge w_i) \neq (f \wedge w_i)_{|x_1=0}$, z.B. für $w_1 = x_1, f = 1$. Dass dies zu Problemen führt, sehen wir an einem einfachen Beispiel. Seien $w_1 = x_1$ und $w_2 = \bar{x}_1$. Die Funktion $f = 1$

hat dann eine Darstellung, die aus zwei OBDDs für x_1 und $\bar{x}_1$ besteht. Wenn nun x_1 durch 0 ersetzt wird, sollte das Resultat weiterhin die Einsfunktion sein. Wenn wir den üblichen Algorithmus für OBDDs anwenden, erhalten wir aber OBDDs für die Funktionen 0 und 1. Dies ist keine legale Darstellung der Einsfunktion, da das zweite OBDD auch für Eingaben x eine 1 berechnet, für die die zugehörige Windowfunktion 0 ergibt.

Unangenehmerweise kann die korrekte Ausführung von Ersetzen durch Konstanten sogar zu einer exponentiellen Vergrößerung der Darstellung führen. Um dies zu zeigen, benutzen wir Funktionen, die auf den Variablen $x_{i,j}$, $1 \leq i, j \leq n$, sowie auf einer Variablen s definiert sind. Die x-Variablen sind also matrixförmig angeordnet. Sei f die Funktion, die testet, ob in jeder Zeile der Matrix aus den x-Variablen genau eine Eins vorkommt. Weiterhin sei g die Funktion, die testet, ob in jeder Spalte aus der Matrix aus den x-Variablen genau eine Eins vorkommt. Die Funktion f hat OBDDs linearer Größe für eine zeilenweise Variablenordnung, die Funktion g für eine spaltenweise Variablenordnung. Nun ist leicht einzusehen, dass die Funktion $h = \bar{s}f \vee sg$ PBDDs linearer Größe hat, die sogar nur aus zwei Teilen bestehen. Der erste Teil hat eine zeilenweise Variablenordnung der x-Variablen gefolgt von s, und die Windowfunktion ist $\bar{s}$; der zweite Teil hat eine spaltenweise Variablenordnung der x-Variablen gefolgt von s, und die Windowfunktion ist s. Wenn wir nun s durch 0 ersetzen, erhalten wir die Funktion f. Der zweite Teil des PBDDs für f muss die Funktion sf für eine spaltenweise Variablenordnung darstellen. Es ist eine einfache Übungsaufgabe zu zeigen, dass dies exponentielle Größe erfordert. Ebenso ist es leicht, Beispiele zu finden, für die auch Quantifizierung zu einer exponentiellen Vergrößerung führt.

Wir wollen aber anmerken, dass die Schwierigkeit mit Ersetzen durch Konstanten nur auftritt, wenn mindestens eine der Windowfunktionen von der ersetzten Variablen x_i essentiell abhängt. Wenn dies nicht der Fall ist, gilt für alle Windowfunktionen $(f_{|x_i=c} \wedge w_j) = (f \wedge w_j)_{|x_i=c}$, sodass wir die Variable x_i in allen OBDDs auf die gewohnte Weise konstantsetzen dürfen.

Es bleibt die Frage nach der Wahl der Windowfunktionen und der Variablenordnungen. Die Komplexität des Problems, diese Wahl so zu treffen, dass die Größe minimal wird, ist unbekannt. Eine naheliegende Heuristik besteht darin, eine Auswahl der Variablen, z.B., $\{x_{i(1)}, \ldots, x_{i(l)}\}$ zu treffen und als Windowfunktionen alle 2^l Monome über diesen Variablen zu wählen. Diese Windowfunktionen sind disjunkt und ihre Disjunktion ergibt, wie gefordert, die Einsfunktion. Auf die einzelnen Teile des PBDDs können dann bekannte Heuristiken wie der Sifting-Algorithmus angewendet werden.

14 Randomisierte OBDDs

Bei randomisierten OBDDs wählen wir für die Fortsetzung der Berechnung an unmarkierten Knoten eine der beiden ausgehenden Kanten mit einer Wahrscheinlichkeit von jeweils $1/2$ aus. Das Resultat einer Berechnung eines randomisierten OBDDs G auf einer Eingabe a ist somit eine Zufallsvariable $G(a)$. Man überlegt sich leicht, dass man randomisierten OBDDs erlauben muss, nicht das richtige Ergebnis zu berechnen: Wenn ein randomisiertes OBDD für alle zufälligen Wahlen immer das richtige Ergebnis liefern würde, könnte man die zufälligen Wahlen weglassen und an diesen Stellen irgendeine (feste) Wahl treffen. Das Ergebnis wäre dann immer noch für alle Eingaben richtig und das OBDD deterministisch.

In der Komplexitätstheorie werden verschiedene Fehlerarten für randomisierte Algorithmen unterschieden. Dieselbe Unterscheidung benutzen wir auch für die randomisierten OBDDs.

Definition 14.1: Sei G ein randomisiertes OBDD auf n Variablen.

1. G stellt f mit *zweiseitigem, unbeschränktem Fehler* dar, wenn für alle Eingaben $a \in \{0,1\}^n$ gilt, dass Prob$(G(a) = f(a)) > 1/2$.

2. G stellt f mit *zweiseitigem, beschränktem Fehler* dar, wenn es ein $\varepsilon > 0$ gibt, sodass für alle Eingaben $a \in \{0,1\}^n$ gilt, dass Prob$(G(a) = f(a)) > 1/2 + \varepsilon$.

3. G stellt f mit *einseitigem, beschränktem Fehler* dar, wenn für alle $a \in f^{-1}(1)$ gilt, dass Prob$(G(a) = 1) \geq 1/2$, und für alle $a \in f^{-1}(0)$ gilt, dass Prob$(G(a) = 0) = 1$.

4. G stellt f *fehlerfrei mit Versagenswahrscheinlichkeit* ε dar, wenn für alle $a \in \{0,1\}^n$ gilt, dass Prob$(G(a) = \neg f(a)) = 0$ und Prob$(G(a) =?) \leq \varepsilon$ ist. (In diesem Fall hat G auch mit ? markierte Senken.)

Wir beachten, dass sich die Wahrscheinlichkeiten nur auf die zufällige Wahl der aus unmarkierten Knoten ausgehenden Kanten bezieht und dass die Forderungen *für alle* Eingaben gelten müssen. Die Begriffe beschränkter und unbeschränkter Fehler werden später noch genauer diskutiert. Ohne Beweis wollen wir nur erwähnen, dass es zu fehlerfreien randomisierten OBDDs äquivalente deterministische OBDDs, die höchstens polynomiell größer sind, gibt. Allerdings ist dies nur eine Existenzaussage; es nicht bekannt, ob die Umformung eines gegebenen fehlerfreien randomisierten OBDDs in ein deterministisches OBDD effizient möglich ist. Wir werden sehen, dass das Erfüllbarkeitsproblem für randomisierte OBDDs mit beschränktem Fehler NP-schwer ist und damit ebenfalls der Äquivalenztest. Somit sind randomisierte OBDDs mit Fehler als Datenstruktur wohl nicht zu verwenden. Allerdings können wir an randomisierten BDDs einige Techniken für randomisierte Rechnungen studieren.

Wir zeigen zunächst, dass die Auswertung von randomisierten OBDDs in polynomieller Zeit möglich ist. Dazu berechnen wir für jeden Knoten die Wahrscheinlichkeit, ihn bei der Berechnung für die gegebene Eingabe a zu erreichen. Der Algorithmus dafür ist ähnlich zum Algorithmus für Erfüllbarkeit-Anzahl bei OBDDs. Wir betrachten die Knoten nacheinander

in einer topologischen Ordnung. Die Wahrscheinlichkeit für die Quelle ist 1. Wenn die Wahrscheinlichkeit an einem x_i-Knoten p ist, beträgt die Wahrscheinlichkeit, dass die Berechnung über die a_i-Kante verläuft, ebenfalls p, sodass wir p zu der Wahrscheinlichkeit für den a_i-Nachfolger addieren. Wenn die Wahrscheinlichkeit an einem unmarkierten Knoten p ist, beträgt für jede der ausgehenden Kanten die Wahrscheinlichkeit, dass die Berechnung über sie verläuft, $p/2$. Also addieren wir zu den Wahrscheinlichkeiten der beiden Nachfolger jeweils $p/2$. Wenn das randomisierte OBDD k unmarkierte Knoten hat, sind alle Wahrscheinlichkeiten Vielfache von 2^{-k}, so dass alle betrachteten Zahlen lineare Länge in der OBDD-Größe haben. Also ist die Rechenzeit quadratisch (bei logarithmischem Kostenmaß). Die für die 0- bzw. 1-Senke berechnete Wahrscheinlichkeit ist dann die Wahrscheinlichkeit, dass das randomisierte OBDD eine 0 bzw. 1 berechnet. Aus dem Wert dieser Wahrscheinlichkeiten für die Eingabe a und der Fehlerart des betrachteten randomisierten OBDDs können wir den Funktionswert bestimmen.

Unter Probability Amplification verstehen wir die Verringerung der Fehlerwahrscheinlichkeit eines randomisierten Algorithmus. Dazu lässt man den Algorithmus mehrfach (mit unabhängigen Zufallsbits) laufen und trifft bei zweiseitigem Fehler eine Mehrheitsentscheidung bzw. berechnet bei einseitigem Fehler das ODER der Ergebnisse. Bei Letzterem nutzt man aus, dass bei einseitigem Fehler die Ausgabe 1 immer korrekt ist, während die Ausgabe 0 falsch sein kann. In diesem Fall multiplizieren sich die Fehlerwahrscheinlichkeiten. (Bei zweiseitigem Fehler erhält man mit Hilfe von Chernoff-Schranken ähnliche Ergebnisse, die wir hier nicht weiter ausführen.)

Bei randomisierten OBDDs ist nun das Problem, dass wir wegen der Variablenordnungsbedingung nicht einfach zwei Berechnungen nacheinander ausführen können. Aber wir können mit Hilfe von Synthese mehrere Rechnungen simultan ausführen.

Dazu müssen wir die Synthese auf randomisierte OBDDs erweitern. Seien $G = (V_G, E_G)$ und $H = (V_H, E_H)$ die gegebenen randomisierten OBDDs mit derselben Variablenordnung. Bei der Produktgraphkonstruktion entstehen Paare (u, v), $u \in V_G$, $v \in V_H$, wobei u und v auch unmarkierte Knoten sein können. Wenn u und v markierte Knoten sind, gehen wir wie bei deterministischen OBDDs vor. Sei nun u ein x_i-Knoten und v ein unmarkierter Knoten mit den Nachfolgern v_0 und v_1. Im Produktgraphen muss dann die zufällige Wahl in H simuliert werden, während in G „gewartet“ wird. Also ist (u, v) ein unmarkierter Knoten mit den Nachfolgern (u, v_0) und (u, v_1). Es bleibt noch der Fall, dass die beiden Knoten u und v (mit den Nachfolgern u_0 und u_1 bzw. v_0 und v_1) unmarkiert sind. Da die zufälligen Wahlen in G und H unabhängig sein sollen, brauchen wir drei unmarkierte Knoten, wobei einer die zufällige Wahl in G und die beiden Nachfolger dieses Knotens die zufällige Wahl in H simulieren. Also ist (u, v) ein unmarkierter Knoten, der zwei weitere unmarkierte Knoten (u_0, v) und (u_1, v) als Nachfolger hat. Die vier Nachfolger dieser beiden Knoten sind dann (u_0, v_0), (u_0, v_1), (u_1, v_0) und (u_1, v_1). Dann simuliert (u, v) die zufällige Wahl in G, und (u_0, v) und (u_1, v) simulieren die zufällige Wahl in H. Die Rechenzeit für die Synthese ist weiterhin $O(|G||H|)$.

Für die Probability Amplification berechnen wir den Produktgraphen von G mit einer Kopie von G. Wie oben diskutieren wir nur den Fall des einseitigen Fehlers. Wenn eine Senke $(0, 0)$ erreicht wird, bedeutet dies, dass beide Kopien von G eine 0 berechnet haben, also geben wir eine 0 aus. Bei allen anderen Senken wissen wir, dass Ergebnis 1 richtig ist, daher

sind dies 1-Senken. Die Fehlerwahrscheinlichkeit ist die Wahrscheinlichkeit, dass bei einem Funktionswert 1 das Resultat 0 ausgegeben wird. Wenn diese bei G durch δ beschränkt ist, ist sie bei diesem Produktgraphen durch δ^2 beschränkt.

Analog können wir die Synthese von konstant vielen Kopien von G durchführen und damit die Fehlerwahrscheinlichkeit auf $\delta^{O(1)}$, also eine beliebig kleine Konstante, verringern. Somit war die Fehlerschranke $1/2$ in der obigen Definition willkürlich gewählt, stattdessen können wir jede andere Konstante einsetzen und erhalten ein gleich mächtiges Modell. Wir beachten aber den Unterschied zu polynomiell zeitbeschränkten randomisierten Algorithmen. Diese dürfen wir sogar polynomiell oft iterieren, sodass man die Fehlerwahrscheinlichkeit sogar auf $1/2^n$ reduzieren kann.

Wir können nun auch die Unterscheidung in Modelle mit beschränktem und unbeschränktem Fehler besser verstehen. Unter unbeschränktem Fehler wird die Situation verstanden, dass man sich einer Fehlerschranke, wo die Berechnung trivial wird, beliebig nähern darf, diese aber nicht erreichen darf. Bei zweiseitigem Fehler ist diese Fehlerschranke $1/2$, da dies die Fehlerschranke ist, die man ohne Kenntnis der zu berechnenden Funktion mit einem Münzwurf erreichen kann. Bei einseitigem Fehler ist dies für den Fall $a \in f^{-1}(1)$ die Forderung Prob$(G(a) = 1) > 0$. Dies entspricht genau dem Nichtdeterminismus. Bei beschränktem Fehler wird dagegen verlangt, dass man von dieser Fehlerschranke um einen konstanten Betrag entfernt bleibt, was wegen der Möglichkeit der Probability Amplification äquivalent zu der angegebenen Definition von randomisierten OBDDs ist.

Wir kommen nun zur wesentlichen Methode für den Entwurf von randomisierten OBDDs, zu der sogenannten Fingerprinting-Methode. Als erstes Beispiel betrachten wir die Ungleichheitsfunktion NEQ$_n(x_1, \ldots, x_n, y_1, \ldots, y_n)$, die den Wert 1 genau dann annimmt, wenn $x_1 \neq y_1 \vee \cdots \vee x_n \neq y_n$. Man überlegt sich leicht, dass OBDDs für NEQ exponentielle Größe haben, wenn zuerst die x- und dann die y-Variablen getestet werden. Das Problem für die OBDDs besteht darin, dass der Wert von allen x-Variablen bekannt sein muss, wenn die y-Variablen getestet werden. Die neue Idee besteht nun darin, anstelle von x nur einen „Fingerabdruck“ von x im OBDD zu speichern und diesen mit einem Fingerabdruck von y zu vergleichen. Um an diesen Fingerabdruck zu kommen, interpretieren wir x als n-stellige Zahl $|x|$ aus $\{0, \ldots, 2^{n-1}\}$. Sei p eine Primzahl. Als Fingerabdruck benutzen wir $|x| \bmod p$. Das OBDD akzeptiert, wenn $|x| \not\equiv |y| \bmod p$ ist.

Wenn x und y gleich sind, ist auch $|x| \equiv |y| \bmod p$, sodass das OBDD auf jeden Fall eine 0 berechnet. Allerdings können wir nicht verhindern, dass z.B. $|x| = |y| + p$ ist, sodass das OBDD verwirft, obwohl x und y verschieden sind. Die entscheidende Idee besteht nun darin, p zufällig zu wählen, z.B. aus $p_1, \ldots, p_{2m}$, wobei dies die $2m$ kleinsten Primzahlen sind und m die kleinste Zweierpotenz ist, die mindestens so groß wie n ist. Wenn x und y gleich ist, wird weiterhin mit Sicherheit eine 0 berechnet. Wenn x und y verschieden sind, ist $| \, |x| - |y| \, | \leq 2^n$, sodass höchstens n der $2m$ verwendeten Primzahlen Teiler von $| \, |x| - |y| \, |$ sein können. Wenn wir eine Primzahl zufällig gemäß Gleichverteilung wählen, ist die Wahrscheinlichkeit, einen Teiler von $| \, |x| - |y| \, |$ zu wählen, durch $n/(2m) \leq 1/2$ beschränkt, sodass mit einer Wahrscheinlichkeit von mindestens $1/2$ das richtige Ergebnis berechnet wird.

Es bleibt die Aufgabe, ein randomisiertes OBDD zu konstruieren, das das beschriebene Verfahren umsetzt. Zunächst betrachten wir die Berechnung von $|x| \bmod p$. Die Knoten dieses

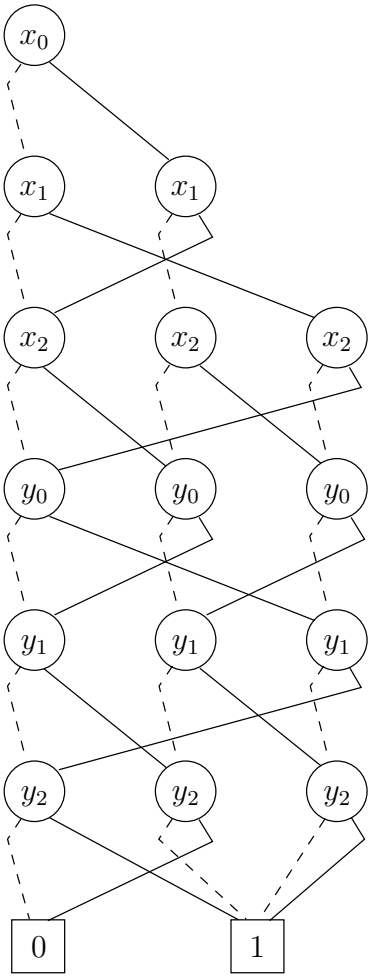


Abbildung 22: Vergleich der 3-Bit-Zahlen (x_2, x_1, x_0) und (y_2, y_1, y_0) modulo 3

OBDDs sind in p Spalten angeordnet. Wenn eine Variable x_i gelesen wird, deren Stellenwert in $|x|$ modulo p der Wert k ist, zeigen die 0-Kanten des j-ten x_i-Knoten auf den j-ten Knoten in der nächsten Ebene und die 1-Kanten auf den $((j + k) \bmod p)$-ten in der nächsten Ebene. Der Startknoten ist der Knoten in der 0-ten Spalte. Am Ende der Berechnung wird eine Senke in der ($|x| \bmod p$)-ten Spalte erreicht. Zu diesem Ergebnis kann auf analoge Weise $-|y| \bmod p$ addiert werden. Falls das Resultat 0 ist, wird verworfen, ansonsten akzeptiert. Die Größe beträgt $O(np)$.

Abbildung 22 zeigt dieses Verfahren für das Beispiel der beiden 3-Bit-Zahlen (x_2, x_1, x_0) und (y_2, y_1, y_0), die modulo 3 auf Ungleichheit getestet werden.

Wenn wir auf diese Weise (deterministische) OBDDs für jede der Primzahlen $p_1, \ldots, p_{2m}$ konstruiert haben, fügen wir diese OBDDs zu einem randomisierten OBDD zusammen. Dazu konstruieren wir einen balancierten Baum der Tiefe $\log(2m)$ von unmarkierten Knoten. Die $2m$ Blätter dieses Baumes ersetzen wir durch die Quellen der konstruierten OBDDs für $p_1, \ldots, p_{2m}$. Der Baum ist balanciert und realisiert somit eine Gleichverteilung, da $2m$ eine Zweierpotenz ist.

Es bleibt noch die Größe des konstruierten randomisierten OBDDs abzuschätzen. Nach dem

Primzahlsatz und wegen $m \leq 2n$ ist $p_{2m} = O(n \log n)$. Wir schätzen alle Primzahlen durch $O(n \log n)$ ab und erhalten damit für die Gesamtgröße $O(n^3 \log n)$.

Man sieht leicht, dass die Variablenordnung in diesem Beispiel keine Rolle spielt. Also hat NEQ für alle Variablenordnungen randomisierte OBDDs mit einseitigem beschränktem Fehler und Größe $O(n^3 \log n)$.

Auf ähnliche Weise können wir weitere Funktionen, bei denen gerechnet wird, mit randomisierten OBDDs darstellen. Als Beispiel betrachten wir noch den Graphen der Multiplikation. Dies ist die Funktion $f(x, y, z)$, die als Eingabe drei n-Bit-Zahlen x, y, z erhält und den Wert 1 ausgibt, wenn $|x||y| = |z|$. Wir beschreiben zunächst, wie $|x||y| \bmod p$ berechnen kann. Zuerst wird, wie oben beschrieben, $|x| \bmod p$ berechnet. Anschließend wird für jeden möglichen Wert von $|x| \bmod p$ in einem separaten OBDD $|y| \bmod p$ berechnet. An den Blättern jedes dieser OBDDs sind die Werte von $|x| \bmod p$ und $|y| \bmod p$ bekannt und damit auch $|x||y| \bmod p$. Für jeden dieser Werte wird nun in einem OBDD der Wert von $-|z| \bmod p$ addiert, sodass wir am Ende für jeden Wert von $(|x||y| - |z|) \bmod p$ eine separate Senke erreichen. Um den Graphen der Multiplikation zu berechnen, akzeptiert das OBDD für den Wert 0 und verwirft ansonsten. Die Breite des OBDDs ist durch p^2 beschränkt, sodass es die Größe $O(np^2)$ hat.

Nun können wir analog zu oben zufällig eine der $2m$ kleinsten Primzahlen wählen, wobei m die kleinste Zweierpotenz ist, die mindestens so groß wie $2n$ ist. Die Fehlerwahrscheinlichkeit ist wieder durch $1/2$ beschränkt, da die größte vorkommende Zahl $|x||y|$ durch 2^{2n} beschränkt ist, also höchstens $2n \leq m$ Teiler unter den gewählten Primzahlen hat. Die Gesamtgröße ist dann $O(n^2 p^2) = O(n^4 \log^2 n)$. Wir beachten, dass das konstruierte OBDD den „umgekehrten“ einseitigen Fehler hat, also mit Sicherheit eine 1 berechnet, wenn 1 auszugeben ist, aber auch eine 1 berechnen kann, wenn 0 auszugeben ist.

Wir benutzen nun eine ähnliche Technik, um zu zeigen, dass das Erfüllbarkeitsproblem für randomisierte OBDDs mit Fehler NP-vollständig ist. Dass das Problem in NP ist, ist leicht einzusehen; es genügt, eine Variablenbelegung zu raten und mit Hilfe des Algorithmus für die Auswertung von randomisierten OBDDs zu verifizieren, dass sie erfüllend ist. Wir benutzen eine Reduktion von dem folgenden als NP-vollständig bekannten Problem.

Quadratic Diophantine Equations
Eingabe: Natürliche Zahlen a^*, b^*, c^*.
Frage: Gibt es natürliche Zahlen x und y mit $a^* x^2 + b^* y = c^*$?

Seien a^*, b^*, c^* Zahlen mit Bitlänge n. Da alle Zahlen natürlich (und damit positiv) sind, haben alle Lösungen x, y auch eine Bitlänge von höchstens n. In ähnlicher Weise wie beim Graphen der Multiplikation können wir ein randomisiertes OBDD G_n in den Variablen $a_0, \ldots, a_{n-1}, b_0, \ldots, b_{n-1}, c_0, \ldots, c_{n-1}, x_0, \ldots, x_{n-1}, y_0, \ldots, y_{n-1}$ konstruieren, das polynomielle Größe hat und mit (umgekehrten) einseitigem Fehler $1/2$ die Funktion f mit

$$f(a, b, c, x, y) = 1 \Leftrightarrow |a||x|^2 + |b||y| = |c|$$

berechnet. Sei nun eine Eingabe (a^*, b^*, c^*) für Quadratic Diophantine Equations gegeben, wobei die Zahlen eine Bitlänge von höchstens n haben. Mit Ersetzung durch Konstanten (das wir für randomisierte OBDDs genauso wie für OBDDs ausführen können) konstruieren wir

aus G_n ein randomisiertes OBDD H für $g(x,y) = f_{|a=a^*,b=b^*,c=c^*}$. Diese Funktion berechnet eine 1 auf den Eingaben (x,y), für die $a^*|x|^2+b^*|y| = c^*$ ist. Also ist H genau dann erfüllbar, wenn es eine Lösung für die Eingabe (a^*,b^*,c^*) für Quadratic Diophantine Equations gibt.

Abschließend merken wir an, dass das Erfüllbarkeitsproblem für fehlerfreie randomisierte OBDDs effizient möglich ist: Wenn es einen Pfad von der Quelle zur 1-Senke gibt, muss für die zugehörige Eingabe a gelten, dass $f(a) = 1$ ist; anderenfalls würde das randomisierte OBDD für die Eingabe a mit positiver Wahrscheinlichkeit einen Fehler machen.

15 Freie BDDs

Bei den PBDDs haben wir gesehen, dass es hilfreich sein kann, in einem BDD mehrere Variablenordnungen zur Verfügung zu haben. Wir wollen jetzt ein deterministisches BDD-Modell behandeln, das es ebenfalls ermöglicht, mehrere Variablenordnungen gleichzeitig zur Verfügung zu haben.

Definition 15.1: Ein freies BDD (FBDD) ist ein BDD, in dem auf jedem Berechnungspfad jede Variable höchstens einmal getestet wird.

FBDDs werden in der Komplexitätstheorie auch als Read-once Branchingprogramme bezeichnet. Sie sind die natürliche Erweiterung von OBDDs, wenn man auf die Variablenordnungsbedingung verzichten will. Wir wollen uns zunächst anhand von zwei einfachen Beispielen davon überzeugen, dass mit polynomiellen FBDDs mehr Funktionen als mit polynomiellen OBDDs dargestellt werden können. Anschließend entwerfen wir Algorithmen für Operationen auf FBDDs bzw. zeigen, dass manche Operationen NP-schwer sind. Dies führt zu einer eingeschränkten Variante von FBDDs, den graphgesteuerten BDDs. Auch diese kann man als natürliche Erweiterung der OBDDs auffassen, da die Variablenordnung durch eine Graphordnung ersetzt wird.

Die Beispielfunktionen, die wir betrachten wollen, sind die indirekte Adressierung ISA sowie die Funktion INDEX-EQ. Wir erinnern uns, dass ISA$_n$ auf $n = 2^k$ Adress- und Datenvariablen $x_0, \ldots, x_{n-1}$, sowie auf k Adressvariablen $y_0, \ldots, y_{k-1}$ definiert ist. Der Wert der y-Variablen wählt einen Block von x-Variablen aus, der seinerseits die auszugebende x-Variable bestimmt. Wir haben in Abschnitt 3 eine exponentielle untere Schranke für OBDD-Größe bei jeder Variablenordnung bewiesen.

Die Konstruktion von FBDDs für ISA ist recht naheliegend: Wir starten mit einem vollständigen Baum der y-Variablen. Jedes Blatt entspricht einem adressierten Wert s. Falls $s \geq \lfloor n/\log n \rfloor$, wird eine 0-Senke erreicht. An den übrigen Blättern beginnt jeweils ein weiterer vollständiger Baum, an dem die x-Variablen des s-ten Blocks getestet werden. An den Blättern jedes dieser Bäume ist bekannt, welche x-Variable auszugeben ist sowie die Werte aller zuvor getesteten x-Variablen. Falls also die adressierte x-Variable zuvor getestet wurde, kann sie direkt ausgegeben werden (indem eine entsprechend markierte Senke erreicht wird), anderenfalls wird sie getestet und ausgegeben. Es ist leicht zu sehen, dass das konstruierte BDD ein FBDD ist. Es enthält $\lfloor n/k \rfloor + 1$ vollständige Bäume auf $\log n$ Variablen (die damit jeweils Größe n haben). Damit lässt sich die Gesamtgröße mit $O(n^2/\log n)$ abschätzen.

Die zweite Funktion, für die wir FBDDs konstruieren wollen, ist die Funktion INDEX-EQ$_{n,l}(x, a, b)$, die auf $n = 2^r$ Datenvariablen $x_0, \ldots, x_{n-1}$ und auf $2l \log n$ Adressvariablen definiert ist. Jeweils $\log n$ der Adressvariablen bilden eine Block, der Werte aus dem Bereich $\{0, \ldots, n-1\}$ beschreibt. Die Blöcke und ihre Werte bezeichnen wir mit $a_1, \ldots, a_l$, $b_1, \ldots, b_l$. Die Funktion nimmt den Wert 1 an, wenn die folgenden zwei Bedingungen gelten:

1. $x_{a_1} = x_{b_1} \wedge \cdots \wedge x_{a_l} = x_{b_l}$ und
2. $a_1 < \cdots < a_l < b_1 < \cdots < b_l$.

Die a- und b-Variablen bilden also Zeiger, die Paare von Datenvariablen adressieren. Diese Paare sind auf Gleichheit zu testen. Die Bedingung 2. verlangt weiterhin, dass diese Zeiger geordnet sind. Es ist eine einfache Übungsaufgabe zu zeigen, dass INDEX-EQ$_{n,l}$ mit $l = \omega(\log n)$ für keine Variablenordnung polynomielle OBDDs hat.

Wenn wir bei der folgenden Konstruktion vom „Speichern“ von a_i sprechen, meinen wir damit, dass für jeden möglichen Wert von a_i einer von n verschiedenen Teilen des FBDDs erreicht wird, so dass später bekannt ist, welchen Wert a_i hat. Das FBDD beginnt damit, die Variablen, die a_1 codieren, zu lesen, und speichert a_1. Anschließend wird dasselbe mit b_1 gemacht. Dann kann getestet werden, ob $a_1 < b_1$ gilt, falls nicht, wird eine 0-Senke erreicht. An der betrachteten Stelle sind a_1 und b_1 bekannt, also können x_{a_1} und x_{b_1} gelesen und verglichen werden. Bei Ungleichheit wird eine 0 ausgegeben, ansonsten wird die Berechnung fortgesetzt.

Wir beschreiben nun den Teil des FBDDs, der a_i und b_i liest und die adressierten Variablen vergleicht. Dabei gehen wir davon aus, dass von dem vorherigen Teil der Rechnung die Werte a_{i-1}, b_1 und b_{i-1} bekannt sind und gespeichert worden sind. Im betrachteten Teil werden a_i und b_i gelesen und gespeichert. Dann wird getestet, ob $a_i > a_{i-1}$, $b_i > b_{i-1}$ und $a_i < b_1$ gilt. Wenn nicht, wird eine 0 ausgegeben. Anderenfalls werden x_{a_i} und x_{b_i} gelesen und verglichen. Durch die vorher durchgeführten Zeigervergleiche wird sichergestellt, dass diese Variablen zuvor nicht getestet wurden, sodass die Read-once-Bedingung erfüllt ist. Wenn $x_{a_i} \neq x_{b_i}$, wird eine 0 ausgegeben, ansonsten werden an die weitere Rechnung die Werte a_i, b_1 und b_i weitergegeben. Im Fall $i = l$ wird stattdessen die 1-Senke erreicht.

Für die Abschätzung der Größe beachten wir, dass zu jedem Zeitpunkt der Wert von fünf Zeigern gespeichert wird. Dazu genügt die Breite $O(n^5)$. Damit kann die Gesamtgröße mit $O(n^6 + n^5 l \log n) = O(n^6 \log n)$ abgeschätzt werden; die letzte Abschätzung ist möglich, da die Funktion nur für $l \leq n/2$ sinnvoll ist.

Wir wollen nun überlegen, warum FBDDs ohne weitere Einschränkung als Datenstruktur für boolesche Funktionen vermutlich ungeeignet sind. Dazu betrachten wir die Komplexität von drei Operationen: dem Test, ob $f \wedge g = 0$, dem Äquivalenztest und der Minimierung. Für die ersten beiden Probleme kennen wir im Fall von OBDDs effiziente Algorithmen, während die Minimierung NP-schwer ist. Wir merken, ohne weiter in Details zu gehen, an, dass sich dieser Beweis leicht auf FBDDs übertragen lässt, sodass auch das Minimierungsproblem für FBDDs, die mehrere Funktionen darstellen, NP-schwer ist. Es ist bekannt, dass dieses Ergebnis auch für FBDDs für eine Funktion gilt und dass auch die Existenz eines polynomiellen Approximationsschemas für die Minimierung von FBDDs P=NP impliziert. Anders als bei OBDDs sind aber keine Resultate über polynomielle Approximationsalgorithmen bekannt.

Wir betrachten nun den Test auf $f \wedge g = 0$, wobei f und g durch FBDDs gegeben sind. Dieses Problem ist NP-schwer, sodass die Synthese von FBDDs wohl nicht effizient möglich ist. Wir zeigen dies mit einer Reduktion von 3-SAT. Sei eine 3CNF-Formel gegeben. Wir ersetzen in der Formel jede mehrfach vorkommende Variable x_i durch Kopien $x_{i,1}, x_{i,2}, \ldots$, sodass jede Variable nur noch einmal vorkommt. Dann ist es leicht, für die resultierende Funktion f ein FBDD linearer Größe in Bezug auf die gegebene Formel zu konstruieren. Dieses ist sogar ein OBDD, wobei die Variablenordnung so gewählt ist, dass die Variablen, die in einer Klausel stehen, auch in der Variablenordnung zusammen getestet werden. Die Funktion g

testet, ob alle Kopien jeder Variable denselben Wert haben. Dies geht ebenfalls mit einem FBDD. Auch dieses FBDD ist ein OBDD, wobei die Variablenordnung so gewählt wird, dass die Kopien einer jeden Variable zusammen angeordnet sind. Wenn es nun eine Eingabe x mit $f(x) = g(x) = 1$ gibt, erhalten wir sofort eine erfüllende Belegung der ursprünglichen Formel. Da $g(x) = 1$, haben alle Kopien einer jeden Variablen denselben Wert. Da $f(x) = 1$, ist die zugehörige Belegung der Variablen der 3CNF-Formel erfüllend. Umgekehrt entspricht jeder erfüllenden Belegung der 3CNF-Formel eine Belegung x mit $f(x) = g(x) = 1$.

Wir stellen nebenbei fest, dass der Test auf $f \wedge g = 0$ sogar für OBDDs NP-schwer ist, wenn f und g durch OBDDs mit verschiedenen Variablenordnungen dargestellt werden dürfen. Wenn f und g durch OBDD mit derselben Variablenordnung dargestellt werden, ist der Test dagegen mit Hilfe der Synthese in polynomieller Zeit möglich.

Wir kommen nun zum Äquivalenztest von FBDDs. Hierfür ist kein Beweis bekannt, der zeigt, dass das Problem NP-schwer ist. Allerdings kennt man seit 1980 nur einen randomisierten Algorithmus für dieses Problem. Wir wollen diesen Algorithmus hier vorstellen, da er eine wesentliche Idee enthält, die auch in anderen Beweisen in der theoretischen Informatik eine Rolle spielt, nämlich die Arithmetisierung von booleschen Berechnungen.

Sei $\mathbb{F}$ ein Körper. Sei G ein FBDD für die boolesche Funktion f, die über den Variablen $x_1, \ldots, x_n$ definiert ist. Wir interpretieren dieses FBDD nun auch als Darstellung einer Funktion $f^* : \mathbb{F}^n \to \mathbb{F}$, die induktiv auf die folgende Weise definiert ist. Sei $a = (a_1, \ldots, a_n) \in \mathbb{F}^n$. Die c-Senken stellen die konstanten Funktionen $c \in \{0, 1\}$ dar. Sei nun v ein x_i-Knoten mit den Nachfolgern v_0 und v_1. Dann sei

$$f_v^*(a) := (1 - a_i) \cdot f_{v_0}^*(a) + a_i \cdot f_{v_1}^*(a).$$

Dabei sind $+$, $-$ und $\cdot$ die Operationen in $\mathbb{F}$. Dies ist eine Verallgemeinerung der Shannon-Zerlegung für beliebige Körper $\mathbb{F}$. Also erhalten wir für den Körper $\mathbb{F} = \mathbb{Z}_2$ wieder die boolesche Funktion f. Wenn wir nur Elemente aus $\{0, 1\}$ einsetzen, erhalten wir auch nur Zwischenergebnisse aus $\{0, 1\}$. Wenn wir also f^* für Eingaben $a \in \{0, 1\}^n \subseteq \mathbb{F}^n$ auswerten, erhalten wir als Resultat $f(a)$. Hieraus folgt insbesondere, dass für verschiedene boolesche Funktionen g und h auch $g^* \neq h^*$ gilt.

Mit Hilfe der verallgemeinerten Shannon-Zerlegung können wir die Funktion f^* bottom-up auswerten. Ein Beispiel für ein FBDD für die Funktion $f(x_1, x_2, x_3) = x_1x_2 \vee x_3$ zeigt Abbildung 23. In der Abbildung ist auch gezeigt, wie $f^* : \mathbb{Z}_5^3 \to \mathbb{Z}_5$ für die Eingabe $(4, 3, 2)$ ausgewertet wird. Das Resultat ist $f^*(4, 3, 2) = 0$.

Wir leiten nun zwei Eigenschaften von FBDDs mit Eingaben aus $\mathbb{F}$ her.

Lemma 15.2: Seien G und H zwei FBDDs für dieselbe boolesche Funktion f. Dann stellen sie auch über jedem Körper $\mathbb{F}$ dieselbe Funktion f^* dar.

Beweis: Durch Einfügen von Dummy-Knoten machen wir die FBDDs G und H vollständig. (Es ist eine einfache Übungsaufgabe zu überlegen, wie man FBDDs vollständig machen kann, sodass die Größe nur um einen Faktor von höchstens $O(n)$ wächst.) Da für einen solchen Dummy-Knoten v der 0-Nachfolger v_0 und der 1-Nachfolger v_1 übereinstimmen,

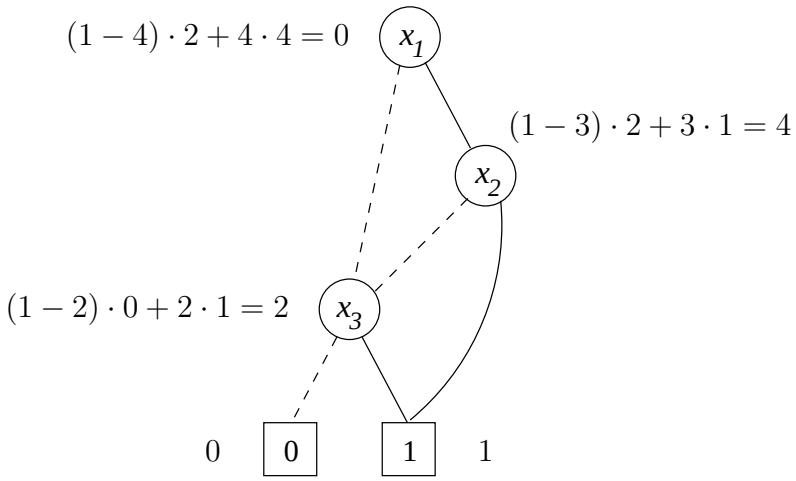


Abbildung 23: Auswertung von $f^*(4,3,2)$

folgt auch für die über $\mathbb{F}$ dargestellte Funktion, dass $f_v^*(a) = f_{v_0}^*(a) = f_{v_1}^*(a)$. Es genügt also, die Behauptung für vollständige FBDDs zu zeigen.

Wir beachten nun, dass es für jede Eingabe $b \in f^{-1}(1)$ genau einen Berechnungspfad von der Quelle zu einer Senke gibt. Ebenso gibt es für jeden Berechnungspfad von der Quelle zur 1-Senke genau eine Eingabe $b \in f^{-1}(1)$. Sei $I_c(b)$ die Menge aller i mit $b(i) = c$. Dann stellt das vollständige FBDD G bezüglich $\mathbb{F}$ die Funktion

$$f^*(x) = \sum_{b \in f^{-1}(1)} \prod_{i \in I_0(b)} (1 - x_i) \prod_{i \in I_1(b)} x_i$$

dar. Dies können wir formal mit einer Induktion über die Anzahl der Variablen zeigen. Für $n = 1$ ist die Aussage leicht zu verifizieren. Sei nun $n > 1$. Sei x_1 die an der Quelle getestete Variable. Die Induktionsvoraussetzung sagt aus, dass die Behauptung für die an den Nachfolgern der Quelle berechneten Subfunktionen $f_{|x_1=0}$ und $f_{|x_1=1}$ gilt. Durch Einsetzen der Induktionsvoraussetzung in die verallgemeinerte Shannon-Zerlegung erhalten wir:

$$\begin{aligned} f^*(x) &= (1 - x_1) \sum_{b=(0,b_2,\ldots,b_n) \in f^{-1}_{|x_1=0}(1)} \prod_{i \in I_0(b), i \neq 1} (1 - x_i) \prod_{i \in I_1(b), i \neq 1} x_i \\ &\quad + x_1 \sum_{b=(1,b_2,\ldots,b_n) \in f^{-1}_{|x_1=1}(1)} \prod_{i \in I_0(b), i \neq 1} (1 - x_i) \prod_{i \in I_1(b), i \neq 1} x_i \\ &= \sum_{b \in f^{-1}(1)} \prod_{i \in I_0(b)} (1 - x_i) \prod_{i \in I_1(b)} x_i. \end{aligned}$$

Wir können also folgern, dass f^* nur von f, nicht aber von G abhängt. Da G und H dieselbe boolesche Funktion f darstellen, stimmen auch die dargestellten Funktionen bezüglich $\mathbb{F}$ überein. □

Lemma 15.3: Seien G und H zwei FBDDs für verschiedene boolesche Funktionen g und h. Seien g^* und h^* die dargestellten Funktionen bezüglich eines endlichen Körpers $\mathbb{F}$. Dann unterscheiden sich die Funktionswerte von g^* und h^* auf mindestens $(|\mathbb{F}| - 1)^n$ Inputs.

Beweis: Wir beweisen die Aussage mit Induktion über n. Für den Induktionsanfang $n = 0$ sind g^* und h^* verschiedene konstante Funktionen, sodass die Behauptung (für die leere Eingabe) gilt. Sei nun die Behauptung für $n-1$ bewiesen. Seien G und H FBDDs auf n Variablen. Sei $x \in \mathbb{F}^n$. Dann ist aufgrund der Definition der dargestellten Funktion in FBDDs

$$\begin{aligned} g^*(x) &= (1-x_1) \cdot g^*_{|x_1=0} + x_1 \cdot g^*_{|x_1=1} \quad \text{und} \\ h^*(x) &= (1-x_1) \cdot h^*_{|x_1=0} + x_1 \cdot h^*_{|x_1=1}. \end{aligned}$$

Da $g \neq h$, folgt $g^* \neq h^*$ und damit $g^*_{|x_1=0} \neq h^*_{|x_1=0}$ oder $g^*_{|x_1=1} \neq h^*_{|x_1=1}$ oder beides. Sei o.B.d.A. $g^*_{|x_1=0} \neq h^*_{|x_1=0}$. Da dies Funktionen auf $n-1$ Variablen sind, unterscheiden sich nach Induktionsvoraussetzung die Funktionswerte für mindestens $(|\mathbb{F}|-1)^{n-1}$ Inputs. Sei nun $a' = (a_2, \ldots, a_n)$ eine Eingabe mit $g^*_{|x_1=0}(a') \neq h^*_{|x_1=0}(a')$. Damit $g^*(a_1, a') = h^*(a_1, a')$ gilt, muss für a_1 gelten, dass

$$(1-a_1) \cdot g^*_{|x_1=0}(a') + a_1 \cdot g^*_{|x_1=1}(a') = (1-a_1) \cdot h^*_{|x_1=0}(a') + a_1 \cdot h^*_{|x_1=1}(a').$$

Dies ist äquivalent zu

$$a_1(g^*_{|x_1=1}(a') - g^*_{|x_1=0}(a') + h^*_{|x_1=0}(a') - h^*_{|x_1=1}(a')) = h^*_{|x_1=0}(a') - g^*_{|x_1=0}(a').$$

Da $h^*_{|x_1=0}(a') \neq g^*_{|x_1=0}(a')$, muss auch $g^*_{|x_1=1}(a') - g^*_{|x_1=0}(a') + h^*_{|x_1=0}(a') - h^*_{|x_1=1}(a') \neq 0$ gelten. Da $\mathbb{F}$ ein Körper ist, gibt es nur eine Lösung für a_1. Damit gibt es für jeden der $(|\mathbb{F}|-1)^{n-1}$ Inputs a' mit $g^*_{|x_1=0}(a') \neq h^*_{|x_1=0}(a')$ mindestens $|\mathbb{F}|-1$ Belegungen für a_1, sodass $g(a_1, a') \neq h(a_1, a')$. Insgesamt gibt es also mindestens $(|\mathbb{F}|-1)^n$ Inputs a mit $g^*(a) \neq h^*(a)$. □

Wir erhalten für den Äquivalenztest den folgenden Algorithmus:

1. Wähle $a \in \mathbb{F}^n$ zufällig gemäß Gleichverteilung.
2. Werte die gegebenen FBDDs für a aus.
3. Falls die Resultate verschieden sind, gib aus, dass die FBDDs nicht äquivalent sind, ansonsten gib aus, dass sie vermutlich äquivalent sind.

Wie groß ist die Fehlerwahrscheinlichkeit des Algorithmus? Falls die FBDDs äquivalent sind, wird dies auch auf jeden Fall ausgegeben. Falls die FBDDs nicht äquivalent sind, ist die Wahrscheinlichkeit, dies auszugeben, mindestens $((|\mathbb{F}|-1)/|\mathbb{F}|)^n = (1-1/|\mathbb{F}|)^n$. Wenn wir für $\mathbb{F}$ den Körper $\mathbb{Z}_p$ mit $p \geq 2n$ wählen, erhalten wir für die Fehlerwahrscheinlichkeit die obere Schranke $1-(1-1/(2n))^n = 1-((1-1/(2n))^{2n})^{1/2} \leq 1-(e^{-1})^{1/2} \leq 1/2$. Wir beachten, dass diese Abschätzung für alle nicht äquivalenten Paare von FBDDs gilt.

Worin besteht das Problem der FBDDs? Wir hatten eine ähnliche Situation schon bei den PBDDs: Dort haben wir sichergestellt, dass in verschiedenen gegebenen PBDDs für eine Eingabe OBDDs mit denselben Variablenordnungen ausgewertet werden. Ebenso wollen wir nun bei den FBDDs verlangen, dass eine verallgemeinerte Variablenordnung eingehalten wird, die erzwingt, dass in verschiedenen FBDDs für dieselbe Eingabe a die Variablen in derselben Reihenfolge gelesen werden, wobei diese Reihenfolge für verschiedene Eingaben verschieden sein darf. Die Datenstruktur, die eine solche Reihenfolge für jede Eingabe angibt, heißt Graphordnung.

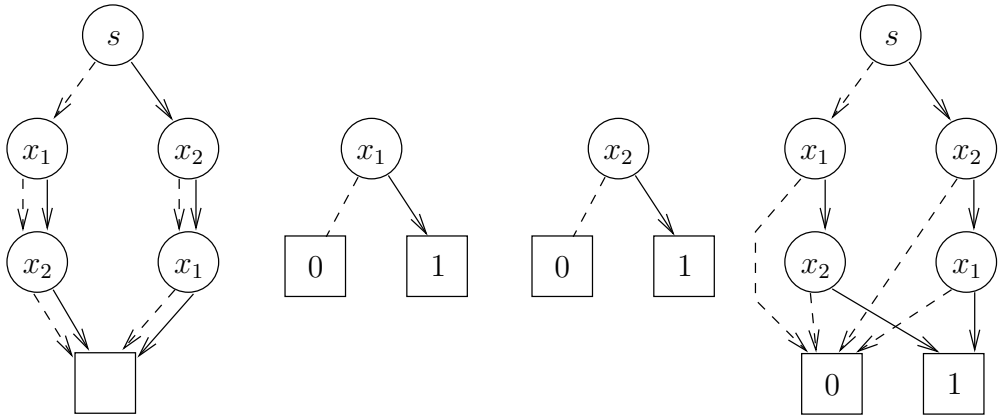


Abbildung 24: Ein Beispiel für Synthese bei G-FBDDs

Definition 15.4: Eine Graphordnung G über den Variablen $x_1, \ldots, x_n$ ist ein vollständiges FBDD mit nur einer (unmakierten) Senke. Zu einer Eingabe a bezeichnet $G(a)$ die Reihenfolge der Variablen auf dem Berechnungspfad für a. Ein FBDD H heißt G-gesteuertes FBDD oder kurz G-FBDD, falls für alle Eingaben a gilt, dass die Variablen in H in der Reihenfolge $G(a)$ getestet werden, wobei Tests ausgelassen werden dürfen.

Wir sehen insbesondere, dass OBDDs ein Spezialfall von G-FBDDs sind, nämlich die Situation, dass die Graphordnung G ein „Liste“ ist, sodass sich für alle Eingaben dieselbe Variablenordnung ergibt. Wir wollen nun einige Eigenschaften von G-FBDDs sammeln. Zunächst beobachten wir, dass die Reduktionsregeln für OBDDs auch für G-FBDDs anwendbar sind. Weiterhin ist es einfach, zu einem FBDD H eine Graphordnung G anzugeben, sodass H ein G-FBDD ist: Wir machen H vollständig und verschmelzen die Senken zu einer unmarkierten Senke. Das Resultat ist die Graphordnung G und es ist leicht zu sehen, dass H ein G-FBDD ist.

Man kann ebenso wie für OBDDs zeigen, dass jede Funktion f für eine feste Graphordnung G ein bis auf Isomorphie eindeutiges G-FBDD hat. Weiterhin kann man einen Linearzeitalgorithmus für die Reduktion konstruieren. Auf den Beweis dieser Aussagen wollen wir hier verzichten. Es folgt aber, dass der Äquivalenztest wie bei den OBDDs ein Isomorphietest der reduzierten Graphen ist und damit in linearer Zeit möglich ist. Auch die Übertragung des OBDD-Algorithmus für Erfüllbarkeit-Anzahl ist einfach.

Bei der Synthese stoßen wir auf ein neues Problem. Wir betrachten die Graphordnung G in Abbildung 24. Wenn $s = 0$, werden die x-Variablen in der Reihenfolge x_1, x_2 getestet, für $s = 1$ in der umgekehrten Reihenfolge. Weiterhin sind zwei G-FBDDs für die Funktionen x_1 und x_2 angegeben. Wenn wir nun ein G-FBDD für $x_1 \wedge x_2$ berechnen wollen, genügt die Produktgraphkonstruktion nicht, da wir nun auch s testen müssen um festzustellen, ob x_1 vor x_2 zu testen ist oder umgekehrt. Also müssen wir die Produktgraphkonstruktion erweitern.

Die wesentliche Idee besteht darin, den Produktgraphen nicht nur von den beiden gegebenen G-FBDDs H und J zu konstruieren, sondern von allen drei Graphen G, H und J. Sei (u, v, w) ein Knoten des Produktgraphen, wobei u ein Knoten aus der Graphordnung G ist.

Seien u_0 und u_1 die beiden Nachfolger von u, die Nachfolger von v und w seien analog bezeichnet. Wir unterscheiden die folgenden Fälle.

1. u, v und w sind mit x_i markiert. Dann ist auch (u, v, w) mit x_i markiert, die Nachfolger sind (u_0, v_0, w_0) und (u_1, v_1, w_1).

2. u ist mit x_i markiert, v und w sind mit anderen Variablen x_v und x_w markiert. Die Idee ist nun, an v und w zu warten. Also ist (u, v, w) mit x_i markiert, die Nachfolger sind (u_0, v, w) und (u_1, v, w). Die Fälle, wo einer der beiden Knoten v oder w mit x_i markiert ist oder wo v und/oder w Senke ist, gehen analog.

3. u, v und w sind Senken, wobei v und w mit c_v und c_w markiert sind. Dann ist (u, v, w) eine Senke, die mit $c_v \otimes c_w$ markiert ist.

4. u ist eine Senke, v und/oder w aber nicht. Beim simultanen Durchlauf durch G, H und J kann dies nicht vorkommen, da G vollständig ist. Also ist (u, v, w) von der Quelle des Produktgraphen nicht erreichbar und braucht daher nicht erzeugt zu werden.

Wir können nun alle Implementierungstricks, die wir bei der OBDD-Synthese beschrieben haben, ebenfalls hier anwenden. Die Rechenzeit und damit auch die Größe der Ausgabe ist durch $O(|G||H||J|)$ beschränkt. Wir beachten aber, dass z.B. für den Fall, dass H und J vollständig sind, die Situation, dass in H oder J gewartet wird, niemals eintritt. In diesem Fall entfällt der Faktor $|G|$ in der Rechenzeit. Die Situation aus dem Bild oben, dass wir Extraknoten brauchen, um die Variablenordnung zu bestimmen, tritt dann nicht ein (da diese Extraknoten schon in H und J enthalten sind). Wir hatten oben angemerkt, dass die Größe eines FBDDs um einen Faktor $O(n)$ wachsen kann, wenn man es in ein vollständiges FBDD umwandelt. Hier müssen wir bei der Umformung in ein vollständiges FBDD jedoch auch die Graphordnung G beachten. Dabei kann die Größe um $O(|G|)$ wachsen. Die wesentliche Idee für diese Umformung besteht darin, den Synthesealgorithmus anzuwenden (Übungsaufgabe).

Als letzte Operation betrachten wir wieder Ersetzen durch Konstanten. Da wir hier wie bei den PBDDs verschiedene Variablenordnungen für verschiedene Inputs erlauben, können wir dieselben Schwierigkeiten wie dort erwarten. Wir benutzen also dasselbe Beispiel: Sei f die Funktion, die die Variablen einer Matrix darauf testet, ob die Matrix eine Zeile enthält, die nur aus Einsen besteht. Sei g die Funktion, die die Matrix auf eine Spalte testet, die nur aus Einsen besteht. Wir betrachten wieder die Funktion $\bar{s}f \vee sg$. In der Graphordnung G entscheidet die Variable s zu Beginn, ob die Variablenordnung eine zeilenweise oder eine spaltenweise Variablenordnung ist. Dann hat die Funktion $\bar{s}f \vee sg$ ein G-FBDD linearer Größe. Wenn wir nun z.B. die Variable s durch 0 ersetzen, dürfen wir die Graphordnung nicht ändern. Da $(\bar{s}f \vee sg)_{|s=0} = f$, muss im Fall $s = 1$ die Funktion f mit einer spaltenweisen Variablenordnung dargestellt werden. Wir haben bereits bei den PBDDs erwähnt, dass dies eine Darstellung exponentieller Größe erfordert.

Ähnlich wie bei den PBDDs gibt es wieder einen Spezialfall, in dem Ersetzen durch Konstanten und die daraus abgeleiteten Operationen ohne Vergrößerung der Darstellung ausgeführt werden können. Wir bezeichnen eine Graphordnung G als *x_i-oblivious*, wenn für alle x_i-Knoten in G der 0- und der 1-Nachfolger übereinstimmen. Damit hängt in G-FBDDs die

Variablenordnung nicht von dem Wert von x_i ab, so dass die Ersetzung von x_i durch Konstanten wie in OBDDs durchgeführt werden kann.

16 Einfache Methoden zum Beweis unterer Schranken für BDDs

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 16.1: Die Anzahl der Funktionen $f : \{0,1\}^n \to \{0,1\}$, die BDDs mit s Knoten (einschließlich Senken) 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. □

Satz 16.2: 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. □

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 16.3: Sei $f : \{0,1\}^n \to \{0,1\}$ eine Funktion, die von allen n Variablen essentiell abhängt, und seien $S_1, \ldots, S_k$ 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 $\text{BDD}_f \geq t_1 + \cdots + t_k + 2$, 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 $t_i + 2$ Knoten aus f berechnen, da alle Variablen außerhalb von S_i konstantgesetzt werden. Aus dem obigen Lemma folgt dann

$$s_i \leq |S_i|^{t_i+2}((t_i+2)!)^2 \leq t_i^{t_i+2}((t_i+2)!)^2 \leq (t_i)^{4t_i},$$

wobei die letzte Abschätzung nur für hinreichend großes t_i gilt. Hieraus folgt die behauptete untere Schranke für t_i. □

Wir wenden die Methode jetzt auf die Funktion ISA an. Sei $n = 2^l$. Für die Mengen S_i wählen wir die Blöcke aus x-Variablen 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 mindestens 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 16.4: $\mathrm{BDD}_{\mathrm{ISA}_n} = \Omega(n^2/\log^2 n)$.

Wir merken an, dass wir bereits FBDDs der Größe $O(n^2/\log n)$ für ISA_n konstruiert haben. Damit haben wir die BDD-Größe von ISA_n bis auf einen Faktor von $O(\log n)$ genau bestimmt.

17 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 $f : X \times Y \to \{0,1\}$, 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 müssen beide Spieler 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 17.1: Die Kommunikationskomplexität von $f : X \times Y \to \{0,1\}$ *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 $f(x,y)$ berechnen und ausgeben (und damit an Alice schicken). Die Anzahl der gesendeten Bits ist gleich $\lceil \log |X| \rceil + 1$. 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 $|x| - |y| \equiv 0 \bmod 5$ ist. Dann genügt es, dass Alice nicht die ganze Zahl x sendet, sondern nur $|x| \bmod 5$. Dazu genügen 3 Bits. Damit kann Bob dann testen, ob $|x| - |y| \equiv 0 \bmod 5$ 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 naheliegende 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 sogenannten Protokollbäume.

Definition 17.2: Sei $f : X \times Y \to \{0,1\}$. 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 $g_v : X \to \{0,1\}$, 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 $g_v : Y \to \{0,1\}$ 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 17.3: Die Kommunikationsmatrix M_f von $f : X \times Y \to \{0,1\}$ 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 25 gezeigt.

Definition 17.4: 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 25. 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.

y_1, y_2, y_3 \ x_1, x_2, x_3	000	001	010	011	100	101	110	111
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 25: Die Kommunikationsmatrix und ein monochromatisches Rechteck für die Funktion DQF_3

Lemma 17.5: 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. □

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 17.6: 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 $M_f = \sum_{l\ 1-\text{Blatt}} M_l$. Da der Rang subadditiv ist, folgt,

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

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. □

Für das Beispiel der Gleichheitsfunktion ist wieder leicht zu sehen, dass der Rang der Kommunikationsmatrix gleich 2^n ist. Der Rang der Kommunikationsmatrix von

$$\text{DQF}_n(x_1, \ldots, x_n, y_1, \ldots, y_n) := x_1y_1 \vee \cdots \vee x_ny_n$$

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 17.7: Eine Menge $S \subseteq X \times Y$ heißt c-Unterscheidungsmenge (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-Unterscheidungsmenge 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 der Definition der Unterscheidungsmengen nicht monochromatisch. Widerspruch. Damit folgt die folgende Aussage.

Satz 17.8: Die Größe einer c-Unterscheidungsmenge 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 Unterscheidungsmengen auf die Gleichheitsfunktion anzuwenden. Stattdessen betrachten wir die Funktion DQF_n. Wir behaupten, dass die Menge $\{(x, \bar{x}) | x \in \{0,1\}^n\}$ eine 0-Unterscheidungsmenge ist, wobei $\bar{x}$ die komponentenweise Negation von x bezeichnet. Man sieht sofort, dass $\mathrm{DQF}(x, \bar{x}) = 0$ 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 $(x', \bar{x})$ oder das Paar $(x, \bar{x}')$ Einsen an derselben Position. Also haben wir eine Unterscheidungsmenge mit 2^n Elementen gefunden und es folgt eine untere Schranke von n für die Kommunikationskomplexität von DQF_n.

Wie kann man nun untere Schranken für die Kommunikationskomplexität nutzen, um untere Schranken für die OBDD-Größe zu zeigen? Wir betrachten wieder eine Funktion $f : \{0,1\}^n \times \{0,1\}^m \to \{0,1\}$. 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 $\lceil \log s \rceil + 1$ 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 $\lceil \log s \rceil + 1$ 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 Abschnitt 3 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 sogenannte Einweg-Kommunikationskomplexität kann man in manchen Fällen größere untere Schranken beweisen. Anders als bei der Mehrweg-Kommunikationskomplexität zählen wir bei der Einweg-Kommunikationskomplexität die Ausgabe von Bob für die Kommunikation nicht mit, es genügt also, dass Bob den Funktionswert erfährt.

Definition 17.9: Sei $f : X \times Y \to \{0,1\}$. Eine Menge $S \subseteq X$ heißt Einweg-Unterscheidungsmenge, wenn es für $x, x' \in S$ mit $x \neq x'$ ein $y \in Y$ gibt, sodass $f(x,y) \neq f(x',y)$ gilt.

Satz 17.10: Sei $f : X \times Y \to \{0,1\}$ und sei S eine Einweg-Unterscheidungsmenge für f. Jedes Kommunikationsprotokoll für f, bei dem nur Alice Information an Bob schickt und Bob nur den Funktionswert ausgibt, 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. □

Die Funktion INDEX spielt bei Beweisen von unteren Schranken für die OBDD-Größe eine wichtige Rolle. Dabei wird die Situation betrachtet, dass Alice die Datenvariablen und Bob die Adressvariablen bekommt. Dann ist $\{0,1\}^n$ eine Einweg-Unterscheidungsmenge für INDEX_n: Seien $x, x' \in \{0,1\}^n$, sodass $x \neq x'$. Dann gibt es eine Position i, für die sich x und x' unterscheiden. Damit folgt $\mathrm{INDEX}(x,i) \neq \mathrm{INDEX}(x',i)$. Die Einweg-Kommunikationskomplexität 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.

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 Variablenordnung entsteht. In Kapitel 3 haben wir bei der Funktion ISA die y-Variablen so konstant gesetzt, dass die INDEX-Funktion mit der schlechten Variablenordnung entsteht. Diese Vorgehensweise wollen wir weiterführen. Zunächst stellen wir ein Werkzeug für die Übertragung von unteren Schranken für die Kommunikationskomplexität zwischen verschiedenen Funktionen vor.

Definition 17.11: Seien $f : X \times Y \to \{0,1\}$ und $g : A \times B \to \{0,1\}$ Funktionen. Wir sagen, dass g rechteckreduzierbar auf f ist ($g \leq_{\mathrm{rect}} f$), wenn es Funktionen $p : A \to X$ und $q : B \to Y$ gibt, sodass für alle $(a,b) \in A \times B$ gilt: $f(p(a), q(b)) = g(a,b)$.

Satz 17.12: Falls die Kommunikationskomplexität von $g : A \times B \to \{0,1\}$ mindestens s beträgt und $g \leq_{\mathrm{rect}} f$, 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. □

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 $f : \{0,1\}^n \to \{0,1\}$ und sei (L,R) eine Partition der Variablenmenge von f. Dann bezeichnen wir mit $f^{(L,R)} : \{0,1\}^{|L|} \times \{0,1\}^{|R|} \to \{0,1\}$ dieselbe Funktion, wobei Alice die Variablen aus L und Bob die Variablen aus R bekommt.

Satz 17.13: Sei n gerade und sei $f : \{0,1\}^n \to \{0,1\}$ eine Funktion über der Variablenmenge Z. Falls es für jede Partition (L,R) von Z mit $|L| = |R|$ eine Funktion $g : A \times B \to \{0,1\}$ 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 $g : A \times B \to \{0,1\}$ 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 gesamte Kommunikation genügen $s-1$ Bits im Widerspruch zur angenommenen unteren Schranke für die Kommunikationskomplexität von g. □

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

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

Satz 17.14: Jedes OBDD für HWB_n hat $2^{\Omega(n)}$ Knoten.

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

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

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

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

Wir definieren nun die Funktionen $p : A \to \{0,1\}^{|L|}$ und $q : B \to \{0,1\}^{|R|}$ der Rechteckreduktion von $\text{INDEX}_{n/8}$ auf $\text{HWB}_n^{(L,R)}$, 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 $b_0, \ldots, b_{\log n-4}$ die Adressvariablen von INDEX$_{n/8}$, die einen Wert $b \in \{0, \ldots, n/8-1\}$ 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 $z_{n/2+1}, \ldots, z_n$ befinden sich mindestens $n/4$ der Variablen in L.

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

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

Wir definieren wieder die Funktionen $p : A \to \{0,1\}^{|L|}$ und $q : B \to \{0,1\}^{|R|}$ der Rechteckreduktion von INDEX$_{n/8}$ auf HWB$_n^{(L,R)}$. 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 $b_0, \ldots, b_{\log n-4}$ die Adressvariablen von INDEX$_{n/8}$, die einen Wert $b \in \{0, \ldots, n/8-1\}$ 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 INDEX$_{n/8}$ gleich $n/8$ ist, folgt mit Satz 17.13 die behauptete untere Schranke für die OBDD-Größe für HWB$_n$ und alle Variablenordnungen. □

18 Nichtdeterministische Kommunikationskomplexität

Das Kommunikationsspiel aus dem vorherigen Abschnitt kann leicht für nichtdeterministische Berechnungsmodelle erweitert werden. Alice und Bob sind jetzt nichtdeterministische Rechner, d.h., sie dürfen jetzt „raten“. Falls $f(x, y) = 1$, muss es mindestens einen Rechenweg geben, sodass die Ausgabe 1 produziert wird; falls $f(x, y) = 0$, müssen alle Rechenwege verwerfen. Die nichtdeterministische Kommunikationskomplexität wird dann analog zu Definition 17.1 definiert, wobei wir, wie bei Nichtdeterminismus üblich, in dem Fall, dass es für eine Eingabe mehrere akzeptierende Berechnungen gibt, die Berechnung mit der kleinsten Komplexität betrachten.

Definition 18.1: Die nichtdeterministische Kommunikationskomplexität von $f : X \times Y \to \{0, 1\}$ *für ein gegebenes Protokoll P und eine Eingabe* $(x, y) \in f^{-1}(1)$ ist die minimale Anzahl von Bits, die das Protokoll für die Auswertung von f benötigt, wobei das Minimum über alle akzeptierenden Berechnungen gebildet wird. Für eine Eingabe $(x, y) \in f^{-1}(0)$ wird die nichtdeterministische Kommunikationskomplexität als 0 definiert. Die nichtdeterministische Kommunikationskomplexität von f *für das Protokoll P* ist das Maximum der nichtdeterministischen Kommunikationskomplexität von f für P und eine Eingabe (x, y), wobei das Maximum über alle Eingaben (x, y) gebildet wird. Die *nichtdeterministische Kommunikationskomplexität von f* ist die minimale nichtdeterministische Kommunikationskomplexität von f für ein Protokoll P, wobei das Minimum über alle Protokolle P gebildet wird.

Analog zu den deterministischen Protokollen können wir auch für den nichtdeterministischen Fall Protokollbäume definieren. In diesen Protokollbäumen dürfen die Spieler „raten“, indem sie an ihren Knoten die ausgehende Kante (und damit die zugehörige Kommunikation) nicht deterministisch berechnen, sondern nichtdeterministisch raten. Die Tiefe eines solchen Protokollbaumes beschreibt dann wieder die Anzahl der ausgetauschten Bits.

Lemma 18.2: Sei $f : X \times Y \to \{0, 1\}$. Die minimale Tiefe eines nichtdeterministischen Protokollbaums für f ist gleich der nichtdeterministischen Kommunikationskomplexität von f.

Beweis: Die Konstruktion eines nichtdeterministischen Protokolls für f mit s Bits an Kommunikation aus einem nichtdeterministischen Protokollbaum der Tiefe s ist offensichtlich. Sei nun ein nichtdeterministisches Protokoll für f mit der Komplexität s gegeben. Wenn wir auf die naheliegende Weise einen Protokollbaum für f konstruieren, kann dieser eine Tiefe von mehr als s haben, da bei der Definition der nichtdeterministischen Kommunikationskomplexität das Minimum über alle akzeptierenden Berechnungen gebildet wird. Das heißt aber auch, dass es für Eingaben, für die es Berechnungen gibt, die nach der Kommunikation von mehr als s Bits akzeptieren, auch akzeptierende Berechnungen mit höchstens s Bits an Kommunikation gibt. Wir dürfen also in dem Protokollbaum alle Pfade nach s Kanten „abschneiden“ und die letzte Kante auf ein 0-Blatt zeigen lassen. □

Der Beweis von Lemma 17.5 zeigt nun wieder, dass die Menge der Eingaben (x, y), für die ein Knoten v im Protokollbaum erreicht wird, ein kombinatorisches Rechteck ist und dass

dieses Rechteck monochromatisch ist, wenn v ein 1-Blatt ist. Der wesentliche Unterschied zum deterministischen Fall besteht nun darin, dass es für eine 1-Eingabe mehrere akzeptierende Pfade im Protokollbaum geben kann, also für diese Eingabe auch mehrere Blätter erreicht werden können.

Im nichtdeterministischen Fall ist das Einweg-Kommunikationsmodell keine Einschränkung mehr: Alice kann zu Beginn die gesamte Kommunikation eines Protokolls, das aus mehreren Runden besteht, raten. Anschließend überprüft sie, ob bei ihrer Eingabe diese Kommunikation möglich ist und wenn ja, schickt sie alles an Bob. Dann kann Bob ebenfalls prüfen, ob die Kommunikation bei seiner Eingabe möglich ist und ob er bei der Kommunikation und seiner Eingabe akzeptieren würde. Wenn ja, akzeptiert er. Es genügt also, nur Protokollbäume zu betrachten, bei denen nur die unmittelbaren Vorgänger der Blätter Bob zugeordnet sind und alle anderen inneren Knoten Alice zugeordnet sind.

Zur Veranschaulichung betrachten wir die Ungleichheitsfunktion
$\text{NEQ}_n : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}$, die auf der Eingabe (x,y) genau dann eine 1 berechnet, wenn x und y nicht gleich sind. Die Kommunikationsmatrix ist eine $2^n \times 2^n$-Matrix, die auf der Hauptdiagonalen Nullen und ansonsten nur Einsen enthält. Die Menge $\{(x,x) \mid x \in \{0,1\}^n\}$ ist eine 0-Unterscheidungsmenge, also beträgt die deterministische Kommunikationskomplexität mindestens n. Ein nichtdeterministisches Kommunikationsprotokoll arbeitet folgendermaßen: Alice rät eine Position $i \in \{1,\ldots,n\}$ und sendet i und x_i an Bob. Bob kann dann verifizieren, dass $x_i \neq y_i$, und das Resultat ausgeben. Die Anzahl der gesendeten Bits ist $\lceil \log n \rceil + 2$. Der zugehörige Protokollbaum und die Kommunikationsmatrix für den Fall $n = 3$ sind in Abbildung 26 gezeigt. Wenn Alice $i = 1$ rät und $x_1 = 0$, $y_1 = 1$ ist, akzeptiert das Protokoll. Diese Situation entspricht dem gestrichelten Rechteck oben rechts in der Kommunikationsmatrix. Der Zusammenhang zwischen den akzeptierenden Blättern des Protokollbaumes und den zugehörigen Rechtecken ist durch die Umrandung der Rechtecke bzw. Blätter angedeutet.

Allerdings wird zum Beispiel die Eingabe $x = (0,0,0)$, $y = (1,1,1)$ auch akzeptiert, wenn Alice $i = 2$ oder $i = 3$ rät. Also wird diese Eingabe von mehreren 1-Rechtecken überdeckt. Im Gegensatz zur Partitionierung der Kommunikationsmatrix im deterministischen Fall sprechen wir hier von einer Überdeckung. Dies führt zu der folgenden Definition.

Definition 18.3: Sei $f : X \times Y \to \{0,1\}$ eine Funktion. Dann bezeichnet $N^1(f)$ die minimale Anzahl von 1-Rechtecken, um alle Einsen der Kommunikationsmatrix zu überdecken. Weiterhin sei $C^1(f) = \log N^1(f)$.

$C^1(f)$ ist eine untere Schranke für die nichtdeterministische Kommunikationskomplexität von f. Falls ein nichtdeterministisches Kommunikationsprotokoll mit s Bits an Kommunikation auskommt, kann der zugehörige Protokollbaum nicht mehr als 2^s 1-Blätter haben. Die zugehörigen 1-Rechtecke überdecken alle 1-Eingaben. Also folgt $N^1(f) \leq 2^s$ bzw. $s \geq C^1(f)$.

Nun können wir mit einer bekannten Methode untere Schranken für die nichtdeterministische Kommunikationskomplexität zeigen.

Lemma 18.4: Sei S eine 1-Unterscheidungsmenge für f. Dann ist $N^1(f) \geq |S|$.

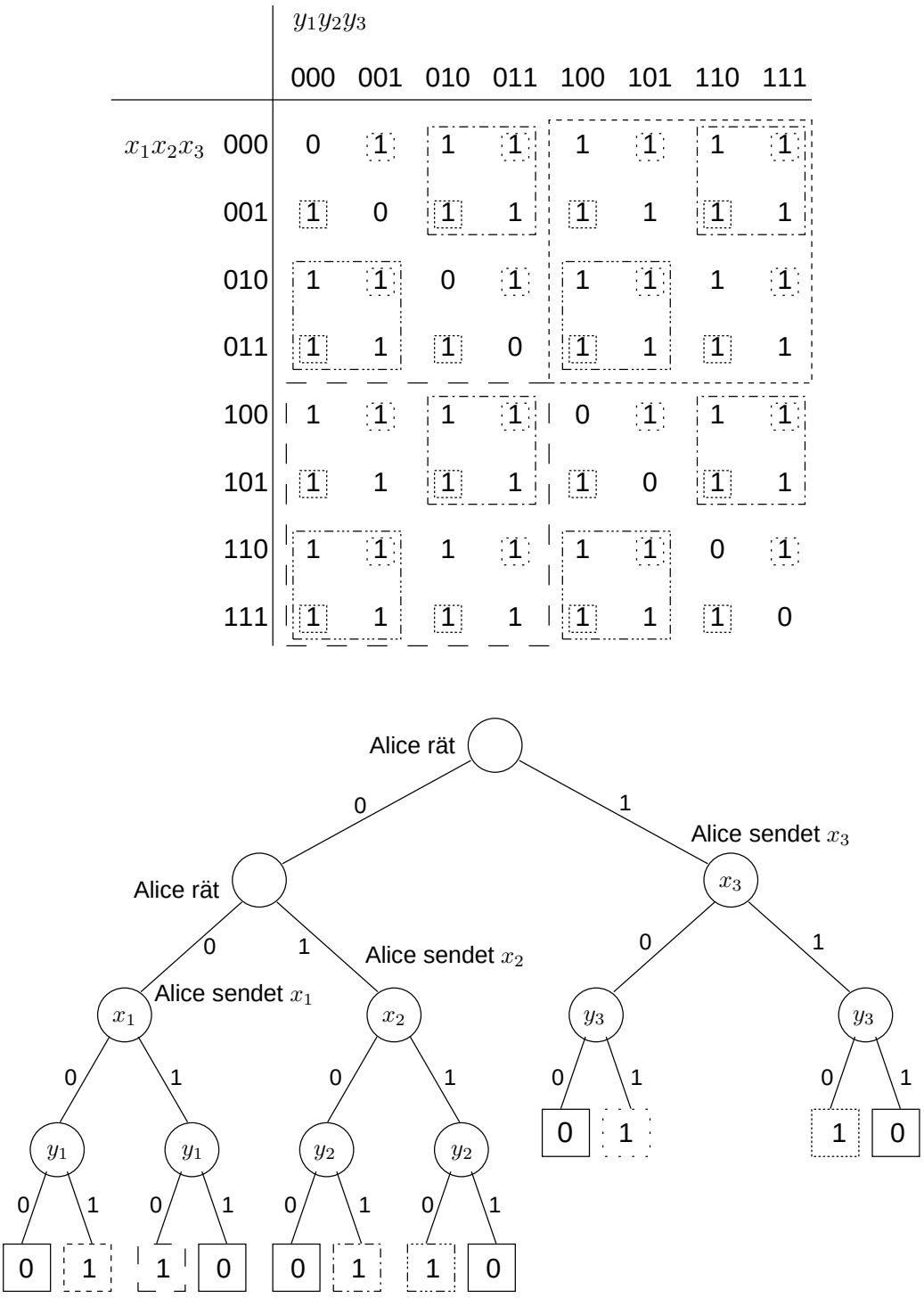


Abbildung 26: Die Kommunikationsmatrix und ein Protokollbaum für die Funktion NEQ_3

Beweis: Aus der Definition der Unterscheidungsmengen folgt, dass zwei verschiedene Elemente von S nicht von demselben 1-Rechteck überdeckt werden können. □

Wir beachten, dass diese Aussage nicht für 0-Unterscheidungsmengen gilt; ein Gegenbeispiel ist die Funktion NEQ$_n$, für die $\{(x,x) \mid x \in \{0,1\}^n\}$ eine 0-Unterscheidungsmenge der Größe 2^n ist, für die wir aber ein nichtdeterministisches Kommunikationsprotokoll mit $O(\log n)$ Bits an Kommunikation konstruiert haben. Der zugehörige Protokollbaum hat $2n$ mit 1 markierte Blätter (für jedes geratene $i \in \{1,\ldots,n\}$ gibt es zwei mit 1 markierte Blätter, nämlich für die Fälle $x_i = 0, y_i = 1$ und $x_i = 1, y_i = 0$. Die Übertragung der Rangmethode erfordert dagegen weitere Ideen, sodass wir darauf hier verzichten wollen.

Wir wollen nun eine exponentielle untere Schranke für die Größe von nichtdeterministischen OBDDs beweisen. Wir benutzen dazu die untere Schranke n für die nichtdeterministische Kommunikationskomplexität der Gleichheitsfunktion, die aus Lemma 18.4 folgt, da die Menge $\{(x,x) \mid x \in \{0,1\}^n\}$ eine 1-Unterscheidungsmenge für die Gleichheitsfunktion ist.

Wie im deterministischen Fall soll diese Schranke für alle Variablenordnungen gelten. Wir erweitern dazu Satz 17.13 für den nichtdeterministischen Fall.

Satz 18.5: Sei N gerade und sei $f : \{0,1\}^N \to \{0,1\}$ eine Funktion über der Variablenmenge Z. Falls es für jede Partition (L,R) von Z mit $|L| = |R|$ eine Funktion $g : A \times B \to \{0,1\}$ mit nichtdeterministischer Kommunikationskomplexität mindestens s und eine Rechteckreduktion $g \leq_{\text{rect}} f^{(L,R)}$ gibt, beträgt die Größe von nichtdeterministischen OBDDs für f und alle Variablenordnungen mehr als 2^{s-1}.

Der Beweis geht genauso wie bei Satz 17.13. Wir beweisen die untere Schranke für die Funktion INDEX-EQ$_{n,l}(x,a,b)$ mit $l = n/(4\log n)$ (was o.B.d.A. eine ganze Zahl ist). Die Variablenzahl N ist dann gleich $3n/2$. Die Funktion haben wir in Abschnitt 15 definiert.

Satz 18.6: Nichtdeterministische OBDDs für INDEX-EQ$_{n,n/(4\log n)}$ haben $2^{\Omega(n/\log n)}$ Knoten.

Beweis: Sei eine Partition (L,R) der Variablenmenge von INDEX-EQ$_{n,n/(4\log n)}$ gegeben. Für g wählen wir die Gleichheitsfunktion EQ$_l(y,z)$ mit $l = n/(4\log n)$. Wir geben nun die Rechteckreduktion auf INDEX-EQ$^{(L,R)}_{n,n/(4\log n)}$ an. Sei also (y,z) die Eingabe für EQ$_l$. Wir beachten zunächst, dass es aufgrund der Wahl von l genau $n/2 = N/3$ Adressvariablen für INDEX-EQ$_{n,n/(4\log n)}$ gibt. Da $|L| = |R| = N/2$, enthalten sowohl L als auch R jeweils mindestens $N/6$ Datenvariablen. Für hinreichend großes n ist $l \leq N/12$, also gilt mindestens eine der beiden folgenden Aussagen:

1. $|L \cap \{x_0,\ldots,x_{n/2-1}\}| \geq l$ und $|R \cap \{x_{n/2},\ldots,x_{n-1}\}| \geq l$,
2. $|R \cap \{x_0,\ldots,x_{n/2-1}\}| \geq l$ und $|L \cap \{x_{n/2},\ldots,x_{n-1}\}| \geq l$.

Im ersten Fall werden von der Rechteckreduktion die ersten l der Variablen in $L \cap \{x_0, \ldots, x_{n/2-1}\}$ mit y und die ersten l der Variablen in $R \cap \{x_{n/2}, \ldots, x_{n-1}\}$ mit z belegt. Die Adressvariablen werden so belegt, dass von den a-Zeigern die gewählten Variablen in $L \cap \{x_0, \ldots, x_{n/2-1}\}$ und von den b-Zeigern die gewählten Variablen in $R \cap \{x_{n/2}, \ldots, x_{n-1}\}$ adressiert werden. Im zweiten Fall werden die ersten l der Variablen in $R \cap \{x_0, \ldots, x_{n/2-1}\}$ mit z belegt und die ersten l der Variablen in $L \cap \{x_{n/2}, \ldots, x_{n-1}\}$ mit y. Die Adressvariablen werden so belegt, dass von den a-Zeigern die gewählten Variablen in $R \cap \{x_0, \ldots, x_{n/2-1}\}$ und von den b-Zeigern die gewählten Variablen in $L \cap \{x_{n/2}, \ldots, x_{n-1}\}$ adressiert werden. In beiden Fällen bekommen die übrigen Datenvariablen den Wert 0. Diese etwas umständliche Wahl stellt sicher, dass nur Variablen aus L Werte von y bekommen und nur Variablen aus R Werte von z, also dass es sich wirklich um eine Rechteckreduktion handelt. Diese Zuordnung ermöglicht es auch, die Adressvariablen so zu wählen, dass die Ordnungsbedingung 2. aus der Definition von INDEX-EQ erfüllt ist. Insgesamt ist sichergestellt, dass INDEX-EQ genau dann eine 1 berechnet, wenn y und z gleich sind. Also haben wir die gewünschte Rechteckreduktion von EQ_l auf $\text{INDEX-EQ}_{n,n/(4\log n)}$ angegeben, und die untere Schranke folgt aus Satz 18.5. □

19 Untere Schranken für FBDDs und PBDDs

Die Methoden zum Beweis unterer Schranken für OBDDs können wir nicht einfach auf FBDDs oder PBDDs übertragen, da in FBDDs oder PBDDs die Variablen für verschiedene Eingaben in verschiedenen Reihenfolgen getestet werden können, so dass wir nicht mehr einen Teil des FBDDs oder PBDDs durch Alice und den anderen durch Bob simulieren lassen können. Daher benötigen wir neue Ideen. Wir beginnen mit einer einfachen Methode zum Beweis unterer Schranken für deterministische FBDDs.

Definition 19.1: Sei $f \in B_n$ über den Variablen der Menge V definiert, und sei $1 \leq k \leq n$. Die Funktion f heißt k-mixed, wenn für alle Teilmengen $W \subseteq V$ mit $|W| = k$ gilt, dass die 2^k Belegungen der Variablen in W zu 2^k verschiedenen Subfunktionen von f führen.

Lemma 19.2: Sei f k-mixed. Die FBDD-Größe von f beträgt dann mindestens $2^k - 1$.

Beweis: Wir zeigen, dass jedes FBDD für f mit einem vollständigen Baum der Tiefe $k-1$ beginnt, woraus die behauptete untere Schranke für die FBDD-Größe folgt. Seien p und p' zwei verschiedene Pfade in einem FBDD für f, die beide eine Länge von höchstens $k-1$ haben und die beide an der Quelle beginnen. Wir zeigen, dass p und p' nicht zum selben Knoten führen. Annahme: p und p' führen zum selben Knoten v. Falls auf p und p' dieselbe Menge von Variablen getestet wird, führen zwei verschiedene Belegungen von höchstens $k-1$ Variablen zur selben Subfunktion von f, was ein Widerspruch zur Definition von k-mixed ist. Also werden auf p und p' verschiedene Mengen von Variablen getestet. Sei x_i eine Variable, die o.B.d.A. auf p', nicht aber auf p getestet wird. Aufgrund der Definition der FBDDs darf x_i nicht an v oder einem Nachfolger von v getestet werden, also hängt f_v von x_i nicht essentiell ab. Wir ergänzen nun die zu p gehörenden Variablenbelegung durch $x_i = 0$ bzw. $x_i = 1$ und erhalten in beiden Fällen dieselbe Subfunktion, nämlich f_v. Es gibt also zwei verschiedene Belegungen von höchstens k Variablen, sodass dieselbe Subfunktion entsteht. Widerspruch. □

Als Beispiel benutzen wir die Funktion $\oplus cl_{n,3}$, die wir in Abschnitt 10 definiert haben. Sei o.B.d.A. n ungerade. Wir zeigen, dass diese Funktion $(n-1)/2$-mixed ist. Sei W eine Menge von Variablen mit $|W| \leq (n-1)/2$ und seien σ_1 und σ_2 verschiedene Belegungen der Variablen in W. Die Variablen in W entsprechen Kanten des Graphen. Wenn die Anzahl der Dreiecke, die sich aus den auf 1 gesetzten Variablen in σ_1 bzw. σ_2 ergeben, in einer der Belegungen gerade und in der anderen ungerade ist, sind die Subfunktionen, die durch Konstantsetzen der W-Variablen gemäß σ_1 bzw. σ_2 entstehen, verschieden: wenn wir alle übrigen Variablen auf 0 setzen, ergibt sich einmal der Funktionswert 0 und einmal der Funktionswert 1. Nehmen wir also an, dass die Anzahl der Dreiecke, die aus den Kanten in σ_1 bzw. σ_2 gebildet werden, in beiden Fällen o.B.d.A. gerade ist. Da σ_1 und σ_2 verschieden sind, gibt es eine Variable, die o.B.d.A. in σ_1 den Wert 0 und in σ_2 den Wert 1 hat. Sei $\{u, v\}$ die zugehörige Kante. Da $|W| \leq (n-1)/2$, gibt es einen Knoten w, der in keiner Kante enthalten ist, die durch eine Variable in W codiert ist. Wir belegen nun die Variablen außerhalb von W so, dass genau die beiden Kanten $\{u, w\}$ und $\{v, w\}$ vorhanden sind. Zusammen mit σ_1

ergibt dies einen Graphen, der weiterhin eine gerade Anzahl von Dreiecken hat, während zusammen mit σ_2 das Dreieck $\{u, v, w\}$ hinzugekommen ist, so dass wir eine ungerade Anzahl von Dreiecken haben. Insgesamt folgt die untere Schranke $2^{\Omega(n)} = 2^{\Omega(N^{1/2})}$ für die Funktion $\oplus cl_{n,3}$, die auf $N = \binom{n}{2}$ Variablen definiert ist.

Wir stellen jetzt eine Methode zum Beweis unterer Schranken für nichtdeterministische FBDDs vor.

Definition 19.3: Sei G ein nichtdeterministisches, vollständiges FBDD. Ein *Schnitt* durch G ist eine Menge V von Knoten von G, sodass jeder Pfad von der Quelle zur 1-Senke durch mindestens einen Knoten aus V verläuft.

Lemma 19.4: Sei G ein nichtdeterministisches, vollständiges FBDD für f und sei V ein Schnitt durch G. Dann lässt sich f schreiben als

$$f = \bigvee_{1 \le i \le |V|} \left(f_{i,1}(X_{i,1}) \wedge f_{i,2}(X_{i,2}) \right),$$

wobei für jedes i die Mengen $X_{i,1}$ und $X_{i,2}$ eine Partition der Variablenmenge in zwei Teile bilden und die Funktionen $f_{i,j}$ nur von den Variablen in $X_{i,j}$ abhängen.

Beweis: Sei v_i ein Knoten des Schnitts. Da G vollständig ist, werden auf allen Pfaden von der Quelle zu v_i dieselben Variablen getestet; dasselbe gilt für alle Pfade von v_i zur 1-Senke. Sei $X_{i,1}$ die Menge der Variablen, die auf den Pfaden von Quelle zu v_i getestet werden (ohne die Variable an v_i), und sei $X_{i,2}$ die Menge der Variablen, die auf den Pfaden von v_i zur 1-Senke getestet werden (einschließlich der Variablen an v_i). Sei $f_{i,1}$ die Funktion, die eine 1 berechnet, wenn die Variablen in $X_{i,1}$ so belegt sind, dass der Berechnungspfad von der Quelle zu v_i führt. Sei $f_{i,2}$ die Funktion, die eine 1 berechnet, wenn die Variablen in $X_{i,2}$ so belegt sind, dass der Berechnungspfad von v_i zur 1-Senke führt. Dann nimmt $f_{i,1}(X_{i,1}) \wedge f_{i,2}(X_{i,2})$ den Wert 1 an, wenn es für die Eingabe x einen Berechnungspfad von der Quelle über v_i zur 1-Senke gibt. Offensichtlich ergibt sich f, wenn wir das ODER über alle v_i bilden. □

Was ist nun die Beziehung zu den Argumenten der Kommunikationskomplexität? Die Mengen von Eingaben, für die $f_{i,1}(X_{i,1}) \wedge f_{i,2}(X_{i,2})$ den Wert 1 annimmt, bilden ein kombinatorisches Rechteck. Wir betrachten dazu die Kommunikationsmatrix, wo die Zeilen mit Belegungen aus $X_{i,1}$ markiert sind und die Spalten mit Belegungen aus $X_{i,2}$. Wir bilden nun die Teilmenge X' der Zeilen, wo $f_{i,1}(X_{i,1})$ den Wert 1 berechnet und die Teilmenge Y' der Spalten, wo $f_{i,2}(X_{i,2})$ den Wert 1 berechnet. Die betrachtete Menge von Eingaben ist dann $X' \times Y'$.

Wenn wir nun eine untere Schranke für die Größe von nichtdeterministischen FBDDs zeigen wollen, genügt es zu zeigen, dass jeder Schnitt groß sein muss. Dazu zeigen wir, dass es viele Eingaben gibt, die auf 1 abgebildet werden, dass aber für jede Partition der Variablenmenge in zwei Teile ein Rechteck klein sein muss. Also sind viele Rechtecke für die Überdeckung der 1-Eingaben nötig, und der Schnitt kann nicht kleiner sein als die Anzahl

dieser Rechtecke. Wir beachten aber, dass wir anders als bei OBDDs nicht davon ausgehen können, dass sich alle Rechtecke auf dieselbe Partition der Variablenmenge beziehen, für jedes Rechteck kann es eine andere Partition geben. Daher funktionieren auch die Argumente mit Unterscheidungsmengen hier nicht und es gibt auch keine schöne graphische Darstellung der Rechtecküberdeckung.

Wir beweisen die untere Schranke für den Test auf Permutationsmatrix. Eingabe sind n^2 boolesche Variablen, die in einer quadratischen Matrix angeordnet sind. Die Funktion PERM$_n$ berechnet eine 1, wenn die Matrix eine Permutationsmatrix ist, d.h., wenn es in jeder Zeile und jeder Spalte genau eine 1 gibt. Es gibt $n!$ Eingaben, die von PERM$_n$ auf 1 abgebildet werden. (Für die Position der 1 in der ersten Zeile haben wir n Möglichkeiten, für die Position der 1 in der zweiten Zeile $n-1$, usw., insgesamt also $n!$ Möglichkeiten.)

Sei o.B.d.A. n gerade. Sei nun G ein vollständiges nichtdeterministisches FBDD für PERM$_n$. Wir fixieren für jede Eingabe, auf der PERM$_n$ eine 1 berechnet, einen der möglicherweise mehreren akzeptierenden Berechnungspfade. Auf diesem Berechnungspfad suchen wir den ersten Knoten v_i, sodass zuvor $n/2$ Einsen gelesen wurden, und fügen diesen Knoten zum Schnitt V hinzu. Da die Eingabe akzeptiert wird, werden auch ab v_i genau $n/2$ Einsen gelesen.

Wir zeigen nun, dass jede Funktion $f_{i,1}(X_{i,1})$ auf höchstens $(n/2)!$ Belegungen der Variablen in $X_{i,1}$ den Wert 1 berechnen kann. Seien also σ_1 und σ_2 Belegungen der Variablen in $X_{i,1}$, sodass $f_{i,1}$ eine 1 berechnet. Dann gibt es in beiden Belegungen genau $n/2$ Einsen, die in $n/2$ verschiedenen Zeilen und $n/2$ verschiedenen Spalten der Matrix stehen. Diese Auswahl von $n/2$ Zeilen und $n/2$ Spalten sind für σ_1 und σ_2 gleich. Um dies zu zeigen, betrachten wir eine Belegung ρ mit $f_{i,2}(\rho) = 1$. Diese Belegung muss die Belegung σ_1 zu einer Permutationsmatrix ergänzen, also in allen Zeilen und Spalten, wo σ_1 keine 1 hat, eine 1 haben. Wenn nun σ_2 in einer Zeile eine 1 hätte, wo σ_1 keine hat, wäre die Kombination aus σ_2 und ρ keine Permutationsmatrix, das FBDD würde aber eine 1 berechnen, was ein Widerspruch ist.

Die Anzahl der Belegungen σ mit $f_{i,1}(\sigma) = 1$ ist damit durch die Anzahl der Permutationsmatrizen auf $(n/2 \times n/2)$-Matrizen, also durch $(n/2)!$ beschränkt. Dasselbe gilt für die Anzahl der Belegungen ρ mit $f_{i,2}(\rho) = 1$. Für einen Knoten v_i des Schnitts kann es also höchstens $((n/2)!)^2$ Eingaben geben, sodass der akzeptierende Berechnungspfad über v_i verläuft. Da $n!$ Eingaben zu überdecken sind, muss der Schnitt mindestens

$$\frac{n!}{((n/2)!)^2} = \binom{n}{n/2} = \Omega(2^n/n^{1/2})$$

Knoten enthalten. Da PBDDs ein Spezialfall von nichtdeterministischen FBDDs sind, haben wir die folgende Aussage bewiesen.

Satz 19.5: PBDDs und nichtdeterministische FBDDs für PERM$_n$ haben $\Omega(2^n/n^{1/2})$ Knoten.

Wir merken nur an, dass das Komplement von PERM polynomielle nichtdeterministische OBDDs hat: Es genügt, die Zeile oder Spalte zu raten, die nicht genau eine 1 enthält, und dies zu verifizieren. Wir haben oben erwähnt, dass die Menge der Funktionen, die durch

nichtdeterministische BDDs polynomieller Größe darstellbar sind, gegen Komplementbildung abgeschlossen ist, d.h., $\mathcal{NL} =$ co-$\mathcal{NL}$ (Satz von Immerman und Szelepcsényi, siehe Vorlesung Komplexitätstheorie). Im Fall von nichtdeterministischen FBDDs gilt nicht, das Gegenbeispiel ist PERM.

Zum Abschluss wollen wir noch eine untere Schranke für die Größe von PBDDs für INDEX-EQ zeigen, wenn die Anzahl der Teile des PBDDs beschränkt ist. Wir benötigen hierzu eine Methode, die sich nicht auf FBDDs übertragen lässt, da FBDDs für INDEX-EQ polynomielle Größe haben (siehe Abschnitt 15).

Satz 19.6: PBDDs mit k Teilen für INDEX-EQ$_{n,n/2^{k+1}}$ haben mindestens $2^{\Omega(n/2^k)}$ Knoten.

Beweis: Sei o.B.d.A. n eine Zweierpotenz. Sei ein PBDD G für INDEX-EQ$_{n,n/2^{k+1}}$ mit k Teilen gegeben und sei π_i die Variablenordnung des i-ten Teils.

Wir berechnen nun iterativ Mengen Y_j und Z_j von Datenvariablen, sodass $|Y_j| = |Z_j| = n/2^{j+1}$ und $Y_j \cap Z_j = \emptyset$ und weiterhin für alle $i \in \{1, \ldots, j\}$ gilt, dass bezüglich π_i alle Variablen aus Y_i vor allen Variablen aus Z_i angeordnet sind oder umgekehrt. Zudem sind alle Y_j Teilmengen von $\{x_0, \ldots, x_{n/2-1}\}$ und alle Z_j Teilmengen von $\{x_{n/2}, \ldots, x_{n-1}\}$.

Für Y_0 wählen wir $\{x_0, \ldots, x_{n/2-1}\}$, für Z_0 wählen wir $\{x_{n/2}, \ldots, x_{n-1}\}$. Dann haben Y_0 und Z_0 die geforderten Eigenschaften. Seien nun Y_{j-1} und Z_{j-1} mit den geforderten Eigenschaften konstruiert. Wir erzeugen nun die Mengen Y_j und Z_j. Dazu schreiben wir die Variablen gemäß der Ordnung π_j auf und durchlaufen diese Folge von links nach rechts. Wir stoppen hinter der ersten Variablen, an der mindestens $n/2^{j+1}$ Variablen aus Y_{j-1} oder $n/2^{j+1}$ Variablen aus Z_{j-1} erreicht wurden. Im ersten Fall wählen wir für Y_j die Variablen aus Y_{j-1} bis zur gewählten Position, und für Z_j wählen wir $n/2^{j+1}$ Variablen aus Z_{j-1}, die hinter der gewählten Position stehen. Der zweite Fall ist symmetrisch. Am Ende haben wir zwei disjunkte Variablenmengen Y_k und Z_k, sodass in jedem Teil-OBDD die Y_k-Variablen vor den Z_k-Variablen getestet werden oder umgekehrt. Weiterhin ist $Y_k \subseteq \{x_0, \ldots, x_{n/2-1}\}$ und $Z_k \subseteq \{x_{n/2}, \ldots, x_{n-1}\}$.

Wir setzen nun in G alle Datenvariablen außerhalb von Y_k und Z_k auf 0. Weiterhin wählen wir die Adressvariablen so, dass die Variablen aus Y_k durch die a-Zeiger und die Variablen aus Z_k durch die b-Zeiger adressiert werden. Dies wird so gemacht, dass die Ordnungsbedingung aus der Definition von INDEX-EQ erfüllt ist. Letzteres ist möglich, da nach Konstruktion die Variablen aus Y_k kleinere Indizes als die Variablen aus Z_k haben. Wir erhalten auf diese Weise ein PBDD G' für die Gleichheitsfunktion der Variablen in Y_k und Z_k, wobei in jedem Teil alle Variablen aus Y_k vor oder nach den Variablen aus Z_k getestet werden.

Wir konstruieren ein nichtdeterministisches Kommunikationsprotokoll für EQ$_s(a, b)$ für $s = n/2^{k+1}$. Seien also $a, b \in \{0,1\}^s$ gegeben. Alice rät, in welchem der k Teile von G' die vorliegende Eingabe akzeptiert wird. Falls in dem Teil die Y_k- vor den Z_k-Variablen getestet werden, simuliert sie die Rechnung auf den Y_k-Variablen mit ihrer Eingabe a und sendet die Nummer des erreichten Knotens an Bob. Bob simuliert die Rechnung mit seiner Eingabe, wobei er an dem Knoten startet, den er von Alice erhalten hat. Bob akzeptiert genau dann, wenn diese Rechnung an einer 1-Senke endet.

Falls in dem von Alice geratenen Teil die Z_k-Variablen vor den Y_k-Variablen getestet werden, rät Alice weiterhin den Knoten v, an dem Bob seine Rechnung auf den Z_k-Variablen beendet. Dann simuliert sie ihre Rechnung beginnend an v. Wenn diese Rechnung an der 1-Senke endet, sendet sie die Nummer von v an Bob. Bob simuliert dann das Teil-OBDD, das den Knoten v enthält, für seine Eingabe und akzeptiert, wenn seine Rechnung an v endet. Da das von Alice geratene Teil-OBDD über die Nummer des gesendeten Knotens bestimmt ist, kann Bob ohne weitere Kommunikation die beiden Fälle unterscheiden.

Insgesamt erhalten wir ein nichtdeterministisches Kommunikationsprotokoll für $\mathrm{EQ}_s(a, b)$ für $s = n/2^{k+1}$, wobei die Anzahl der gesendeten Bits $O(\log |G|)$ ist. Da die nichtdeterministische Kommunikationskomplexität $\Omega(n/2^k)$ ist, folgt die untere Schranke $2^{\Omega(n/2^k)}$ für die Größe von G. □

Wir merken noch an, dass wir im Beweis nicht ausgenutzt haben, dass die einzelnen Teile des PBDDs deterministisch sind, d.h., derselbe Beweis funktioniert auch, wenn das PBDD aus nichtdeterministischen OBDDs besteht. Die entscheidende Einschränkung gegenüber über den FBDDs besteht in diesem Beweis also darin, dass die Anzahl der verschiedenen Variablenordnungen beschränkt ist, was bei FBDDs nicht der Fall ist.

20 Implizite Graphdarstellungen und Flussmaximierung

In diesem Abschnitt wollen wir eine Graphdarstellung mit Hilfe von OBDDs kennenlernen und die Komplexität von Graphproblemen auf dieser neuen Darstellung untersuchen. Graphen werden üblicherweise als Adjazenzmatrix oder Adjazenzlisten dargestellt. Der Nachteil beider Darstellungen besteht darin, dass ihr Speicherbedarf mindestens linear in der Anzahl der Knoten und der Kanten wächst. D.h., dass z.B. Graphen mit 10^{20} Knoten bei den heute üblichen Speicherkapazitäten prinzipiell nicht gespeichert werden können. Um dieses Problem zu umgehen, kann man kompaktere Darstellungen von Graphen definieren. Wir wollen schon vorab erwähnen, dass solche Darstellungen nicht generell kompakter sein können; da es $2^{\binom{n}{2}}$ verschiedene ungerichtete Graphen mit n Knoten gibt, muss es für jede Graphdarstellung Graphen mit mindestens $\binom{n}{2}$ Bits geben. Dies ist analog zu der Aussage, dass es bei jeder Darstellungsform für boolesche Funktionen auch Funktionen auf n Variablen geben muss, die 2^n Bits benötigen. Allerdings kann es Darstellungen geben, die für viele wichtige Funktionen oder Graphen kompakter sind.

Die Idee der betrachteten Darstellung besteht darin, die Kantenrelation des Graphen mit Hilfe von OBDDs darzustellen.

Definition 20.1: Sei $G = (V, E)$ ein Graph mit $V = \{1, \ldots, n\}$. Sei $k = \lceil \log |V| \rceil$ und sei f_G auf den Variablen $x_{k-1}, \ldots, x_0, y_{k-1}, \ldots, y_0$ definiert durch $f_G(x, y) = 1 \Leftrightarrow \{x, y\} \in E$ (bzw. $(x, y) \in E$ für den Fall von gerichteten Graphen). Die Funktion f_G ist also die charakteristische Funktion der Kantenrelation. Die Darstellung von f_G durch ein OBDD heißt implizite Darstellung von G.

Als einfaches Beispiel betrachten wir einen Graphen $G = (V, E)$, der eine Linie ist. Sei $V = \{1, \ldots, n\}$ mit $n = 2^k$ und $E = \{\{i, i + 1\} \mid 1 \leq i \leq n - 1\}$. Dann ist $f_G(x, y)$ die Funktion, die testet, ob sich x und y als Binärzahl interpretiert um genau 1 unterscheiden. Es ist leicht, OBDDs mit linearer Größe für diese Funktion anzugeben. Hierfür ist allerdings entscheidend, dass die einzelnen Bits von x und y abwechselnd getestet werden, die Variablenordnung also $x_{k-1}, y_{k-1}, \ldots, x_0, y_0$ (oder umgekehrt) ist. Nur so kann man sicherstellen, dass zusammengehörende Bits auch zusammen getestet werden. Diese Variablenordnungen werden als *interleaved* bezeichnet.

Es stellt sich nun die Frage, wie Algorithmen für die bekannten Graphprobleme modifiziert werden können, um Algorithmen zu erhalten, die auf der impliziten Darstellung arbeiten. Es ist sicherlich nur in Ausnahmefällen sinnvoll, die Algorithmen unverändert zu übernehmen und für jede Kante, die der Algorithmus abfragt, das OBDD auszuwerten, da man dann zwar eine kompaktere Darstellung hat, aber keinen Vorteil gegenüber der konventionellen Darstellung in Bezug auf die Rechenzeit. Analog würden OBDDs für die Darstellung von booleschen Funktionen kaum Vorteile bringen, wenn wir sie nur auswerten dürften und keine anderen Operationen anwenden dürften. Also brauchen wir auch für die impliziten Graphdarstellungen neue Algorithmen.

Wir beginnen mit einfachen positiven Beispielen. Die Anzahl erfüllender Belegungen von f_G ist bei gerichteten Graphen gleich der Anzahl der Kanten, bei ungerichteten Graphen gleich

zweimal der Anzahl der Kanten, da für jede Kante zwischen x und y gilt, dass $f_G(x,y) = f_G(y,x) = 1$. Also kann die Anzahl der Kanten leicht mit dem Algorithmus für Erfüllbarkeit-Anzahl bestimmt werden. Ein weiteres Beispiel ist der Test, ob der gegebene Graph ein Dreieck enthält. Wir benutzen die Variablenordnung $x_{k-1}, y_{k-1}, z_{k-1}, \ldots, x_0, y_0, z_0$, d.h., wir haben jetzt Variablen, die insgesamt drei Knoten x, y und z codieren. Es ist leicht, aus dem gegebenen OBDD für $f_G(x,y)$ OBDDs für $f_G(x,z)$ und $f_G(y,z)$ zu konstruieren, da es genügt, die Variablenmarkierungen an den inneren Knoten anzupassen. Mit Synthese berechnen wir jetzt ein OBDD für $f_G(x,y) \wedge f_G(x,z) \wedge f_G(y,z)$. Dieses berechnet auf dem Tripel (x,y,z) von Knoten genau dann eine 1, wenn der Graph auf x, y und z ein Dreieck enthält. Der Test auf die Existenz eines Dreiecks ist also ein Erfüllbarkeitstest. Da es für jedes Dreieck genau 6 erfüllende Belegungen gibt, erhalten wir die Anzahl der Dreiecke mit dem Algorithmus für Erfüllbarkeit-Anzahl und anschließender Division durch 6.

Leider sind aber die meisten Operationen „schwierig“. Zunächst fällt auf, dass sich viele Grapheigenschaften mit Quantoren beschreiben lassen. Beispiele: „Ist der Graph 3-regulär?“ lässt sich formulieren als „Haben alle Knoten den Grad 3?“ oder etwas genauer: „Gibt es für alle Knoten v genau drei verschiedene Knoten w_1, w_2, w_3, die Nachbarn von v sind?“ Oder: „Gibt es einen isolierten Knoten?“ lässt sich noch genauer formulieren als „Gibt es einen Knoten v, sodass für alle Knoten w gilt, dass v und w nicht verbunden sind?“ Eine kleine implizite Darstellung hat polynomielle Größe in $k = \lceil \log n \rceil$. Wenn wir nun über k Variablen quantifizieren, die einen Knoten beschreiben, kann dies zu einer exponentiellen Vergrößerung der Darstellung führen. Dass die Auswertung von Ausdrücken mit Quantoren nicht zwangsläufig zu einer exponentiellen Rechenzeit führen muss, zeigt das Beispiel aus dem letzten Absatz, wo wir einen polynomiellen Algorithmus für den Test auf Dreiecke angegeben haben (der sich formulieren lässt als „Gibt es drei Knoten x, y und z, die paarweise durch Kanten verbunden sind?“). Also müssen wir die Härte von Operationen beweisen. Dies machen wir für das sogenannte Graph Accessibility Problem (GAP), das für einen gerichteten Graphen $G = (V,E)$ und zwei Knoten s und t testet, ob es einen gerichteten Weg von s nach t gibt. Dieses Problem wird sich als PSPACE-vollständig erweisen. Dabei ist PSPACE die Menge aller Entscheidungsprobleme mit polynomiell platzbeschränkten Algorithmen. Insbesondere enthält PSPACE die Klasse NP; die PSPACE-vollständigen Probleme sind auch NP-schwer und gelten als „noch schwerer“ als die NP-vollständigen Probleme. Der Begriff PSPACE-vollständig ist über polynomielle Reduktionen ($\leq_p$) definiert, d.h., ein Problem Π ist PSPACE-vollständig, wenn es in PSPACE enthalten ist und alle Probleme aus PSPACE polynomiell auf Π reduziert werden können.

Satz 20.2: Für implizit dargestellte Graphen ist GAP PSPACE-vollständig.

Beweis: Zunächst ist leicht einzusehen, dass man GAP nichtdeterministisch mit polynomiellem Platz lösen kann; es genügt beginnend mit s jeweils den nächsten Knoten auf dem s-t-Pfad zu raten und anschließend durch Auswertung des OBDDs für f_G zu verifizieren, dass die Kante dorthin vorhanden ist. Damit ist gezeigt, dass GAP $\in$ NPSPACE (Nondeterministic PSPACE). Der Satz von Savitch aus der Vorlesung Komplexitätstheorie impliziert, dass PSPACE $=$ NPSPACE. Damit folgt GAP $\in$ PSPACE.

Bei der nun folgenden Reduktion gehen wir ähnlich wie beim Beweis des Satzes von Cook vor. Wir starten mit einem beliebigen Problem L aus PSPACE. Über L wissen wir nur, dass es eine polynomiell platzbeschränkte deterministische Turingmaschine M für L gibt. Wir zeigen dann, wie wir die Berechnungen von M durch eine implizite Graphdarstellung codieren können. Sei x eine Eingabe. Die Reduktion berechnet aus x einen implizit dargestellten Graphen sowie zwei Knoten s und t, sodass $x \in L$ genau dann gilt, wenn es einen Weg von s nach t gibt.

Sei $p(|x|)$ die Platzschranke von M, d.h., M arbeitet nur auf den $2p(|x|) + 1$ Bandzellen $-p(|x|), \ldots, p(|x|)$. O.B.d.A. habe M nur eine akzeptierende Konfiguration; dies können wir z.B. erreichen, indem M vor Erreichen des akzeptierenden Endzustandes das Band löscht und den Kopf auf Position 0 fährt. Eine Konfiguration von M beschreiben wir durch Angabe der Bandinschrift, wobei wir links neben der Kopfposition zusätzlich den Zustand einfügen. Da das Bandalphabet und die Zustandsmenge konstante Größe haben, kann eine Konfiguration durch einen String der Länge $c(2p(|x|)+2)$ beschrieben werden, wobei c eine geeignete Konstante ist. Wir definieren nun den Konfigurationsgraphen von M für die Eingabelänge n. Als Knoten enthält dieser Graph alle möglichen 0-1-Strings der Länge $c(2p(n) + 2)$. Es gibt eine Kante vom Knoten u zum Knoten v, wenn M in einem Schritt von der Konfiguration u zur Konfiguration v kommt. Wir zeigen gleich, dass diese Relation OBDDs polynomieller Größe hat, die auch leicht in polynomieller Zeit berechnet werden können. Sei nun $s(x)$ die Startkonfiguration für die gegebene Eingabe x und t die akzeptierende Endkonfiguration von M. Dann gilt offensichtlich, dass es in dem Konfigurationsgraphen genau dann einen Pfad von $s(x)$ nach t gibt, wenn M die Eingabe x akzeptiert. Also haben wir eine Reduktion von L auf GAP angegeben.

Es bleibt noch zu zeigen, dass der Konfigurationsgraph eine implizite Darstellung polynomieller Größe hat, die in polynomieller Zeit berechnet werden kann. Sei Q die Zustandsmenge von M und Γ das Bandalphabet. Dann wählen wir $c = \lceil \log(|Q| + |\Gamma|) \rceil$. Eine Konfiguration wird nun, wie oben erklärt, durch einen 0-1-String der Länge $l = c(2p(|x|)+2)$ beschrieben. Seien nun $y = (y_0, \ldots, y_{l-1})$ und $z = (z_0, \ldots, z_{l-1})$ zwei solche Strings. Um zu testen, ob (y, z) ein Kante im Konfigurationsgraphen ist, müssen wir in einem OBDD simultan testen, ob y und z gültige Beschreibungen von Konfigurationen sind und ob M in einem Schritt von y nach z kommt. Die Variablenordnung ist interleaved und gemäß der Anordnung der Buchstaben in der Beschreibung der Konfiguration. Wir beachten weiterhin, dass $|Q|$ und $|\Gamma|$ und damit auch c Konstanten sind. Um zu testen, ob y eine Konfiguration beschreibt, genügt es zu testen, ob y genau einen Buchstaben aus Q enthält und alle übrigen Buchstaben Buchstaben aus Γ sind. Für die betrachtete Variablenordnung ist dies einfach möglich. Dasselbe gilt für den analogen Test für z. Die OBDDs für diese Tests nennen wir G_y bzw. G_z. Beide haben die Größe $O(p(|x|))$.

Um zu testen, ob z eine Nachfolgekonfiguration von y codiert, genügen die folgenden Tests. Seien p_y und p_z die Positionen des Buchstabens aus Q in y bzw. z, also die Kopfpositionen in den betrachteten Konfigurationen.

1. p_y und p_z unterscheiden sich um höchstens 1.

2. y und z stimmen in allen Bits überein, die den Bandinhalt bis zur Position $\min\{p_y, p_z\} - 1$ beschreiben, sowie den Bits, die den Bandinhalt ab Position $\max\{p_y, p_z\} + 2$ beschreiben.

3. Die höchstens $3c$ Bits, die den Bandinhalt zwischen den Positionen $\min\{p_y, p_z\}, \ldots, \max\{p_y, p_z\} + 1$ beschreiben, codieren einen korrekten Übergang gemäß der Übergangsfunktion von M.

Also werden jeweils bezüglich der Variablenordnung benachbarte Bits auf Gleichheit getestet. Dies geht mit OBDDs in linearer Größe bezüglich $p(|x|)$. Die einzige Ausnahme ist eine Umgebung konstanter Größe um die Kopfposition. Hier geht die Übergangsfunktion von M ein. Da hier eine Funktion auf konstant vielen Bits berechnet wird, trägt dies nur einen konstanten Summanden zur OBDD-Größe bei. Das resultierende OBDD, das testet, ob z Nachfolgekonfiguration von y ist, nennen wir G'. Das OBDD für die implizite Darstellung des Konfigurationsgraphen erhalten wir dann mit der $\wedge$-Synthese von G_y, G_z und G'. □

Das Fazit ist, dass das Graphproblem GAP, das für konventionelle Graphdarstellungen in P (und sogar in NL (nondeterministic logspace)) liegt, für implizite Graphdarstellungen PSPACE-vollständig ist. Sind damit implizite Graphdarstellungen prinzipiell unbrauchbar? Wir sollten die Situation mit der Darstellung von booleschen Funktionen vergleichen. Auf Wertetabellen sind viele Operationen in polynomieller Zeit möglich, wobei sich das Polynom aus der Rechenzeit auf die (aufgeblähte) Eingabelänge der Wertetabellen bezieht. Diese Ergebnisse sind damit relativ wertlos. Sobald wir zu kompakteren Darstellungen übergehen, werden wichtige Operationen NP-schwer, z.B. der Erfüllbarkeitstest für Schaltkreise oder das Minimierungsproblem für OBDDs. Eine Faustregel ist: Je kompakter die Darstellung, desto schwieriger werden die Probleme auf diesen Darstellungen. Dasselbe gilt für die impliziten Graphdarstellungen. Einerseits bieten sie die Möglichkeit manche Graphen darzustellen, die wir konventionell gar nicht speichern können. Andererseits werden die Operationen schwierig, sodass wir nur auf Heuristiken hoffen können und nur erwarten können, in manchen Fällen zum Erfolg zu kommen, in denen konventionelle Ansätze garantiert nicht zum Erfolg kommen. Daher ist es auch nicht sinnvoll, die Worst-Case-Rechenzeiten der Algorithmen auf impliziten Graphdarstellungen mit den konventionellen Algorithmen zu vergleichen; wir haben ja auch nicht die Effizienz der OBDD-Algorithmen mit der Effizienz von Algorithmen auf Wertetabellen verglichen.

Wir wollen nun einen Algorithmus für die Flussmaximierung konstruieren, wobei der gegebene Graph implizit gegeben ist. Das Problem der Flussmaximierung wird ausführlich in der Vorlesung Effiziente Algorithmen behandelt. Wir können hier nur die für das Verständnis des Algorithmus benötigten Grundlagen ohne Beweise wiederholen.

Definition 20.3: Ein Netzwerk $G = (V, E)$ ist ein gerichteter, asymmetrischer Graph mit einer Quelle s ohne eingehende Kanten und einer Senke t ohne ausgehende Kanten.

Ein 0-1-Fluss ist eine Funktion $\varphi : E \to \{0, 1\}$, für die die Kirchhoff-Regel gilt:

$$\forall v \in V : \sum_{(u,v)\in E} \varphi(u, v) = \sum_{(v,w)\in E} \varphi(v, w),$$

d.h., dass in jeden Knoten genausoviel hereinfließt, wie herausfließt. Der Wert $w(\varphi)$ des Flusses φ ist

$$w(\varphi) = \sum_{(s,w)\in E} \varphi(s,w),$$

also der Gesamtfluss, der aus der Quelle herausfließt. Die Kirchhoff-Regel impliziert, dass dies gleich dem Gesamtfluss in die Senke ist. Das Problem der Flussmaximierung besteht darin, zu einem gegebenen Netzwerk einen Fluss mit maximalem Wert zu berechnen.

Asymmetrie bedeutet, dass es die Kanten (u, v) und (v, u) nicht zugleich geben darf. Die Forderung nach Asymmetrie wird üblicherweise zur Vereinfachung gemacht. Gegenüber der Vorlesung Effiziente Algorithmen haben wir hier noch einige weitere Vereinfachungen gemacht: Wir betrachten nur Flüsse, die auf den einzelnen Kanten den Wert 0 oder 1 haben dürfen (gegenüber beliebigen reellen Zahlen im allgemeinen Fall). Jede vorhandene Kante kann einen Fluss von 1 aufnehmen (gegenüber der Angabe einer Kapazität für die Kanten im allgemeinen Fall). In der Vorlesung Effiziente Algorithmen werden für den allgemeinen Fall polynomielle Algorithmen angegeben. Wir wollen anmerken, dass auch das hier betrachtete Flussproblem für implizite Graphdarstellungen PSPACE-schwer ist, da sich GAP leicht auf das Flussproblem reduzieren lässt: Es gibt in G genau dann einen Weg von s nach t, wenn G als Flussnetzwerk mit Quelle s und Senke t einen zulässigen Fluss mit Wert mindestens 1 hat.

Die bekannten Algorithmen zur Flussmaximierung basieren auf flussvergrößernden Pfaden. Sei G ein Netzwerk und φ ein Fluss auf G. Ein flussvergrößernder Pfad ist eine Folge von Knoten $s = v_1, \ldots, v_l = t$, sodass für jedes $i \in \{1, \ldots, l-1\}$ eine der beiden folgenden Aussagen gilt:

1. Es gibt die Kante $(v_i, v_{i+1}) \in E$ und $\varphi(v_i, v_{i+1}) = 0$. Die Kante (v_i, v_{i+1}) heißt dann Vorwärtskante.

2. Es gibt die Kante $(v_{i+1}, v_i) \in E$ und $\varphi(v_{i+1}, v_i) = 1$. Die Kante (v_{i+1}, v_i) heißt dann Rückwärtskante.

Wenn wir zu G und φ einen flussvergrößernden Pfad haben, können wir den Fluss φ um 1 vergrößern, indem wir für alle Vorwärtskanten den Fluss von 0 auf 1 vergrößern und für alle Rückwärtskanten den Fluss von 1 auf 0 verringern. Man überlegt sich leicht, dass die Kirchhoffregel weiterhin gilt und der Wert des neuen Flusses um 1 größer als der Wert von φ ist. Es ist bekannt, dass ein Fluss maximal ist, wenn es keinen flussvergrößernden Pfad mehr gibt (Beweis in der Vorlesung Effiziente Algorithmen).

Bei der Analyse der Algorithmen zur Flussmaximierung stellt sich heraus, dass es günstig ist, nach kürzesten flussvergrößernden Pfaden zu suchen und möglichst alle kürzesten flussvergrößernden Pfade zugleich abzuarbeiten. Es kann bewiesen werden, dass dann die Länge eines kürzesten flussvergrößernden Pfades um mindestens 1 wächst, so dass man obere Schranken für die Rechenzeit bekommt. Eine Datenstruktur, die es erlaubt alle flussvergrößernden Pfade zugleich darzustellen und zu behandeln, ist das Niveaunetzwerk zu G und φ.

Definition 20.4: Sei G ein Netzwerk und φ ein Fluss auf G. Sei V_i die Menge aller Knoten u, die auf einem flussvergrößernden Pfad von s nach t liegen, wobei die Anzahl der Kanten auf einem kürzesten flussvergrößernden Pfad von s nach u genau i ist. Sei

$$\begin{aligned} E_i &= \{(v,w)|v \in V_i, w \in V_{i+1}, (v,w) \in E, \varphi(v,w) = 0\} \cup \\ &\quad \{(v,w)|v \in V_i, w \in V_{i+1}, (w,v) \in E, \varphi(w,v) = 1\}, \end{aligned}$$

also die Menge der Kanten zwischen den Niveaus V_i und V_{i+1}, die auf flussvergrößernden Pfaden liegen können, wobei die Rückwärtskanten umgedreht sind. Sei V_φ die Vereinigung aller V_i, und sei E_φ die Vereinigung aller E_i. Dann heißt $G_\varphi = (V_\varphi, E_\varphi)$ Niveaunetzwerk zu G und φ.

Ein Niveaunetzwerk enthält also alle kürzesten flussvergrößernden Pfade, wobei die Rückwärtskanten umgedreht sind und damit nicht anders als die Vorwärtskanten behandelt werden müssen. (Wegen der Asymmetrie entstehen hierdurch keine Mehrfachkanten.) Wir beachten, dass flussvergrößernde Pfade nicht notwendigerweise disjunkt sind. Um alle kürzesten flussvergrößernden Pfade zu zerstören, genügt es, einen sogenannten Sperrfluss auf G_φ zu berechnen. Dies ist ein Fluss auf G_φ, sodass es in G_φ keinen s-t-Weg mehr gibt, auf dem nicht mindestens eine Kante den Fluss 1 bekommen hat. Solche Kanten werden als saturiert bezeichnet. Manche der bekannten Flussalgorithmen arbeiten nun folgendermaßen. Sie starten mit dem Nullfluss, der sicherlich ein zulässiger Fluss ist. Sie berechnen das zugehörige Niveaunetzwerk und einen Sperrfluss auf dem Niveaunetzwerk. Der Sperrfluss wird dann kantenweise zu dem Fluss auf G addiert (bzw. für Rückwärtskanten subtrahiert). Auf diese Weise erhält man einen größeren Fluss und alle kürzesten flussvergrößernden Pfade sind zerstört. Dieses Verfahren wird iteriert, bis die Senke nicht mehr über einen flussvergrößerenden Pfad erreichbar ist.

Beispiele für solche Algorithmen sind der Algorithmus von Karzanov und der Algorithmus von Malhotra, Pramodh Kumar und Maheshwari. Beide Algorithmen unterscheiden sich nur in der Sperrflussberechnung. Sie machen sich zunutze, dass man Sperrflüsse leichter berechnen kann als maximale Flüsse: Zum einen ist ein Sperrfluss nicht notwendigerweise ein maximaler Fluss. Weiterhin sind die Niveaunetzwerke, auf denen ja die Sperrflüsse berechnet werden, geschichtet und damit insbesondere azyklisch.

Ein Beispiel für eine Flussmaximierung mit Hilfe von Sperrflussberechnungen befindet in Abbildung 29 am Ende dieses Abschnitts. Das erste Niveaunetzwerk hat die Knotenmengen $V_0 = \{s\}$, $V_1 = \{a, b\}$, $V_2 = \{f, g\}$ und $V_3 = \{t\}$. Der auf dem Niveaunetzwerk berechnete Fluss ist ein Sperrfluss, der offensichtlich nicht maximal ist, da er ungeschickterweise die Kante (b, f) benutzt. Der Wert des Flusses nach der ersten Flussvergrößerung beträgt damit nur 1. Anschließend ist die Kante (b, f) (wie auch (s, b) und (f, t)) Rückwärtskante. Im zweiten Niveaunetzwerk haben alle flussvergrößernden Pfade jeweils fünf Kanten. Der Knoten b gehört nun zu V_3. Wieder wird ein Sperrfluss berechnet. Da im Niveaunetzwerk der Fluss 1 über die Kante (f, b) fließt und beim vorher berechneten Fluss über die Kante (b, f) ebenfalls Fluss 1 fließt, hat die Kante (b, f) am Ende keinen Fluss mehr. Man sieht leicht, dass der am Ende berechnete Fluss maximal ist.

Wir wollen im Folgenden einen Algorithmus zur Flussmaximierung für implizit dargestellte Flussnetzwerke entwickeln, d.h., es gibt ein OBDD, das die Funktion $E(x, y)$ darstellt,

wobei genau dann $E(x,y) = 1$ ist, wenn es eine Kante von x nach y gibt. Zur Vereinfachung der Notation identifizieren wir hier wie auch im Folgenden Mengen (wie E) mit ihren charakteristischen Funktionen (hier $E(x,y)$ genannt). Wir gehen davon aus, dass die Variablenordnung interleaved ist und dass alle vorkommenden OBDDs dieselbe Variablenordnung haben. Weiterhin gibt es OBDDs für die Funktionen $s(x)$ und $t(x)$, wobei genau dann $s(x) = 1$ ist, wenn x gleich der Quelle s ist, und $t(x)$ analog für die Senke t definiert ist. Der berechnete Fluss muss dann auch durch ein OBDD dargestellt werden, d.h., es gibt ein OBDD für eine Funktion $\varphi(x,y)$, die den Wert 1 genau dann annimmt, wenn der Fluss 1 über die Kante (x,y) fließt.

Der Algorithmus stammt von Hachtel und Somenzi. Er folgt der Idee, zunächst ein Niveaunetzwerk zu berechnen und darauf einen Sperrfluss zu berechnen. Dieses Verfahren wird, wie oben beschrieben, iteriert. Wir können nur hoffen, dass der Algorithmus in interessanten Fällen effizient ist; eine obere Schranke für die Rechenzeit ist nicht bekannt, da der Algorithmus viele Schritte enthält, deren Rechenzeit wir nicht auf einfache Weise hinreichend gut abschätzen können. Wie diskutiert ist es aber auch nicht das Ziel, die konventionellen Algorithmen im worst-case zu schlagen.

Wir beginnen mit der Berechnung eines Niveaunetzwerkes. Sei $\varphi(x,y)$ die boolesche Funktion, die den bereits berechneten Fluss angibt. Zunächst berechnen wir eine Funktion $A(x,y)$, die angibt, ob es eine Kante von x nach y auf einem flussvergrößernden Weg geben kann, also ob (x,y) vorhanden ist und keinen Fluss hat (damit Vorwärtskante sein kann) oder ob (y,x) vorhanden ist und Fluss hat (damit Rückwärtskante sein kann):

$$A(x,y) = E(x,y)\overline{\varphi(x,y)} \vee E(y,x)\varphi(y,x).$$

Also kann ein OBDD für $A(x,y)$ mit OBDD-Operationen berechnet werden. Wir beachten aber, dass eine Veränderung der Variablenordnung nötig ist, um aus einem OBDD für $E(x,y)$ ein OBDD für $E(y,x)$ zu berechnen. Da die Variablenordnung interleaved ist, werden bei dieser Änderung nur Paare von jeweils benachbarten Variablen vertauscht. Da jeweils disjunkte Bereiche des OBDDs von den einzelnen Vertauschungen betroffen sind, kann sich dabei die OBDD-Größe höchstens verdoppeln. Die Einzelheiten zur Berechnung der OBDDs gelten analog für die im Folgenden berechneten OBDDs und werden dort nicht mehr extra erwähnt.

Wir wollen nun induktiv die Menge V_i der Knoten bestimmen, die über i Kanten aus $A(x,y)$ von der Quelle s aus erreichbar sind. Zunächst ist $V_0(x) = s(x)$. Sei

$$R_m(x) = V_0(x) \vee \cdots \vee V_m(x),$$

also die Menge der Knoten auf den ersten m Schichten. Die Vorwärtskanten von Schicht m zu Schicht $m+1$ sind dann

$$F_m(x,y) = V_m(x) \wedge E(x,y) \wedge \overline{\varphi(x,y)} \wedge \overline{R_m(y)},$$

d.h., (x,y) ist Vorwärtskante von Schicht m zur Schicht $m+1$, wenn x in Schicht m liegt, die Kante (x,y) im Netzwerk vorhanden ist, bisher noch keinen Fluss hat und der Zielknoten y nicht in einer früheren Schicht liegt. Die Leserin und der Leser sollten auch die folgenden

Formeln in ähnlicher Weise lesen. Die Rückwärtskanten von Schicht m zu Schicht $m+1$ ergeben sich analog durch

$$\begin{aligned} B_m(x,y) &= V_m(x) \wedge E(y,x) \wedge \varphi(y,x) \wedge \overline{R_m(y)} \\ &= V_m(x) \wedge \varphi(y,x) \wedge \overline{R_m(y)}. \end{aligned}$$

Die Gleichheit gilt, da nur vorhandene Kanten Fluss haben können. Alle Kanten zwischen den Schichten m und $m+1$ ergeben sich dann durch

$$U_m(x,y) = F_m(x,y) \vee B_m(x,y).$$

Die Knotenmenge für Schicht $m+1$ erhalten wir dann durch

$$V_{m+1}(x) = \exists y\colon U_m(y,x).$$

Dieses Verfahren wir so lange iteriert, bis die Senke t erreicht wird, also bis $V_{m+1}(x) \wedge t(x) \neq 0$ ist, oder bis feststeht, dass die Senke nicht mehr erreichbar ist. Letzteres ist der Fall, wenn keine neue Knoten mehr zum Niveaunetzwerk hinzugefügt werden können, also wenn $V_{m+1} = 0$ gilt. Dann ist der Fluss maximal und der Algorithmus stoppt.

Wir beachten, dass die berechneten Mengen V_i möglicherweise mehr Knoten als die i-te Schicht im Niveaunetzwerk enthalten, da nicht sichergestellt wurde, dass von allen gefundenen Knoten aus die Senke erreichbar ist. Also müssen in einer zweiten Phase durch eine Art Rückwärtssuche die Knoten gesucht werden, von denen aus t erreichbar ist. Dies verläuft völlig analog zu der beschriebenen Vorwärtssuche von s aus, sodass wir dies nicht extra ausführen.

Auf diese Weise erhalten wir das Niveaunetzwerk. Die Knotenmengen nennen wir weiterhin V_m, die Menge der Vorwärtskanten und Rückwärtskanten, die an Knoten in V_m beginnen, nennen wir weiterhin F_m bzw. B_m. Ebenso sei $U_m = F_m \cup B_m$. Die nächste Aufgabe ist die Berechnung eines Sperrflusses auf diesem Niveaunetzwerk.

Die Idee der Sperrflussberechnung ist die Folgende: Es werden induktiv Kanten $S_m \subseteq U_m$ auf eine Weise ausgewählt, dass ein Fluss von 1 auf jeder Kante aus S_m über Kanten aus $S_{m+1}, S_{m+2}, \ldots$ zur Senke weitergeleitet werden kann. Ein Fluss, der nur ausgewählte Kanten benutzt, wird anschließend berechnet und die Kanten mit Fluss werden aus dem Niveaunetzwerk entfernt. Dieses Verfahren wir iteriert, bis es keinen s-t-Pfad im Niveaunetzwerk mehr gibt. Damit ist dann ein Sperrfluss berechnet.

Um im folgenden Algorithmus Symmetrien aufzulösen, benutzen wir eine sogenannte Prioritätsfunktion $\pi(x,y,z)$. Zu jedem Knoten x wird eine Ordnung $<_x$ auf allen anderen Knoten definiert, und es ist genau dann $\pi(x,y,z) = 1$, wenn $y <_x z$ gilt. Wir wollen die Wahl der Prioritätsfunktion hier nicht genauer diskutieren; wir merken nur an, dass sie kleine OBDDs haben sollte. Die Wahl der Prioritätsfunktion beeinflusst die Effizienz des Algorithmus, wie wir noch an einem Beispiel sehen werden; die Korrektheit des Algorithmus ist allerdings unabhängig von der gewählten Prioritätsfunktion.

Sei l die Schicht, die die Senke t enthält, also ist $V_l = \{t\}$. Offensichlich können wir alle in die Senke eingehenden Kanten auswählen, da Fluss auf diesen Kanten zur Senke weitergeleitet werden kann. Also

$$S_{l-1} = U_{l-1}.$$

Wir betrachten nun die Situation, dass wir die Kanten in S_m bereits bestimmt haben und die Kanten in S_{m-1} berechnen wollen. (Ein Beispiel zu dieser Situation findet sich weiter unten.) Zunächst bestimmen wir Tripel (x, y, z) von Knoten mit $x \in V_{m-1}, y \in V_m, z \in V_{m+1}$, sodass es die Kante (x, y) in U_{m-1} und die ausgewählte Kante (y, z) in S_m gibt.

$$P_m(x, y, z) = U_{m-1}(x, y) \wedge S_m(y, z).$$

Da uns der mittlere Knoten auf diesen Pfaden der Länge zwei im Moment nicht interessiert, berechnen wir

$$P_m^*(x, z) = \exists y\colon P_m(x, y, z).$$

Das Ziel des Algorithmus besteht nun darin, auf dem bipartiten Graphen mit der Kantenmenge P_m^* ein (möglichst großes) Matching zu berechnen. Wenn die Kante (x, z) in diesem Matching enthalten ist, bedeutet dies, dass ein Fluss von 1 von x nach z geleitet werden kann. Das Matching wird berechnet, indem zunächst sichergestellt wird, dass es zu jedem x nur einen Partner z mit $P_m^*(x, z)$ gibt, nämlich den ersten Knoten bezüglich der Ordnung $<_x$. Die resultierende Kantenmenge heißt Q_m.

$$Q_m(x, z) = P_m^*(x, z) \wedge [\neg\exists z'\colon (P_m^*(x, z') \wedge \pi(x, z', z))].$$

Anschließend wird auf dieselbe Weise sichergestellt, dass auch jedes z nur einen Partner x hat. Die resultierende Kantenmenge heißt D_m.

$$D_m(x, z) = Q_m(x, z) \wedge [\neg\exists x'\colon (Q_m(x', z) \wedge \pi(z, x', x))].$$

Wir erhalten nun kantendisjunkte Pfade der Länge 2 zwischen den Knoten in V_{m-1} und V_{m+1}, sodass die erste Kante im Niveaunetzwerk vorhanden ist und die zweite Kante ausgewählt ist, indem wir

$$T_m(x, y, z) = D_m(x, z) \wedge P_m(x, y, z)$$

berechnen. Die jeweils ersten Kanten auf diesen Pfaden sollen gewählt werden, also sei

$$S_{m-1}(x, y) = \exists z\colon T_m(x, y, z).$$

Auf diese Weise erhalten wir die Mengen $S_0, \ldots, S_{l-1}$ von ausgewählten Kanten. In einer zweiten Phase schicken wir einen möglichst großen Fluss über die ausgewählten Kanten. Wir starten an der Quelle s. Die Konstruktion der ausgewählten Kanten stellt sicher, dass wir Fluss über alle aus der Quelle ausgehenden ausgewählten Kanten zur Senke weiterleiten können. Mit C_m bezeichnen wir die Kanten aus S_m, über die Fluss geleitet wird. Also setzten wir

$$C_0(x, y) = S_0(x, y).$$

Für die übrigen Schichten leiten wir den Fluss gemäß der Pfade der Länge 2, die in $T_m(x, y, z)$ gespeichert sind, weiter, d.h.

$$C_m(y, z) = \exists x\colon C_{m-1}(x, y) \wedge T_m(x, y, z).$$

Der durch $C_0, \ldots, C_{l-1}$ beschriebene Fluss ist nicht notwendigerweise ein Sperrfluss. Also entfernt man die zugehörigen Kanten aus dem Niveaunetzwerk und iteriert das Verfahren,

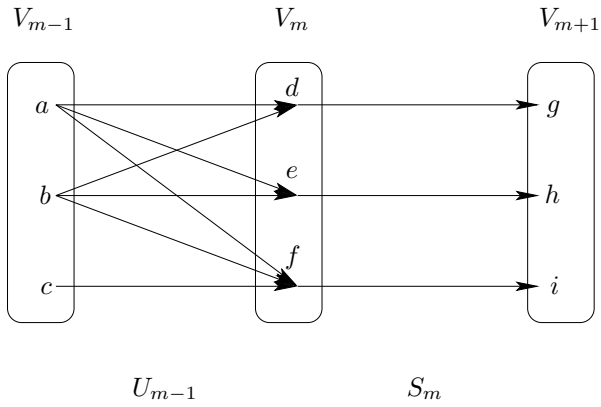


Abbildung 27:

bis es keinen s-t-Pfad mehr im Niveaunetzwerk gibt. Auf diese Weise erhalten wir einen Sperrfluss, der durch die Menge $\psi(x, y)$ von Kanten beschrieben wird, die einen Fluss von 1 haben.

Als letzten Schritt geben wir an, wie der Sperrfluss ψ zu dem Fluss φ auf dem ursprünglichen Netzwerk addiert wird. Bei der Addition erhält eine Kante (x, y) den Fluss 1, wenn sie Vorwärtskante ist und $\psi(x, y) = 1$, wenn sie Rückwärtskante ist und $\psi(y, x) = 0$ oder wenn sie weder Vorwärtskante noch Rückwärtskante ist und $\varphi(x, y) = 1$. Also

$$\varphi_{\text{neu}}(x, y) = (F(x, y) \wedge \psi(x, y)) \vee (B(y, x) \wedge \overline{\psi(y, x)}) \vee (\overline{F(x, y)} \wedge \overline{B(y, x)} \wedge \varphi(x, y)),$$

wobei F und B die Menge aller Vorwärts- bzw. Rückwärtskanten beschreibt. Diese kann man durch

$$\begin{aligned} F(x, y) &= F_0(x, y) \vee \cdots \vee F_{l-1}(x, y) \qquad \text{und} \\ B(x, y) &= B_1(x, y) \vee \cdots \vee B_l(x, y) \end{aligned}$$

berechnen. Das gesamte Verfahren wird nun iteriert, bis die Senke nicht mehr im Niveaunetzwerk enthalten ist. Es sollte klar sein, dass es schwierig bis unmöglich ist, die Rechenzeit auf eine vernünftige Weise abzuschätzen. Dieser Algorithmus ist also nur sinnvoll, wenn man mit Hilfe von Experimenten zeigen kann, dass man damit Flussprobleme lösen kann, die konventionell nicht lösbar sind. Wir beachten weiterhin, dass die Nummerierung der Knoten die Größen der benutzten OBDDs und damit die Effizienz beeinflussen kann. Es lohnt sich daher, die zu lösenden Flussprobleme genauer zu analysieren und zu versuchen, passende Knotennummerierungen zu finden. Wir merken noch an, dass in dieser Darstellung einige Verbesserungen des Algorithmus von Hachtel und Somenzi weggelassen sind.

Abschließend wollen wir zwei der Schritte des Algorithmus mit Beispielen verdeutlichen. Das erste Beispiel bezieht sich auf die Berechnung der Menge S_{m-1} aus S_m. Wir betrachten die Situation in Abbildung 27.

Die Menge S_m besteht aus den Kanten $(d, g), (e, h), (f, i)$. Im ersten Schritt werden die Pfade der Länge 2 bestimmt, die sich aus Kanten aus U_{m-1} und S_m zusammensetzen lassen. Dies ergibt die Menge P_m, die aus $(a, d, g), (a, e, h), (a, f, i), (b, d, g), (b, e, h), (b, f, i), (c, f, i)$

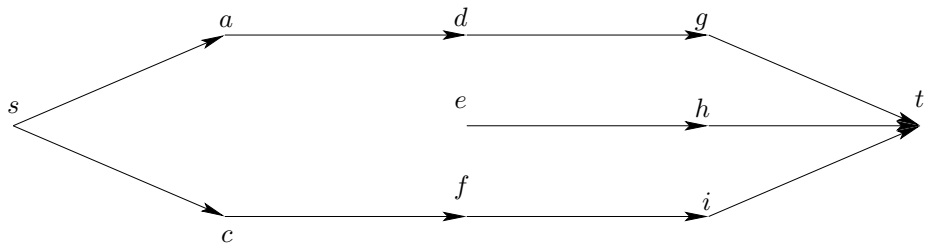


Abbildung 28:

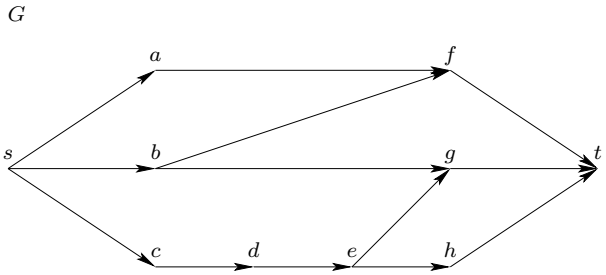
besteht. Durch Existenzquantifizierung wird der mittlere Knoten entfernt, also besteht P_m^* aus den Paaren $(a,g),(a,h),(a,i),(b,g),(b,h),(b,i),(c,i)$. Wir geben nun eine Prioritätsfunktion vor, indem wir die benötigten Ordnungen $\leq_x$ angeben:

$$g \leq_a h \leq_a i, \quad g \leq_b h \leq_b i, \quad i \leq_c h \leq_c g, \quad a \leq_g b \leq_g c.$$

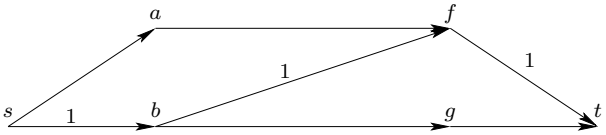
Damit erhalten wir für Q_m die Menge $(a,g),(b,g),(c,i)$ und für D_m die Menge $(a,g),(c,i)$ von Paaren. Schließlich ergibt sich damit T_m als $(a,d,g),(c,f,i)$ und S_{m-1} als (a,d) und (c,f). Wir sehen, dass S_{m-1} kleiner als nötig ist, es hätte auch noch die Kante (b,e) enthalten können. Dass diese Kanten nicht in S_{m-1} aufgenommen wurde, liegt an der Wahl der Prioritätsfunktion. Um ein besseres Gefühl für die Prioritätsfunktion zu bekommen, kann man versuchen, eine bessere Funktion für dieses Beispiel zu finden.

Als Beispiel für die Berechnung des Flusses betrachten wir das Niveaunetzwerk in Abbildung 28, in dem nur die ausgewählten Kanten (d.h. die Kanten aus den Mengen $S.$) eingezeichnet sind und was auf dem zuvor betrachteten Beispiel basiert. Es sollte sofort klar sein, warum die Flussberechnung einen zweiten Durchlauf beginnend an der Quelle erfordert: Wir haben keine andere Möglichkeit herauszufinden, dass über den Pfad e,h,t kein Fluss geleitet werden kann. Aufgrund der Wahl der ausgewählten Kanten ist es dagegen nicht möglich, dass Fluss auf einem Pfad von der Quelle an irgendeiner Stelle nicht mehr weitergeleitet werden kann.

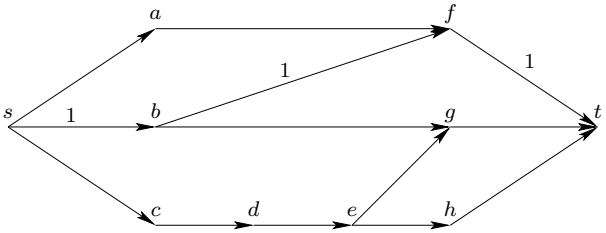
Wir geben nun nur noch die berechneten Mengen von Kanten mit Fluss 1 an:
$C_0 = \{(s,a),(s,c)\}$; $C_1 = \{(a,d),(c,f)\}$; $C_2 = \{(d,g),(f,i)\}$; $C_3 = \{(g,t),(i,t)\}$.



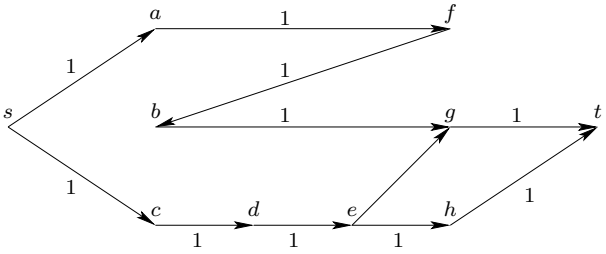
1. Niveaunetzwerk mit Sperrfluss



G nach der ersten Flussvergrößerung



2. Niveaunetzwerk mit Sperrfluss



G nach der zweiten Flussvergrößerung

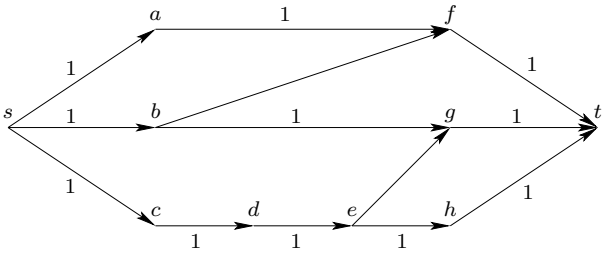


Abbildung 29: Ein Beispiel für eine Flussberechnung mit Hilfe von Niveaunetzwerken