Universität Trier

Fachbereich IV - Informatik

Christoph Meinel:

Binäre Entscheidungsgraphen

Vorlesungsskript Sommersemester 1997

Inhaltsverzeichnis

1 Boolesche Algebra **1**

1.1 Notationen 1
1.2 Boolesche Algebra 2
1.2.1 Algebraische Strukturen 2
1.2.2 Posets 2
1.2.3 Verbände 3
1.2.4 Boolesche Algebra 4
1.2.5 Beispiele und Eigenschaften Boolescher Algebren 5
1.3 Boolesche Funktionen 5
1.4 Tabellendarstellung 7
1.5 Shannon'scher Entwicklungssatz 7
1.6 Boolesche Formeln 9
1.7 Partielle Boolesche Funktionen 11

2 OBDDs – Datenstruktur für Schaltfunktionen **13**

2.1 Einführung 13
2.1.1 Beispiele 13
2.1.2 OBDDs für wichtige Funktionen 14
2.1.3 Konstruktion von OBDDs 15
2.1.4 Reduzierte OBDDs 16
2.1.5 Weitere Eigenschaften 17
2.1.6 Einfluß der Variablenordnung 17
2.1.7 Praktische Arbeit mit OBDDs 19
2.1.8 Darstellung partieller Funktionen 20
2.2 Grundlegende Datenstruktur und Algorithmen 21
2.2.1 Datenstruktur der OBDDs 21

2.3 ite–Synthesealgorithmus für OBDDs 23
2.3.1 OBDD-Konstruktion durch „Symbolische Simulation“ 23
2.3.2 Kompositionsalgorithmus für OBDDs 23
2.3.3 Implementierung der Unique-Tabelle 24
2.3.4 Implementierung der Computed-Tabelle 24
2.3.5 Pseudocode für den *ite*-Algorithmus 25
2.3.6 Komplexitätsanalyse 26
2.3.7 Nutzen komplementierter Kanten 26
2.3.8 Nutzen der Computed-Tabelle 27
2.3.9 Memory Management 29

3 Grundlegende OBDD-basierte Algorithmen 31
3.1 Implikationstest 31
3.2 Erfüllbarkeitsproblem 32
3.2.1 Finden erfüllender Belegung 33
3.2.2 Zahl der erfüllenden Belegungen 34
3.2.3 Finde billigste erfüllende Belegung 35
3.3 Komposition von Funktionen 35
3.4 Quantifizierung 36
3.5 Berechnung verallgemeinerter Kofaktoren 37
3.5.1 Orthogonale Entwicklung von Schaltfunktionen 37
3.5.2 Constrain Operator 39
3.5.3 Restrict Operator 42
3.6 Mengenoperationen und Relationen 44

4 Optimierung der Variablenordnung 47
4.1 Einführung 47
4.2 Bestimmung einer optimalen Variablenordnung 48
4.2.1 Komplexitätsbetrachtung 48
4.2.2 Algorithmus zur Bestimmung einer optimaler Variablenordungen . . 48
4.3 Heuristiken zur Bestimmung guter Variablenordnungen 53
4.3.1 Single-Output Funktionen 53
4.3.2 Multiple-Output Funktionen 56
4.4 Iterative Verbesserung der Variablenordnung 57
4.4.1 Möglichkeiten zum Vertauschen von Variablen 58

4.4.2 Static Reordering und Dynamic Reordering 60

5 Erreichbarkeitsanalyse und Verifikation sequentieller Schaltkreise 62

5.1 Vorbemerkung 62
5.2 Äquivalenztest für Finite State Machines (FSMs) 63
5.3 FSM Traversal 65
5.3.1 Depth-First Traversal 66
5.3.2 Symbolisches Breadth-First Traversal 67
5.4 Bildberechnung 68
5.4.1 Allgemeine Problemstellung 68
5.4.2 Bildberechnung für die Übergangsfunktion 69
5.4.3 Bildberechnung für die Übergangsrelation 73
5.5 Iteriertes Quadrieren 79
5.6 Approximative Erreichbarkeitsanalyse 81
5.7 Erreichbarkeitsanalyse bei unvollständig spezifizierter FSM 82
5.8 Erreichbarkeisanalyse für nondeterministische FSM 82

Überarbeitung: Harald Sack und Christian Stangier
LaTeXsatz: Joachim Marder, Harald Sack und Christian Stangier

Fehler und Änderungsvorschläge bitte an
sack@TI.uni-trier.de
weiterleiten.

Kapitel 1

Boolesche Algebra

1.1 Notationen

Mengen

- S : Menge von Objekten
- #S : Kardinalität von S (mit $\#S < \infty$)
- $\emptyset$: leere Menge

Operationen auf Mengen

- $\overline{A}$: Komplement
- $\cap$: Durchschnitt
- $\cup$: Vereinigung
- $A - B := A \cap \overline{B}$
- $A \times B := \{(x, y) \mid x \in A, y \in B\}$
- $2^A := \{B \mid B \subseteq A\}$

Relationen

- $\subseteq$: Inklusion
- A, B Mengen
- $R \subseteq A \times B$ heißt Relation zu A und B (hier: binäre Relation)
- Schreibe $xRy \iff (x, y) \in R$

- $R^{-1} := \{(x, y) \mid yRx\}$

Funktionen

- $f : A \to B$
- A : Domain
- B : Co-Domain

1.2 Boolesche Algebra

1.2.1 Algebraische Strukturen

$f : A^n \to A$ definiert n-äre Operation auf A.

Ein algebraisches System ist eine Menge A (Träger/Carrier) zusammen mit einer Menge von Operationen und einer Menge von Relationen über $A^n; n \in \mathbb{N}$.

Beispiel 1.2.1:

- $[K, +, \cdot, 0, 1]$ Körper
- $[G, \cdot, 1]$ Gruppen
- $[\{0, 1\}, \wedge, \vee, 0, 1]$

1.2.2 Posets

Eine Relation $\leq \subseteq A^2$ heißt partielle Ordnung (Halbordnung,**P**artial **O**rderded set), falls gilt:

- $\leq$ ist reflexiv, (d. h. $x \leq x$)
- $\leq$ ist antisymmetrisch (d. h. $x \leq y \wedge y \leq x \Rightarrow x = y$)
- $\leq$ ist transitiv (d. h. $x \leq y \wedge y \leq z \Rightarrow x \leq z$).

Ein algebraisches System $P = [A, \leq]$ heißt Poset.

Eigenschaften

- m heißt *meet* von $a, b \in P$, falls $m \leq a, b$ und $\forall m' \leq a, b : m' \leq m$
  (Schreibe $m = a \cdot b = ab$)
- m heißt *join* von $a, b \in P$, falls $m \geq a, b$ und $\forall m' \geq a, b : m \leq m'$
  (Schreibe $m = a + b$).

Bemerkung 1.2.2: *meet* $\equiv$ Infimum, *join* $\equiv$ Supremum.

1.2.3 Verbände

Definition 1.2.3: $[P, \leq]$ heißt *Verband*, falls für alle $a, b \in P$ meet und join existieren.

Satz 1.2.4 *Sei $[P, \leq]$ Verband; $P \neq \emptyset$, so gilt für alle $x, y \in P$:*

1. $x + x = x; \quad xx = x$ *(Idempotenz)*
2. $x + y = y + x; \quad xy = yx$ *(Kommutativität)*
3. $(x + y) + z = x + (y + z); \quad (xy)z = x(yz)$ *(Assoziativität)*
4. $x(x + y) = x; x + (xy) = x$ *(Absorption)*

Beweis: Wir zeigen nur den Beweis der Absorption:
Es gilt: $x \leq z \Leftrightarrow xz = x$ und $x + z = z$.
Also folgt aus $x \leq (x + y)$ die Behauptung.
Rest Übungsaufgabe. □

Lemma 1.2.5: Sei $[P, \leq]$ Verband, dann gilt: $x(y + z) \geq (xy) + (xz)$

Beweis: Es genügt zu zeigen: $xy \leq x(y + z)$ und $xz \leq x(y + z)$.
Es gilt:

$$\left.\begin{array}{r} xy \leq x \\ xy \leq (y + z) \end{array}\right\} \Rightarrow xy \leq x(y + z)$$

$xz \leq x(y + z)$ folgt analog.
Ebenso kann man $x + (yx) \leq (x + y)(y + z)$ beweisen. □

Bemerkung 1.2.6: Alle endlichen Verbände haben ein größtes Element (1) und ein kleinstes Element (0).

Satz 1.2.7 (Dualitätsprinzip) *In einem endlichen Verband mit 0,1 kann aus jeder wahren Aussage eine neue wahre Aussage gewonnen werden, durch Vertauschung von:*

1. *$+$ und $\cdot$*
2. *$\leq$ und $\geq$*
3. *0 und 1*

Beweis: Übungsaufgabe. □

Definition 1.2.8: Gilt $xy = 0$ und $x + y = 1$, dann ist x *komplementär* zu y (Bezeichnung: $\overline{y}$ bzw. $\neg y$).

Definition 1.2.9: Ein Verband heißt *distributiv*, wenn gilt: $x(y + z) = (xy) + (xz)$.

Bemerkung 1.2.10: Mit Hilfe des Dualitätsprinzips folgt aus $x(y + z) = (xy) + (xz)$ direkt $x + (yz) = (x + y)(x + z)$.

Definition 1.2.11: Ein Verband heißt *komplementär*, wenn zu jedem Element ein Komplement existiert.

1.2.4 Boolesche Algebra

Definition 1.2.12: Eine *Boolsche Algebra* ist ein komplementärer, distributiver Verband.

Satz 1.2.13 *In einer Booleschen Algebra gilt:*

1. $x + x = x; \quad xx = x$ *(Idempotenz)*
2. $x + y = y + x; \quad xy = yx$ *(Kommutativität)*
3. $x + (y + z) = (x + y) + z; \quad x(yz) = (xy)z$ *(Assoziativität)*
4. $x(x + y) = x; \quad x + (xy) = x$ *(Absorption)*
5. $x + (yz) = (x + y)(x + z); \quad x(y + z) = (xy) + (xz)$ *(Distributivität)*
6. $\forall x \exists x' : xx' = 0$ *und* $x + x' = 1$ *(Komplement).*

Beweis: Übungsaufgabe □

Bemerkung 1.2.14: Die Eigenschaften (1)-(6) können als Axiome für die Definition einer Booleschen Algebra genommen werden.

Bemerkung 1.2.15: Zu (1)-(6) sind die folgenden Axiome äquivalent (*Huntington*):

1. $x + y = y + x; \quad x \cdot y = y \cdot x$
2. $x + (yz) = (x + y)(x + z); \quad x \cdot (y + z) = (x \cdot y) + (x \cdot z)$
3. $x + 0 = x; \quad x \cdot 1 = x$
4. $x + \overline{x} = 1; \quad x \cdot \overline{x} = 0$
5. $0 \neq 1$.

Beispiel 1.2.16: $(\{0,1\}^n, \vee, \wedge, \underline{0}, \underline{1})$

1.2.5 Beispiele und Eigenschaften Boolescher Algebren

1. $(\{0,1\}, \vee, \wedge, 0, 1)$ „Schaltalgebra“, diese ist mit die wichtigste unter allen Booleschen Algebren. Oft spricht man nur von ,,der“ Booleschen Algebra, meint dann aber diese Schaltalgebra.

2. Sei S eine endliche Menge, dann bezeichnet $(2^S, \cap, \cup, \emptyset, S)$ eine Potenzmengen-BA.

3. $(T_n, \text{ggT}, \text{kgV}, 1, n) \quad T_n := \{t : t \mid n\}$

Satz 1.2.17 (Stone'scher Repräsentationssatz) *Jede endliche Boolesche Algebra ist isomorph zu einer Potenzmengen-BA.*

$\Rightarrow$ *Eigentlicher Unterschied zwischen verschiedenen Booleschen Algebren ist die Kardinalität der Grundmenge, die eine Zweierpotenz sein muß.*

Eigenschaften einer Booleschen Algebra

- $\overline{\overline{x}} = x$ (Involution)
- $\overline{x + y} = \bar{x}\bar{y}; \quad \overline{xy} = \overline{x} + \overline{y}$ (DeMorgan)
- $xy + \overline{x}z + yz = xy + \overline{x}z; \quad (x + y)(\overline{x} + z)(y + z) = (x + y)(\overline{x} + z)$ (Consensus).

1.3 Boolesche Funktionen

Definition 1.3.1: Sei $\mathbb{B}$ Boolesche Algebra. Eine *Boolesche Funktion* in n Variablen (über $\mathbb{B}^n$) ist rekursiv definiert:

1. für alle $b \in \mathbb{B}$ ist $f(x_1, \ldots, x_n) = b$ eine n-stellige Boolesche Funktion (Konstante)

2. für alle $x_i \in \{x_1, \ldots, x_n\}$ ist die Projektion $f(x_1, \ldots, x_n) = x_i$ eine n-stellige Boolesche Funktion

3. Sind g, h n-stellige Boolesche Funktionen, so auch

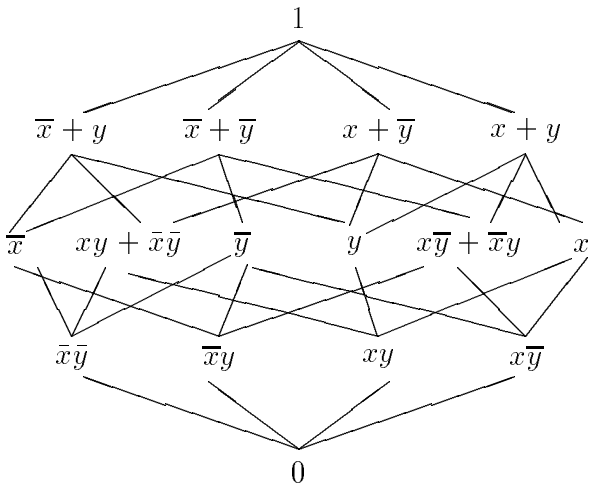
   - $(g + h)(x_1, \ldots, x_n) = g(x_1, \ldots, x_n) + h(x_1, \ldots, x_n)$,
   - $(gh)(x_1, \ldots, x_n) = g(x_1, \ldots, x_n)h(x_1, \ldots, x_n)$,
   - $\overline{g}(x_1, \ldots, x_n) = \overline{g(x_1, \ldots, x_n)}$.

4. Nur die auf obige Weise in endlich vielen Schritten ableitbaren Funktionen sind n-stellige Boolesche Funktionen mit Werten aus $\mathbb{B}$.

Satz 1.3.2 *Die Menge der n-stelligen Booleschen Funktionen über einer Booleschen Algebra $\mathbb{B}$ ist selbst wieder eine Boolesche Algebra (meet entspricht der Addition und join der Multiplikation von Funktionen. Das Nullelement ist die konstante 0-Funktion, und das Einselement die konstante 1-Funktion).*

Beweis: Übungsaufgabe. □

Beispiel 1.3.3: 1. Boolesche Algebra der 2-stelligen Booleschen Funktionen mit Werten in $\mathbb{B} = \{0, 1\}$:



2. $[a, b] = \{s \mid a \leq s \leq b\} \subseteq \mathbb{B}$ ist eine Boolesche Unteralgebra.

   Die Booleschen Funktionen über [a,b] bilden eine Boolesche Algebra

   (z. B. : $a = x\overline{y}, b = x + y$).

Bemerkung 1.3.4: Nicht jede n-stellige Funktion über einer Booleschen Algebra $\mathbb{B}$ ist eine Boolesche Funktion. Denn es gibt $\#\mathbb{B}^{\#\mathbb{B}^n}$ verschiedene Funktionen über $\mathbb{B}^n$, aber nur $\#\mathbb{B}^{2^n}$ Boolesche Funktionen. Für die Schaltalgebra gilt $\#\mathbb{B} = 2$ also 2^{2^n} Boolesche Funktionen = Funktionen über $\{0, 1\}$.

1.4 Tabellendarstellung

Betrachte eine Schaltfunktion, d. h. eine Boolesche Funktion über der Schaltalgebra. Wir können diese durch eine Wahrheitstabelle beschreiben.

#	x_1	x_2	x_3	x_4	f
0	0	0	0	0	0
1	0	0	0	1	1
2	0	0	1	0	0
3	0	0	1	1	1
4	0	1	0	0	0
5	0	1	0	1	1
6	0	1	1	0	0
7	0	1	1	1	0
8	1	0	0	0	0
9	1	0	0	1	1
10	1	0	1	0	0
11	1	0	1	1	1
12	1	1	0	0	0
13	1	1	0	1	1
14	1	1	1	0	1
15	1	1	1	1	1

Problem: Äquivalenz zweier Boolescher Funktionen in Tabellenrepräsentation benötigt exponentiell viele Vergleiche.

Wir können diese Tabelle vereinfachen, indem wir alle 0- (bzw. 1-) Einträge weglassen.
Spezielles Arrangement : *Karnaugh-Veitch*-Diagramm

	x_1x_2			
x_3x_4	00	01	11	10
00	0	0	0	0
01	1	1	1	1
11	1	0	1	1
10	0	0	1	0

1.5 Shannon'scher Entwicklungssatz

Satz 1.5.1 (Shannon'scher Entwicklungssatz) *Sei* $\mathbb{B}$ *eine Boolesche Algebra. Ist* $f : \mathbb{B}^n \to \mathbb{B}$ *eine Boolesche Funktion, so gilt für alle* $(x_1, \ldots, x_n) \in \mathbb{B}^n$*:*

$$f(x_1, \ldots, x_n) = \overline{x_1} f(0, x_2, \ldots, x_n) + x_1 f(1, x_2, \ldots, x_n)$$

Beweis: Induktion über Aufbau der Booleschen Funktion f:

IA: i) f ist Konstante

$$\begin{aligned} f(x_1,\ldots,x_n) &= b \qquad (b \in \mathbb{B}) \\ &= 1 \cdot b \\ &= (\overline{x_1} + x_1)b \\ &= \overline{x_1}b + x_1 b \\ &= \overline{x_1} f(0,x_2,\ldots,x_n) + x_1 f(1,x_2,\ldots,x_n) \end{aligned}$$

ii) f ist Projektion $f(x_1,\ldots,x_n) = x_i$

a) $i = 1$

$$\begin{aligned} \Rightarrow f(x_1,\ldots,x_n) &= x_1 \\ &= \overline{x_1} \cdot 0 + x_1 \cdot 1 \\ &= \overline{x_1} f(0,x_2,\ldots,x_n) + x_1 f(1,x_2,\ldots,x_n) \end{aligned}$$

b) $i > 1$

$$\begin{aligned} \Rightarrow f(x_1,\ldots,x_n) &= x_i \\ &= 1 \cdot x_i \\ &= (\overline{x_1} + x_1)x_i \\ &= \overline{x_1}x_i + x_1 x_i \\ &= \overline{x_1} f(0,x_2,\ldots,x_n) + x_1 f(1,x_2,\ldots,x_n) \end{aligned}$$

IS: Sei $f = g \cdot h$ und gelte

$$\begin{aligned} g(x_1,\ldots,x_n) &= \overline{x_1} g(0,x_2,\ldots,x_n) + x_1 g(1,x_2,\ldots,x_n); \\ h(x_1,\ldots,x_n) &= \overline{x_1} h(0,x_2,\ldots,x_n) + x_1 h(1,x_2,\ldots,x_n) \end{aligned}$$

Nun gilt:

$$\begin{aligned} f(x_1,\ldots,x_n) &= g(x_1,\ldots,x_n)h(x_1,\ldots,x_n) \\ &= (\overline{x_1} g(0,x_2,\ldots,x_n) + x_1 g(1,x_2,\ldots,x_n))(\overline{x_1} h(0,x_2,\ldots,x_n) \\ &\quad + x_1 h(1,x_2,\ldots,x_n)) \\ &= \overline{x_1}\ \overline{x_1}\ g(0,x_2,\ldots,x_n)h(0,x_2,\ldots,x_n) \\ &\quad + \overline{x_1} x_1 g(0,x_2,\ldots,x_n)h(1,x_2,\ldots,x_n) \\ &\quad + x_1 \overline{x_1} g(1,x_2,\ldots,x_n)h(0,x_2,\ldots,x_n) \\ &\quad + x_1 x_1 g(1,x_2,\ldots,x_n)h(1,x_2,\ldots,x_n) \\ &= \overline{x_1} f(0,x_2,\ldots,x_n) + x_1 f(1,x_2,\ldots,x_n) \end{aligned}$$

Analog: $f = g + h, f = \overline{g}$

□

Satz 1.5.2 *Sei* $\mathbb{B}$ *eine Boolesche Algebra und* f *eine n-stellige Boolesche Funktion, so gilt:*

$$f(x_1, \ldots, x_n) = [\overline{x_1} + f(1, x_2, \ldots, x_n)][x_1 + f(0, x_2, \ldots, x_n)]$$

Beweis: Folgt aus Dualitätssatz, Distributivgesetz und Consensus. □

Bemerkung 1.5.3: Wir können den Shannon'schen Entwicklungssatz verallgemeinern auf:

- beliebige x_i anstelle von x_1
- mehrere Variablen $x_{i_1}, \ldots, x_{i_r}$.

1.6 Boolesche Formeln

Wir können Boolesche Funktionen mit Hilfe von Booleschen Formeln darstellen.

Achtung: Boolesche Formeln stellen Boolesche Funktionen nicht eindeutig dar!

Beispiel 1.6.1: Seien G, H Boolesche Formeln, $G = 1$ und $H = x \vee \overline{x}$. H und G stellen dieselbe Funktion dar.

Definition 1.6.2: Sei $\mathbb{B}$ eine Boolesche Algebra. Eine *Boolesche Formel* über n Symbole $x_1, \ldots, x_n$ ist rekursiv definiert:

1. Die Elemente von $\mathbb{B}$ sind Boolesche Formeln.
2. Die Symbole $x_1, \ldots, x_n$ sind Boolesche Formeln.
3. Sind G, H Boolesche Formeln, so auch $(G) + (H)$; $\quad (G) \cdot (H)$; $\quad (\overline{G})$
4. Eine Zeichenkette ist nur dann eine Boolesche Formel, wenn sie durch endliche Anwendungen der Regeln 1) - 3) erzeugt werden kann.

Vereinbarung 1.6.3: Vorrangsregel „$\cdot$“ vor „$+$“, dann können viele Klammern weggelassen werden. Wir schreiben $fg + h$ für $((fg) + h)$.

Definition 1.6.4: Wir können jede Boolesche Formel als Boolesche Funktion interpretieren

a) $F = b, \quad b \in \mathbb{B}$, interpretiere $F = b$ als Boolesche Funktion b.

b) $F = x_i$,(eine Boolesche Variable), interpretiere $F = x_i$ als Projektion auf x_i.

c) Sei $F = (G) + (H)$ und werden (G) als f_g und (H) als f_h interpretiert, dann wird F als $f_g + f_h$ interpretiert.

Analog folgt: $F = (G)(H)$, wird interpretiert als $f_g \cdot f_h$ und $F = \overline{(G)}$ wird interpretiert als $f_{\overline{g}}$.

Achtung:

- Verschiedene Formeln können die gleiche Funktion darstellen. Z. B. : $x_1 + \overline{x_1}x_2$ und $x_1 + x_2$.
- Jede Boolesche Funktion kann durch unendlich viele Boolesche Formeln dargestellt werden. Z. B. : F Formel für $f \Rightarrow F + (x_1 + \overline{x_1})^n$ Formel für f, $\forall n \in \mathbb{N}$.
- Die Überprüfung ob zwei verschiedene Boolesche Formeln dieselbe Boolesche Funktion darstellen heißt auch *Äquivalenzproblem* für Boolesche Formeln (NP-vollständig).
- Das Äquivalenzproblem ist ein zentrales Problem in der Logikminimierung, des CAD-Logikentwurfs und der Optimierung.

Shannon-Entwicklung:

$$\begin{aligned} F(x_1, \ldots, x_n) &= \overline{x_1} F(0, x_2, \ldots, x_n) + x_1 F(1, x_2, \ldots, x_n) \\ F_{\overline{x_i}} &:= F(x_1, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_n) \\ F_{x_i} &:= F(x_1, \ldots, x_{i-1}, 1, x_{i+1}, \ldots, x_n) \end{aligned}$$

Kanonische DNF

Sei f eine Schaltfunktion. Die rekursive Anwendung der Shannonentwicklung liefert:

$$\left.\begin{aligned} f(x_1, \ldots, x_n) = \quad & f(0, \ldots, 0)\overline{x_1} \cdots \overline{x_n} \\ & + f(0, \ldots, 0, 1)\overline{x_1} \cdots \overline{x_{n-1}} x_n \\ & \vdots \\ & + f(1, \ldots, 1) x_1 \cdots x_n \end{aligned}\right\} 2^n \text{ Summanden}$$

Man kann zeigen : $(f_{x_1})_{x_2} = (f_{x_2})_{x_1}$

$f(0, \ldots, 0), \ldots, f(1, \ldots, 1) \in \{0, 1\}$ heißen *Diskriminanten* von f.

$\overline{x_1} \ldots \overline{x_n}, \ldots, x_1 \ldots x_n$ heißen *Minterme*.

Definition 1.6.5: Die Formeldarstellung von f:

$$f(x_1, \ldots, x_n) = f(0, \ldots, 0)\overline{x_1} \cdots \overline{x_n} + \cdots + f(1, \ldots, 1) x_1 \cdots x_n$$

heißt *kanonische DNF (kDNF)* von f.

Bemerkung 1.6.6: Die kDNF-Darstellung ist bis auf die Reihenfolge der Summanden eindeutig bestimmt. Die kDNF stellt somit ein wichtiges Hilfsmittel zur Lösung des Äquivalenzproblems dar.

Duale Shannonentwicklung:

$$\begin{aligned} f(x_1,\ldots,x_n) &= [f(0,\ldots,0)+x_1+\cdots+x_n] \\ &\quad\cdot[f(0,\ldots,0,1)+x_1+\cdots+x_{n-1}+\overline{x_n}] \\ &\quad\vdots \\ &\quad\cdot[f(1,\ldots,1)+\overline{x_1}+\cdots+\overline{x_n}] \end{aligned}$$

Definition 1.6.7: Diese Darstellung heißt *kanonische KNF (kKNF).*

$x_1+\cdots+x_n,\ldots,\overline{x_1}+\cdots+\overline{x_n}$ heißen Maxterme.

Achtung: Werden anstelle von Schaltfunktionen beliebige Boolesche Funktionen betrachtet, so ist es nicht mehr so klar, daß die Funktionen eindeutig durch die Diskriminanten bestimmt sind!

1.7 Partielle Boolesche Funktionen

Sei $\mathbb{B}=\{0,1\}$ eine Boolesche Algebra und $f:\mathbb{B}^n \overset{\supset}{\to} \mathbb{B}$ eine partielle Funktion (d. h. nicht vollständig definiert).

Nun können wir eine Menge über f definieren:

- $\mathrm{on}(f)=\{x\in\{0,1\}^n : f(x)=1\}=f^{-1}(1)$ bezeichnet die *On-Menge* von f, d. h. f ist für alle Werte aus $\mathrm{on}(f)$ gleich 1.
- $\mathrm{off}(f)=\{x\in\{0,1\}^n : f(x)=0\}=f^{-1}(0)$ bezeichnet die *Off-Menge* von f, d. h. f ist für alle Werte aus $\mathrm{off}(f)$ gleich 0.
- $\mathrm{dc}(f)=\{x\in\{0,1\}^n : f(x)\ \text{undefiniert}\}=\{0,1\}^n-(\mathrm{on}(f)\cup\mathrm{off}(f)$ bezeichnet die *Don't Care-Menge* von f. Sie enthält alle Werte, auf denen f undefiniert ist.

Wichtig:
Unvollständig spezifizierte Boolesche Funktionen:

$\mathcal{F}=\{f : g(x)\le f(x)\le h(x);\quad \forall x\in\mathbb{B}^n\}$ (g, h sind Boolesche Funktionen)

Wir betrachten eine Menge S von Intervallen auf $\mathbb{B}$. $S=\{[a,b]\mid a,b\in\mathbb{B}, a<b\}$
Nun können wir f durch eine einzige Abbildung $f:\mathbb{B}^n\to S$ definieren.
$f(x)=[g(x),h(x)]$; für alle $x\in\mathbb{B}^n$

Beispiel 1.7.1: $[0,0]\hat{=}0$; $[0,1]\hat{=}$dc(don't care); $[1,1]\hat{=}1$

Definition 1.7.2: $f:\mathbb{B}^n\to S$ heißt *unvollständig spezifiziert.*

Beispiel 1.7.3: $B = \{0,1\}, S = \{[a,b] : a \leq b\} = \{[0,0],[0,1],[1,1]\}$

$$f : B \to S \qquad f(x) = \begin{cases} [0,0] & \hat{=} & 0 \\ [0,1] & \hat{=} & \text{dc} \quad \text{oder } _ \text{ „don't care“} \\ [1,1] & \hat{=} & 1 \end{cases}$$

x_1	x_2	
0	0	0
0	1	$_$
1	0	$_$
1	1	1

Kapitel 2

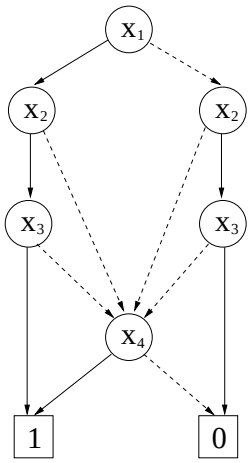
OBDDs – Datenstruktur für Schaltfunktionen

2.1 Einführung

2.1.1 Beispiele

Betrachte $f(x_1, x_2, x_3, x_4) = x_1x_2x_3 + \overline{x_2}x_4 + \overline{x_3}x_4$

Ordered Binary Decision Diagram (OBDD)[1] P für f:



x_1	x_2	x_3	x_4	f
0	1	1	0	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
⋮	⋮	⋮	⋮	⋮

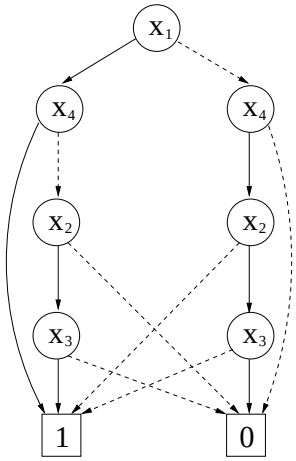
$$f_p : \{0,1\}^4 \to \{0,1\}$$

Der Graph ordnet jeder Argumentfolge eindeutig einen Funktionswert zu. Der Graph definiert die Funktion durch eine Auswertungsvorschrift (DNF, KNF geben eine Berechnungsvorschrift).

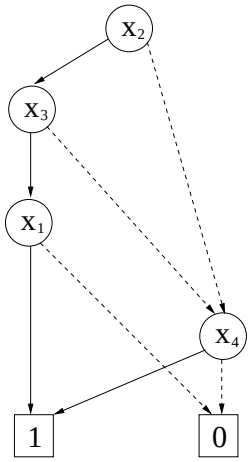
[1] Zur Unterscheidung von then- und else-Kanten zeichnen wir die else-Kanten mit einer gestrichelten Linie.

P ist geordnet, d. h. auf jedem Weg werden die Variablen gemäß einer fixierten Ordnung getestet (Obiges Beispiel: $x_1 < x_2 < x_3 < x_4$).

Betrachte OBDD P' für f mit Ordnung $x_1 < x_4 < x_2 < x_3$:



OBDD P'' für f mit Ordnung $x_2 < x_3 < x_1 < x_4$:

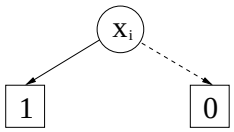


Komplexitätsmaß: Anzahl der inneren Knoten (alle außer Senken).

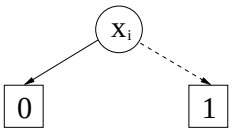
P'' ist optimal für f, da jede Variable nur einmal getestet wird, und f von allen Variablen abhängt.

2.1.2 OBDDs für wichtige Funktionen

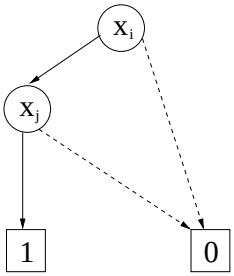
x_i :



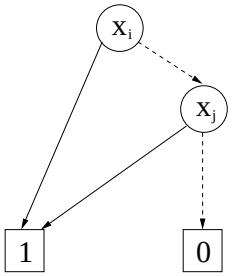
$\overline{x_i}$:



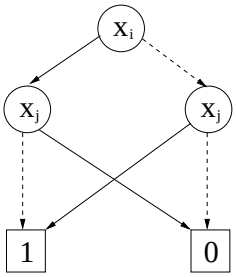
$x_i x_j$:



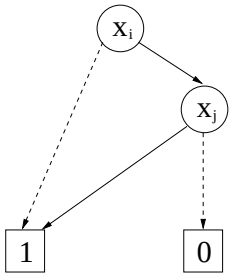
$x_i + x_j$:



$x_i \oplus x_j$:



$x_i \Rightarrow x_j$:

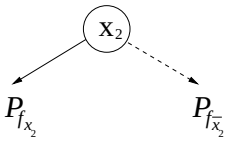


2.1.3 Konstruktion von OBDDs

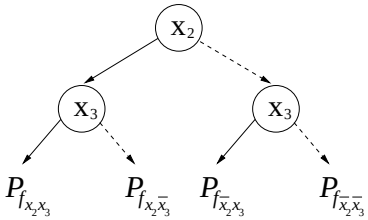
Wir nutzen die *Boole-/Shannonentwicklung*: $f = x_i f_{x_i} + \overline{x_i} f_{\overline{x_i}}$.
$f_{x_i} = f(x_1, \ldots, x_{i-1}, 1, x_{i+1}, \ldots, x_n)$ heißt *positiver Kofaktor* von f an der Stelle x_i.
$f_{\overline{x_i}} = f(x_1, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_n)$ heißt *negativer Kofaktor* von f an der Stelle x_i.

Wir konstruieren den OBDD $f(x_1, x_2, x_3, x_4) = x_1 x_2 x_3 + \overline{x_2} x_4 + \overline{x_3} x_4$ gemäß der Ordnung $x_2 < x_3 < x_4 < x_1$.

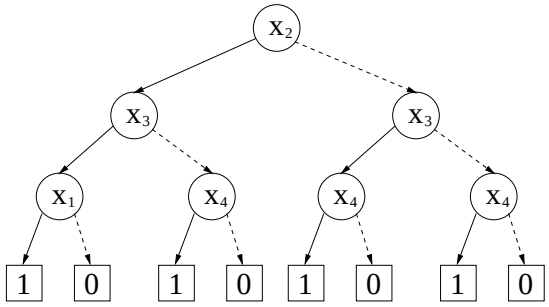
$$f_{x_2}(x_1, x_3, x_4) = x_1 x_3 + \overline{x_3} x_4$$
$$f_{\overline{x_2}}(x_1, x_3, x_4) = x_4 + \overline{x_3} x_4$$



$$f_{x_2 x_3}(x_1, x_4) = x_1$$
$$f_{x_2 \overline{x_3}}(x_1, x_4) = x_4$$
$$f_{\overline{x_2} x_3}(x_1, x_4) = x_4$$
$$f_{\overline{x_2}\,\overline{x_3}}(x_1, x_4) = x_4$$



$f_{x_2x_3x_4} = f_{x_2x_3\overline{x_4}} = x_1$
$f_{x_2\overline{x_3}x_4} = 1$
$f_{x_2\overline{x_3}\,\overline{x_4}} = 0$
$f_{x_2x_3x_4x_1} = 1$
$f_{x_2x_3x_4\overline{x_1}} = 0$



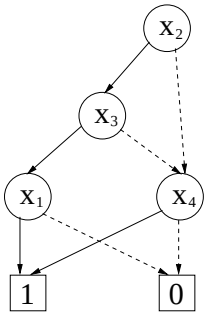
Achtung: Bei der Konstruktion des OBDDs treten viele isomorphe Subgraphen auf, die eliminiert werden können!

2.1.4 Reduzierte OBDDs

Ein OBDD kann durch die Anwendung von Reduktionsregeln in einen reduzierten OBDD transformiert werden.

$f(x_1, x_2, x_3, x_4) = x_1x_2x_3 + \overline{x_2}x_4 + \overline{x_3}x_4$ Ordnung: $x_2 < x_3 < x_4 < x_1$.

- Die Größe von P kann durch Identifizierung isomorpher Untergraphen verringert werden (*Algebraische Reduktion, Merging-rule*).
- Triviale Knoten können entfernt werden (*Eliminationsregel, Deletion-rule*).



Die iterierte Anwendung der Merging- und Deletion-rule führt in endlich vielen Schritten zu einem OBDD P' ohne isomorphe Untergraphen und ohne redundante Knoten.

Definition 2.1.1: Ein OBDD heißt *reduziert*, wenn er keine isomorphen Untergraphen und keine redundanten Knoten enthält.

Bemerkung 2.1.2:

- Die Reduktion verändert die Variablenordnung nicht!

- Der reduzierte OBDD (ROBDD) ist eindeutig bestimmt. Daher stellt ein ROBDD eine kanonische Form dar, d. h. der Test auf funktionale Äquivalenz von ROBDDs gleicher Variablenordnung kann durch Reduktion und anschließenden Isomorphietest durchgeführt werden.
- Der bottom-up Reduktionsalgorithmus nach Bryant (1986) arbeitet in Zeit $O(|P| \log |P|)$ (Sieling/Wegener: $O(|P|)$).
- Die Reduktion wird im allgemeinen während der OBDD-Synthese durchgeführt.

2.1.5 Weitere Eigenschaften

+/- Die OBDD-Darstellung für die „meisten" Funktionen (Bryant: Multiplikation) ist exponentiell bzgl. der Anzahl der Variablen.

\+ Für die meisten praktisch wichtigen Funktionen ist die OBDD-Größe moderat, zumindest bzgl. einer geeigneten Variablenordnung.

\+ Wir können OBDDs für $(f \wedge g), (f \vee g)$ in polynomialer Zeit („effizient") aus der OBDD-Darstellung von f und g konstruieren. Der OBDD für $\overline{f}$ kann in konstanter Zeit berechnet werden. (0- und 1-Senken vertauschen)

\+ Der Erfüllbarkeitstest ($f \not\equiv 0$) und der Tautologietest ($f \equiv 1$) können in konstanter Zeit gelöst werden.

\- Die OBDD-Größe übersteigt manchmal die Größe von DNF und PNF.

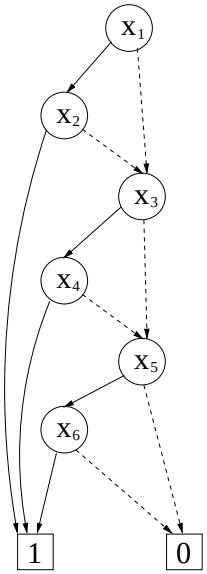
\- DNF bzw. PNF sind implementierungsnäher als OBDDs (PLA-Realisierung). Allerdings können auch OBDDs mittels Multiplexer-Realisierung direkt implementiert werden. Multiplexer funktionieren wie Schalter.

Bemerkung 2.1.3: Die OBDD-Größe hängt stark von der Variablenordnung ab. Oft ist es schwierig eine gute Variablenordnung zu finden.

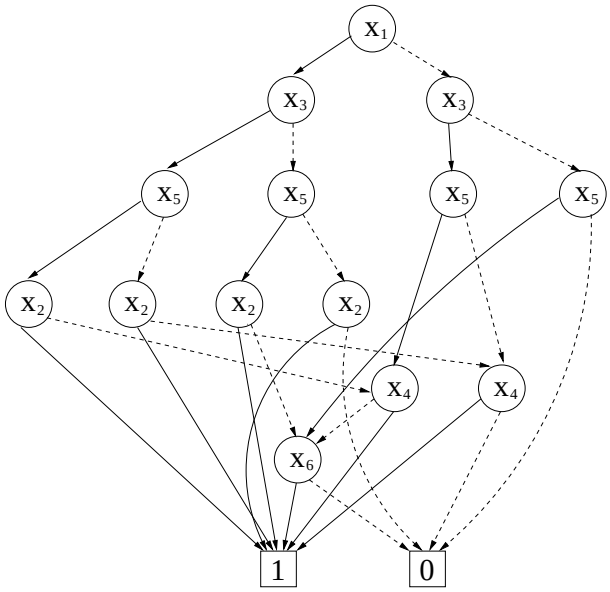
2.1.6 Einfluß der Variablenordnung

$f(x_1, \ldots, x_6) = x_1x_2 + x_3x_4 + x_5x_6$

- $\pi_1 : x_1 < x_2 < x_3 < x_4 < x_5 < x_6$



- $\pi_2 : x_1 < x_3 < x_5 < x_2 < x_4 < x_6$



Beide OBDDs sind reduziert.

Die Ordnung π_1 ermöglicht das Ablesen der einzelnen Monome.

Die Ordnung π_2 erzwingt, daß alle möglichen Belegungen der Variablen x_1, x_3, x_5 gespeichert werden, um dann entsprechend reagieren zu können.

Die Bestimmung der optimalen Ordnung ist NP-hart (OBDD,DNF,...).

Bemerkung 2.1.4: Im naiven Ansatz können wir eine Funktionstabelle erstellen und für alle der $n!$ Permutationen einen vollständigen Entscheidungsbaum erstellen, reduzieren

und anschließend die Größe der reduzierten OBDDs vergleichen ($O(n!2^n)$ Schritte). Der Algorithmus von Friedman/Supowit (1990) arbeitet in Zeit $O(n^2 3^n)$ und benötigt Platz $O(3^n/n^{1/2})$ (Sieling/Wegener: Zeit $O(n3^n)$).

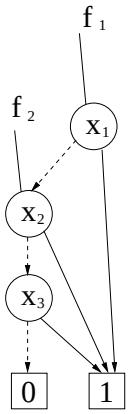
2.1.7 Praktische Arbeit mit OBDDs

Um praktisch mit OBDDs arbeiten zu können, wird ein OBDD-basiertes Programmsystem „OBDD-Paket“ benötigt. Es gibt verschiedene Implementierungen (Brace, Rudell, Bryant CMU, 90; Long; Somenzi, CUDD).

Entwurfssystem für gute OBDD-Pakete beinhalten:

Shared-OBDDs: Mit jedem OBDD-Knoten ist eine Funktion assoziert. In der Praxis haben wir es mit Funktionen $F = (f_1, \ldots, f_m) \in \mathbb{B}_{n,m}, m > 1$ und nicht mit $f_i \in \mathbb{B}_{n,1}$ zu tun. Komponentenfunktionen haben oft gemeinsame Unterfunktionen, wie z. B. $F = (f_1, f_2)$ mit $f_1 = x_2 + x_3; \quad f_2 = x_1 + x_2 + x_3$.

Nun können wir die Darstellung von f_1 und f_2 „mischen“:



Äquivalente Funktionen werden vom gleichen OBDD repräsentiert.

Unique Table: Wir sind an einer Darstellung reduzierter OBDDs interessiert und wollen keine unreduzierten OBDDs erzeugen, um diese dann später zu reduzieren. Wir wollen ausschließlich mit reduzierten OBDDs arbeiten (D. h.: zu keinem Zeitpunkt sollen verschiedene isomorphe Untergraphen oder redundante Knoten gespeichert werden.).

Idee: Wir überprüfen vor dem Einfügen eines neuen Knotens, ob die entsprechende Funktion bereits vorhanden ist. Das können wir effizient implementieren:

Wir speichern alle bereits repräsentierten Funktionen in einer Tabelle (*Unique-Tabelle*) ab. Eine effiziente Implementierung ist mit Hash-Tabellen möglich.

Strenge Kanonizität: Wir können mit der Unique-Tabelle einen Äquivalenztest in konstanter Zeit ausführen:

Dazu brauchen wir lediglich zu testen, ob die mit den Funktionen f_1 und f_2 assoziierten Pointer identisch sind.

Attributierte Kanten: Da die OBDDs für f und $\overline{f}$ sehr ähnlich sind (lediglich die Senken sind vertauscht), benutzen wir nur einen Graphen zur Repräsentation beider Funktionen:

Sei v die Wurzel des OBDD für f, dann repräsentieren wir $\overline{f}$ durch einen Pointer auf v mit einem *Komplement-Attribut*. Kanten mit einem solchen Attribut heißen komplementierte (invertierte) Kanten, die übrigen Kanten heißen regulär. Die Benutzung komplementierter Kanten erschwert die Manipulation der OBDDs etwas, aber:

- großer Speichergewinn (max. 50 %).
- Eine Komplementierung ist in konstanter Zeit möglich (sehr aufwendig bei DNF).
- Der Test, ob g das Komplement von f ist, ist in konstanter Zeit möglich.
- Die systematische Arbeit mit komplementierten Kanten macht nur noch eine Senke (1-Senke) erforderlich.
- Die Idee der Attributierung kann noch erweitert werden (Edge-valued-BDDs).

Computed Table: Eine Geschwindigkeitserhöhung ist durch eine Tabellierung der zuletzt bearbeiteten Funktionen möglich: Wir können Fragen der Art „Wurde kürzlich das OR von f_1 und f_2 berechnet?“ beantworten.

Memory Management: Typischerweise werden in Anwendungen viele OBDDs erzeugt und dann wieder freigegeben. Aufgrund der Größe der OBDDs ist eine effiziente Speicherverwaltung erforderlich. Wir benutzen eine *Garbage-Collection*: Die Knoten werden nun nicht mehr sofort, sondern lediglich von Zeit zu Zeit freigegeben.

2.1.8 Darstellung partieller Funktionen

Partielle Funktionen können mit BDDs dargestellt werden, indem man einfach einen Terminalknoten einführt, der den *don't care*-Wert darstellt. Dazu müßten wir unsere Algorithmen und Datenstrukturen entsprechend anpassen. Um dies zu umgehen verwenden wir eine alternative Möglichkeit: Wir können eine partielle Funktion mithilfe von zwei totalen Funktionen spezifizieren.

$$f = (f_{\mathrm{ON}}, f_{\mathrm{DC}})$$

oder

$$f_L \leq f \leq f_U$$

mit $f_{\mathrm{ON}}, f_{\mathrm{DC}}, f_L$ und f_U totale Funktionen:

$$f_{\mathrm{ON}} = f_L \qquad f_{\mathrm{DC}} = f_U \cdot \overline{f_L} \qquad f_U = f_{\mathrm{ON}} + f_{\mathrm{DC}}.$$

Sei $f \in \mathbb{B}^n$, wir definieren eine totale Funktion $\tilde{f} \in \mathbb{B}^{n+1}$:

$$\tilde{f} = f_{\mathrm{ON}} + d \cdot f_{\mathrm{DC}} = \overline{d} \cdot f_L + d \cdot f_U.$$

Es folgt:

$$f_{\mathrm{ON}} = f_L = \tilde{f}_{\overline{d}} \qquad f_U = \tilde{f}_d \qquad f_{\mathrm{DC}} = \tilde{f}_d \cdot \overline{\tilde{f}_{\overline{d}}}.$$

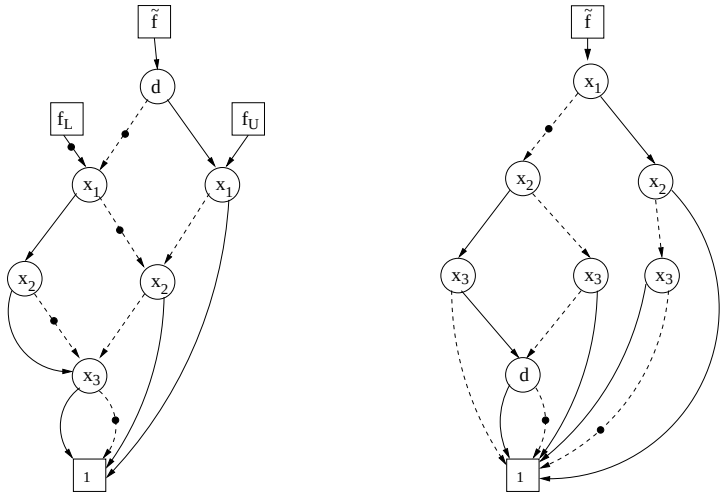
Wir können feststellen, ob für eine Belegung $a_1, \ldots, a_n$ die Funktion $f(a_1, \ldots, a_n)$ gleich 0, 1 oder don't care ist, indem wir festlegen:

$$\begin{aligned} \tilde{f}(a_1, \ldots, a_n, 0) = \tilde{f}(a_1, \ldots, a_n, 1) \quad &\Rightarrow \quad f(a_1, \ldots, a_n) = \tilde{f}(a_1, \ldots, a_n, 0) \\ \tilde{f}(a_1, \ldots, a_n, 0) \neq \tilde{f}(a_1, \ldots, a_n, 1) \quad &\Rightarrow \quad f(a_1, \ldots, a_n) = don't \quad care \end{aligned}$$

Wir können f darstellen durch einen OBDD für $\tilde{f}$. Die Variable d kann in der Variablenordnung an beliebiger Stelle stehen. Ist d die *Topvariable* (Variable mit niedrigstem Index), dann gibt es nur einen Knoten mit Index d, der mit den OBDDs für f_L und f_U verbunden ist. Steht d an letzer Stelle der Variablenordnung, gibt es ebenfalls nur einen Knoten mit Index d. Alle Pfade, die dann im OBDD zu d führen, repräsentieren eine don't-care-Belegungen.

Beispiel 2.1.5:

$$x_2\overline{x_3} + \overline{x_1}x_3 + \overline{x_2}x_3 \leq f \leq x_1 + x_2 + x_3$$



2.2 Grundlegende Datenstruktur und Algorithmen

2.2.1 Datenstruktur der OBDDs

(Brace, Bryant, Rudell, CMU '90)

Definition 2.2.1:
Ein *OBDD* P ist ein azyklischer gerichteter Graph $(V \cup F \cup \{1\}, E)$ zur Repräsentation einer Funktion $f = (f_1, \ldots, f_m) \in \mathbb{B}_{n,m}$ mit

- V: Menge der inneren Knoten $outdeg(v) = 2; \quad \forall v \in V$
- Knotenbelegung: $l : V \to X_n = \{x_1, \ldots, x_n\}$
- 1: terminaler Knoten, $outdeg(1) = 0$
- F: Menge von Funktionsknoten, $\#F = m, \quad \varphi \in F \Rightarrow outdeg(\varphi) = 1, indeg(\varphi) = 0$. Jedes $\varphi \in F$ korrespondiert zu einem $f_i; i = 1, \ldots, m$. Die ausgehende Kante kann komplementiert sein.
- Ausgehende Kanten von inneren Knoten sind bewertet mit T und E (*then*-Kante, *else*-Kante). E-Kanten können komplementiert sein.
- $(l(v), T, E)$ bezeichnet einen inneren Knoten mit ausgehenden Kanten T und E, sowie Knotenbelegung l_v.
- Auf den Variablen aus X_n ist eine Ordnung definiert: $x_{i_1} < x_{i_2} < \ldots < x_{i_n}$. Ist u_j ein Nachfolger von u_i, $(u_i, u_j) \in E$, dann gilt: $l(u_j) > l(u_i)$.
- Die Knoten u und Kanten e repräsentieren Funktionen: f_u bzw. f_e

  1. Ist $l(u) = 1 \Rightarrow f_u \equiv 1$
  2. Ist e eine Kante, die nach u führt, und ist f_u definiert, dann gilt:
  $$f_e \equiv \begin{cases} f_u, & \text{falls } e \text{ nicht komplementiert} \\ f_{\overline{u}}, & \text{falls } e \text{ komplementiert} \end{cases}$$
  3. Ist u ein Knoten mit $(l(u), T, E)$ dann gilt: $f_u \equiv l(u) \cdot f_T + \overline{l(u)} \cdot f_E$
  4. $$\varphi \in F; f_\varphi \equiv \begin{cases} f_e, & \text{falls } e \text{ nicht komplementiert} \\ f_{\overline{e}}, & \text{falls } e \text{ komplementiert} \end{cases}$$
  e ist die φ verlassende Kante.

Es gilt:
OBDDs repräsentieren eine Boolesche Funktion $f = (f_1, \ldots, f_m)$ eindeutig, falls

- alle inneren Knoten Nachfolger der Funktionsknoten sind,
- es keine isomorphen Unterprogramme gibt und
- für alle $u \in V$ mit $(l(u), T, E) : f_T \neq f_E$.

Achtung: Wir brauchen die Festlegung, daß nur E-Kanten komplementiert sein dürfen (Kanonizität)!

2.3 ite–Synthesealgorithmus für OBDDs

2.3.1 OBDD-Konstruktion durch „Symbolische Simulation“

Sei $f \in \mathbb{B}_n$, bzw. $f \in \mathbb{B}_{n,m}$ durch Circuit-Netlist[2] (Schaltkreis) C gegeben.

Konstruktion einer OBDD-Darstellung durch „symbolische Simulation“

Ausgehend von der OBDD-Darstellung für primäre Inputs werden sukzessive OBDDs für die Funktionen erzeugt, die in den einzelnen Nachfolgegattern berechnet werden.

2.3.2 Kompositionsalgorithmus für OBDDs

Wir wollen eine Boolesche Operation $* \in \mathbb{B}_2$ (z. B. AND, OR, XOR,...) auf existierende OBDDs anwenden.

Grundidee: Rekursive Anwendung von $f * g = x_i(f_{x_i} * g_{x_i}) + \overline{x_i}(f_{\overline{x_i}} * g_{\overline{x_i}})$

Anwendung: Seien F, G OBDDs für f, g. Ferner sei x_i die Topvariable.

Berechne nun $F_{x_i}, G_{x_i}, F_{\overline{x_i}}, G_{\overline{x_i}}$ für $f_{x_i}, g_{x_i}, f_{\overline{x_i}}, g_{\overline{x_i}}$.

Wir komponieren x_i, F_{x_i} und G_{x_i} zu H_1 für $x_i(f_{x_i} * g_{x_i})$.

Wir komponieren $\overline{x_i}, F_{\overline{x_i}}$ und $G_{\overline{x_i}}$ zu H_2 für $\overline{x_i}(f_{\overline{x_i}} * g_{\overline{x_i}})$.

Die tatsächliche Realisierung benutzt den *ite-Operator* („if-then-else“-Operator):
$ite(F, G, H) = F \cdot G + \overline{F} \cdot H$, wobei $F, G, H \in B_m$.

Wir können alle binären Operatoren mit *ite*-Operatoren ausdrücken:

	Bezeichnung	Ausdruck	mit *ite*-Operator
0000	0	0	0
0001	$AND(F,G)$	$F \cdot G$	$ite(F, G, 0)$
0010	$F > G$	$F \cdot \overline{G}$	$ite(F, \overline{G}, 0)$
0011	F	F	F
0100	$F < G$	$\overline{F} \cdot G$	$ite(F, 0, G)$
0101	G	G	G
0110	$XOR(F,G)$	$F \oplus G$	$ite(F, \overline{G}, G)$
0111	$OR(F,G)$	$F + G$	$ite(F, 1, G)$
1000	$NOR(F,G)$	$\overline{F + G}$	$ite(F, 0, \overline{G})$
1001	$NXOR(F,G)$	$\overline{F \oplus G}$	$ite(F, G, \overline{G})$
1010	$NOT(G)$	$\overline{G}$	$ite(G, 0, 1)$
1011	$G \Rightarrow F$	$F + \overline{G}$	$ite(F, 1, \overline{G})$
1100	$NOT(F)$	$\overline{F}$	$ite(F, 0, 1)$
1101	$F \Rightarrow G$	$\overline{F} + G$	$ite(F, G, 1)$
1110	$NAND(F,G)$	$\overline{F} + \overline{G}$	$ite(F, \overline{G}, 1)$
1111	1	1	1

[2] Liste von Gattern (AND, OR, EXOR, NAND, ...) und Vernetzungsinformation, die einen Schaltkreis für die Realisierung von f ergibt.

Sei $z = ite(F, G, H)$, ferner sei v Topvariable von F, G, H

$$\begin{aligned}
\Rightarrow z &= v \cdot Z_v + \overline{v} \cdot Z_{\overline{v}} \\
&= v \cdot (F \cdot G + \overline{F} \cdot H)_v + \overline{v}(F \cdot G + \overline{F} \cdot H)_{\overline{v}} \\
&= v \cdot (F_v \cdot G_v + \overline{F}_v \cdot H_v) + \overline{v} \cdot (F_{\overline{v}} \cdot G_{\overline{v}} + \overline{F}_{\overline{v}} \cdot H_{\overline{v}}) \\
&= v \cdot ite(F_v, G_v, H_v) + \overline{v} \cdot ite(F_{\overline{v}}, G_{\overline{v}}, H_{\overline{v}}) \\
&= ite(v, ite(F_v, G_v, H_v), ite(F_{\overline{v}}, G_{\overline{v}}, H_{\overline{v}}))
\end{aligned}$$

Knoten zu $Z : (v, ite(F_v, G_v, H_v), ite(F_{\overline{v}}, G_{\overline{v}}, H_{\overline{v}}))$

Terminalfälle: $F = ite(1, F, G) = ite(0, G, F) = ite(F, 1, 0) = ite(G, F, F)$

Bemerkung 2.3.1: *ite* muß exponentiell oft ausgeführt werden („Brute Force“).

2.3.3 Implementierung der Unique-Tabelle

Die Unique-Tabelle wird als Hash-Tabelle implementiert: Das Tupel (v, G, H), wobei v der Index der Variable, G der Pointer auf den then-Sohn und H der Pointer auf den else-Sohn ist, wird mittels der Hash-Funktion auf einen Wert in $[1, h]$, wobei h die Grösse der Hash-Tabelle ist, abgebildet.

Die Hash-Funktion ist nicht eineindeutig.

$\Rightarrow$ Verschiedene Tupel können in dieselbe Zeile der Tabelle geschrieben werden.

Diese Kollisionen müssen aufgelöst werden. Eine Möglichkeit besteht darin, in der Hashtabelle nicht nur einzelne Tupel abzuspeichern, sondern ganze Listen von Tupeln. Nach der Berechnung der Position innerhalb der Tabelle muß nun die entsprechende Liste durchsucht werden.

Eine gute Hashfunktion, die die Tupel möglichst gleichmäßig verteilt, erlaubt eine kurze Suchzeit.

2.3.4 Implementierung der Computed-Tabelle

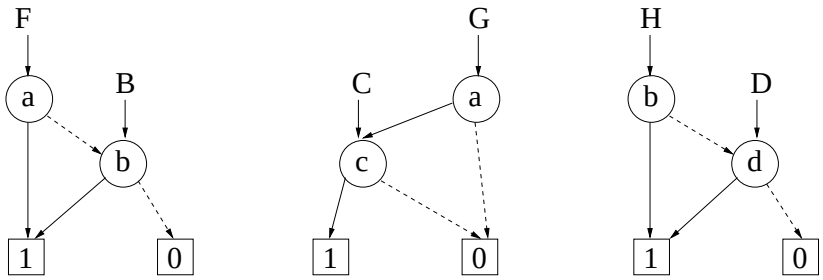
Die Computed-Tabelle wird als hash-basierter Cache realisiert. Als Schlüssel wird (F, G, H) benutzt, wobei F, G und H die Eingänge für $ite(F, G, H)$ sind.

Auf jeder Rekursionsstufe der *ite*-Brechnung wird in der Computed-Tabelle nach diesem Schlüssel gesucht. Falls er existiert und damit die entsprechende *ite*-Operation ausgeführt wurde, wird die Operation nicht noch einmal ausgeführt. Ansonsten wird $ite(F, G, H)$ berechnet und der Schlüssel (F, G, H) in die Computed-Tabelle aufgenommen.

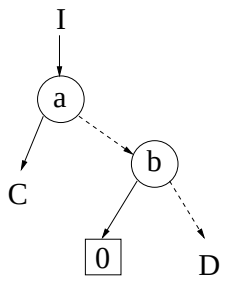
2.3.5 Pseudocode für den *ite*-Algorithmus

```
ite(F,G,H) {
    if (TerminalCase) { return result; }
    else if (Computed-Tabelle enthält (F,G,H)) { return result; }
    else {
        Sei v Topvariable von F,G,H;
        T = ite(F_v, G_v, H_v);
        E = ite(F_v̄, G_v̄, H_v̄);
        if (T==E) { R = T; }
        else {R = find-or-add-unique-table(v,T,E);}
        insert-computed-table((F,G,H),R);
        return R;
    }
}
```

Beispiel 2.3.2: Anwendung des *ite*-Operators:



$$
\begin{aligned}
I &= ite(F,G,H) \\
&= (a, ite(F_a, G_a, H_a), ite(F_{\overline{a}}, G_{\overline{a}}, H_{\overline{a}})) \\
&= (a, ite(1, C, H), ite(B, 0, H)) \\
&= (a, C, (b, ite(B_b, 0_b, H_b), ite(B_{\overline{b}}, 0_{\overline{b}}, H_{\overline{b}}))) \\
&= (a, C, (b, ite(1,0,1), ite(0,0,D))) \\
&= (a, C, (b, 0, D))
\end{aligned}
$$



2.3.6 Komplexitätsanalyse

Ohne Computed Tabelle: Wir nehmen an, daß unbegrenzter Speicher für die Unique-Tabelle zur Verfügung steht, und daß ein Zugriff auf die Unique-Tabelle in konstanter Zeit ausgeführt wird.

- Bis auf die rekursiven Aufrufe benötigen alle Befehle konstante Zeit.
- Jeder Aufruf von *ite* erzeugt zwei andere Aufrufe von *ite*, falls kein terminaler Fall erreicht worden ist.

$\Rightarrow$ Die Komplexität ist exponentiell in der Anzahl der Variablen: $O(2^n)$

Mit Computed Tabelle: *ite* wird für jede Kombination von F, G und H höchstens einmal aufgerufen. $\Rightarrow$ Es gibt maximal $\#F \cdot \#G \cdot \#H$ Aufrufe.

- Sind F, G und H von polynomialer Größe, dann ist auch der Berechnungsaufwand der Synthese polynomial. ($\uparrow$ Dynamic Programming)
- Ist ein Argument von *ite* konstant, dann benötigt *ite* lediglich quadratischen Berechnungsaufwand.
- Sind zwei Argumente von *ite* gleich bzw. komplementiert (unter Verwendung komplementierender Kanten) so ist der Berechnungsaufwand ebenfalls quadratisch (Übungsaufgabe).

Satz 2.3.3 *Mit Hilfe von ite können alle Booleschen Operationen mit quadratischem Rechen- und Speicherplatzaufwand berechnet werden.*

Beweis: Übungsaufgabe. □

2.3.7 Nutzen komplementierter Kanten

Wir wissen bereits: Die Verwendung von komplementierten Kanten ermöglicht die Komplementierung einer Funktion in konstanter Zeit.

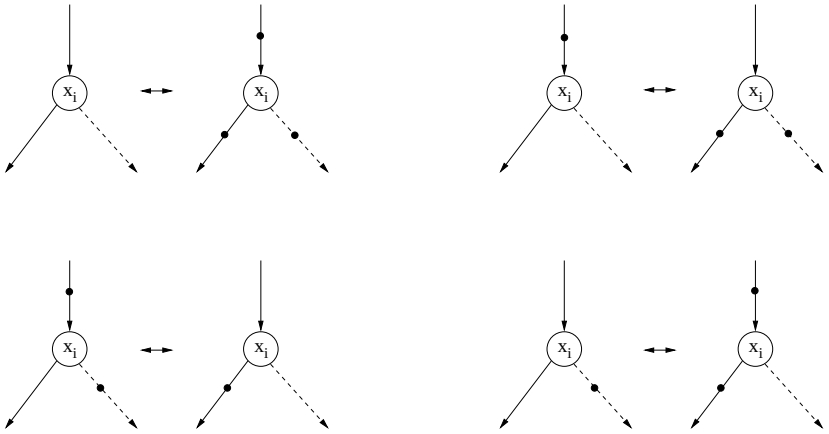
Nun gilt: Alle möglichen Realisierungen der Komplementierung ($2^3 = 8$) können vorkommen.

Um die Kanonizität zu erhalten, legen wir folgendes fest:

T-Kanten werden nie komplementiert.

Wir müssen sichern, daß *ite* jede nicht kanonische Situation in die entsprechende kanonische Situation umwandelt.

Das ist ohne großen Aufwand möglich. Falls die T-Kante komplementiert ist, kann man die Situation wie folgt umwandeln:



Konventionen:

- Wir brauchen nur ein Bit zur Darstellung des Komplement-Attributs, indem wir das *least significant bit* des Kanten-Pointers benutzen.
- Es gilt: $ite(F, 0, 1) = \overline{F}$.

  Die Verwendung komplementierter Kanten ermöglicht einen Abbruch in dieser Situation und die Ausgabe des Komplements des ersten Arguments.
- Komplementierte Kanten ermöglichen eine Verkleinerung der Unique-Tabelle: F und $\bar{F}$ können an derselben Stelle gespeichert werden.

2.3.8 Nutzen der Computed-Tabelle

Nehmen wir an, es steht ein unbegrenzter Speicher zur Verfügung. Dann können in der Computed-Tabelle alle bisher berechneten Ergebnisse gespeichert werden.

- Die Verwendung der Computed-Tabelle ermöglicht eine Ausführung von ite in polynomialer Zeit. Der Speicherplatz ist bei der OBDD-Berechnung die kostbarste Ressource. Deshalb benötigen wir eine speichereffiziente Realisierung der Computed-Tabelle.
- Für jedes Tripel F, G, H existieren andere Tripel $\tilde{F}, \tilde{G}, \tilde{H}$ mit der Eigenschaft, daß $ite(F, G, H) = ite(\tilde{F}, \tilde{G}, \tilde{H})$.

  Ziel: Identifizierung aller Tripel, die bei ite Anwendung das gleiche Ergebnis liefern. Statt mehrerer Einträge benutzen wir nur noch einen Eintrag in der Computed-Tabelle.
- Die vollständige Bestimmung aller solcher Tupel ist zu aufwendig, aber einige Tripel können leicht ermittelt werden:

$$ite(F, G, F) = ite(F, G, 0) = ite(G, F, G) = ite(G, F, 0) = F \cdot G$$

.

- Prädikate, die leicht bestimmt werden können:
  - zwei Argumente sind gleich (z. B. $ite(F, G, F)$)
  - ein Argument ist konstant (z. B. $ite(F, 0, H)$)
  - ein Argument ist Komplement eines anderen Arguments (z. B. $ite(F, G, \overline{F})$).
- Transformationen, die möglichst viele Argumente durch Konstante ersetzen:
  - $ite(F, F, G) \Rightarrow ite(F, 1, G)$
  - $ite(F, G, F) \Rightarrow ite(F, G, 0)$
  - $ite(F, G, \overline{F}) \Rightarrow ite(F, G, 1)$
  - $ite(F, \overline{F}, G) \Rightarrow ite(F, 0, G)$.
- Transformationen zur Permutation der Argumente:
  - $ite(F, 1, G) \Rightarrow ite(G, 1, F)$
  - $ite(F, G, 0) \Rightarrow ite(G, F, 0)$
  - $ite(F, G, 1) \Rightarrow ite(\bar{G}, \bar{F}, 1)$
  - $ite(F, 0, G) \Rightarrow ite(\bar{G}, 0, \bar{F})$
  - $ite(F, G, \overline{G}) \Rightarrow ite(G, F, \bar{F})$.
- Man kann solche Transformation benutzen, um das Argument mit kleinstem Index bzgl. der gewählten Variablenordnung als erstes Argument zu erhalten. Sei z. B. x_2 Topvariable von F und x_7 Topvariable von G. Dann arbeitet man besser mit $ite(F, G, 1)$ als mit $ite(\overline{G}, \overline{F}, 1)$.
- Weitere Transformationen:

$$\begin{aligned} ite(F, G, H) &= ite(\overline{F}, H, G) \\ &= \overline{ite(F, \overline{G}, \overline{H})} \\ &= \overline{ite(\overline{F}, \overline{H}, \overline{G})}. \end{aligned}$$

  Unter diesen vier äquivalenten Möglichkeiten existiert genau eine, bei der die beiden ersten Argumente unkomplementiert sind.

  Z. B. : F unkomplementiert, G komplementiert $ite(F, G, H) \Rightarrow \overline{ite(F, \overline{G}, \overline{H})}$.
- Die Computed-Tabelle kann aber tatsächlich nur einen beschränkten Speicher nutzen. Eine effiziente Implementierung erreicht man durch eine Hash-basierte Cache Realisierung der Computed-Tabelle. Mit einer Hashfunktion berechnet man die Position in der Computed-Tabelle. Der Cache speichert in jeder Position nur ein Element. Bei einem neuem Eintrag verschwindet der alte dort plazierte Eintrag. Die Verwendung einer guten Hash-Funktion sichert, daß i. a. Elemente nicht sofort überschrieben werden.

2.3.9 Memory Management

OBDDs werden für gewöhnlich durch symbolische Simulation erzeugt. Deshalb werden im Berechnungsverlauf sehr viele OBDDs für innere Gatter berechnet, die ohne weiteres Interesse sind, sobald die OBDDs für die Nachfolgegatter berechnet sind. Bei der Beseitigung dieser „uninteressanten“ OBDDs treten zwei Probleme auf:

- Ein innerer Knoten hat im allgemeinen mehrere Vorgänger (Die Pointer verweisen lediglich auf die beiden Nachfolgeknoten).
- Die OBDD-Knoten werden aus der Unique-Tabelle, der Computed-Tabelle und von anderen Knoten referenziert.

Lösung: Eine *Garbage-Collection* besteht aus dem Sammeln des freizugebenden Speichers, bis der nötige Aufwand zum Gewinn an Speicherplatz in einem gutem Verhältnis steht.

Idee: Einführung eines *Referenzzählers* R für jeden Knoten v. Der Referenzzähler von v zählt die Knoten, die v referenzieren. R wird mit Einführung eines neuen Knotens, der auf v zeigt, erhöht (meist im Laufe der *ite*-Ausführung). R wird gesenkt, falls die Formel, für die v Funktionsknoten ist, freigegeben wird. Erreicht der Zähler 0, so werden rekursiv absteigend die Referenzzähler der Kinder erniedrigt. Knoten, deren Referenzzähler auf 0 steht, heißen *tot*. Sind genügend viele Knoten tot, so wird eine Garbage-Collection gestartet: Es werden Unique-Tabelle und Computed-Tabelle untersucht, und alle Einträge für tote Knoten gestrichen. Alle toten Knoten werden freigegeben. Die Kosten amortisieren sich bei einer relativ hohen Zahl toter Knoten. Zum Zeitpunkt der Garbage-Collection können die Tabellen leicht umstrukturiert werden (Vergrößerung, andere Hash-Funktion). Das Sammeln toter Knoten ohne sofortige Freigabe gibt die Möglichkeit zur Wiederbelebung toter Knoten im Bedarfsfall (*Reclaim-Operation*: erhöht die Referenzzähler von Knoten und deren Kindern).

Speicherlayout:
Gesamtspeicherplatz für einen Knoten : 21 Bytes

- 4 Worte für den Knoten selbst:

  4 Byte T-Kante
  \+ 4 Byte E-Kante
  \+ 4 Byte Next-Pointer (Collision-List)
  \+ 2 Byte Referenzzähler
  \+ 2 Byte Index
  = 16 Byte

- 1/4 des Unique-Tabelle-Eintrags : 1 Byte

Annahme: durchschnittliche Länge der Collision-List beträgt 4.

- 1/4 des Computed-Tabelle-Eintrags : 4 Bytes.

Kapitel 3

Grundlegende OBDD-basierte Algorithmen

3.1 Implikationstest

Berechnungsziel: Wir wollen für zwei Funktionen $f, g \in \mathcal{B}_n$ festellen, ob $f \leq g$ ($\Leftrightarrow f \Rightarrow g$) gilt. Es gilt: $f \leq g \Leftrightarrow \overline{f} + g \equiv 1$.
$\Rightarrow$ Wir berechnen $\overline{f} + g$ mit *ite* und testen, ob das Ergebnis eine Tautologie ist.

Effizienterer Berechnungsweg
Wir nutzen eine spezielle Version des *ite*-Algorithmus:

„*ite-constant*“ **(Pseudocode)**

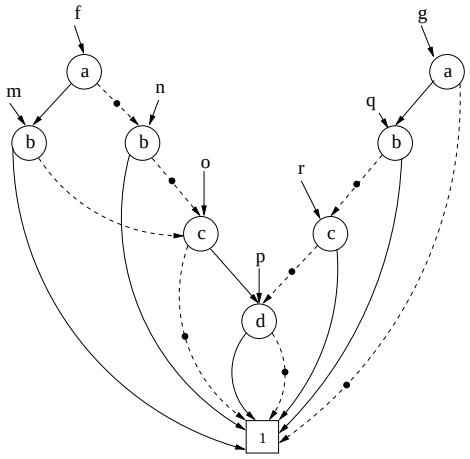
ite-constant(F, G, H)\{
    if (terminalcase) then {
        return result; /* Ergebnis: 0,1 oder nicht konstant*/
    }else if (Computed Tabelle enthält (F, G, H))
        return result;
    }else { Sei v Topvariable von $\{F, G, H\}$;
        $T = ite\text{-}constant(F_v, G_v, H_v)$;
        if ($T \neq 0$ und $T \neq 1$) return nicht-konstant;
        $E = ite\text{-}constant(F_{\overline{v}}, G_{\overline{v}}, H_{\overline{v}})$;
        if ($T \neq E$) return nicht-konstant;
        insert-computed-tabelle($\{F, G, H\}$);
        return T; }\}

$f \Rightarrow g$ kann durch die Berechnung von $ite\text{-}constant(f, g, 1)$ entschieden werden.

Idee: Ist die berechnete Funktion konstant, dann müssen beide Kofaktoren konstant und gleich sein. Trifft eine dieser Bedingung nicht zu, so gibt *ite-constant* „nicht-konstant“ aus. Das frühe Terminieren verhindert den unnötigen Aufbau vieler Zwischenresultate und spart daher Rechenzeit.

Achtung: Nur im Fall $f \not\Rightarrow g$ ergibt sich ein Geschwindigkeitsgewinn (speed-up). Die typische Beschleunigung beträgt etwa Faktor 20!

Beispiel 3.1.1: Seien $f = ab + \overline{b}cd$ und $g = ab + a\overline{c}d$. Frage: Gilt $f \Rightarrow g$?
Antwort: Für die Belegung $w = (a, b, c, d) = 1011$ gilt: $f = 1$ und $g = 0$, also $f \not\Rightarrow g$.
Lösungsweg: $ite\text{-}constant(f, g, 1) \equiv 1$?



$ite\text{-}constant(f, g, 1)$
$= (a, ite\text{-}constant(m, q, 1), ite\text{-}constant(\overline{n}, 0, 1))$
$= (a, (b, ite\text{-}constant(1, 1, 1), ite\text{-}constant(0, \overline{r}, 1)), ite\text{-}constant(\overline{n}, 0, 1))$
$= (a, (b, 1, 1), ite\text{-}constant(\overline{n}, 0, 1))$
$= (a, 1, ite\text{-}constant(\overline{n}, 0, 1))$
$= (a, 1,$nicht-konstant$)$

Bemerkung 3.1.2: Anders als bei *ite* spielt bei *ite-constant* die Sequentialität der Berechnung der Nachfolger eine Rolle ($ite\text{-}constant(\overline{n}, 0, 1)$ wird erst dann berechnet, wenn das Ergebnis aus der anderen Berechnung vorliegt).

3.2 Erfüllbarkeitsproblem

Motivation: Das *Verifikationsproblem* stellt die Frage, ob zwei gegebene Schaltkreise $\mathcal{C}_f$ und $\mathcal{C}_g$ äquivalent sind, d. h. ob $\mathcal{C}_f \equiv \mathcal{C}_g$. Wenn $\mathcal{C}_f \equiv \mathcal{C}_g$ gilt, dann folgt daraus $\mathcal{C}_f \oplus \mathcal{C}_g \equiv 0$. Daher: Teste, ob $\mathcal{C}_f \oplus \mathcal{C}_g$ erfüllbar ist.

Sei $f \in \mathbb{B}_n$ gegeben. Wir sind nun an einer erfüllenden Belegung interessiert, d. h. wir suchen ein $x \in \{0,1\}^n$ mit $f(x) = 1$.

Dieses Problem ist NP-vollständig, falls f als KNF oder als Schaltkreis gegeben ist. Liegt f als OBDD vor, kann das Problem in konstanter Zeit gelöst werden: Es ist lediglich zu testen, ob $P_f \neq 0$-Senke ($\Leftrightarrow f \equiv 0$, d. h. nicht erfüllbar). Analoge Überlegungen können für den Tautologietest gemacht werden.

Verwandte Probleme:

- Finde eine erfüllende Belegung.
- Ermittle die Anzahl der erfüllenden Belegungen.
- Finde eine erfüllende Belegung mit minimalen Kosten („binate covering path"), d. h. der OBDD soll möglichst viele 0-Kanten enthalten.

Idee: Wir nutzen den Zusammenhang zwischen einer erfüllenden Belegung und einem Weg vom Funktionsknoten zur 1-Senke. Dabei müssen wir die möglichen Komplementierungen der Kanten, die sogenannte *Komplementierungsparität*, beachten. Ist die Zahl der komplementierten Kanten eines Weges gerade, so ist die Komplementierungsparität 1, ansonsten 0.

Ein Weg kann von verschiedenen Belegungen genutzt werden, da nicht jede Variable auf dem Weg vorkommen muß. Die Anzahl der Belegungen, die den gleichen Weg nutzen ist 2^r, wobei r die Anzahl der auf diesem Weg nicht vorkommenden Variablen ist.

Die Wege mit Komplementierungsparität 1 definieren die Implikanten der Funktion, bestehend aus allen Literalen, die auf dem Weg komplementiert bzw. unkomplementiert vorkommen.

3.2.1 Finden erfüllender Belegung

Es genügt, einen Weg mit Komplementierungsparität 1 zu finden.

```
OneSat(v[1],p[2],sat[3])
    if (v terminal) return p;
    sat[v→index] = 1;
    if (OneSat(v→T,p,sat)) return 1;
    sat[v→index] = 0;
    if (v→E komplementiert) komplementiere p;
    return OneSat(v→E,p,sat);
```

Analyse: Von jedem nicht terminalen Knoten starten stets Wege mit gerader und ungerader Parität zur Senke. Wäre das nicht der Fall, würde die dargestellte Funktion nicht essentiell vom betrachteten Knoten abhängen und die Existenz würde die Kanonizität des Knotens verletzen. Die Prozedur springt nur dann zurück, falls der erste Sohn terminal und die Komplementierungsparität 0 ist. Letztendlich kommt die Prozedur mit Sicherheit über den zweiten Sohn zum erfolgreichen Abschluß. Die Anzahl der besuchten Knoten ist maximal $2n + 1$, d. h. der Algorithmus hat die Komplexität $O(n)$.

[1] Top-Knoten

[2] Komplementierungsparität (Am Anfang 1 bzw. 0, je nach Komplementierung des Funktionsknotens)

[3] Feld von n Zellen, das anfangs mit „don't care" initialisiert wird und am Ende die Belegungen der Variablen enthält.

3.2.2 Zahl der erfüllenden Belegungen

Wir wollen die Anzahl der Minterme der On-Menge einer Funktion bestimmen. Eine Lösung besteht in einem post-order Traversal des OBDD G. Für jeden Knoten v aus G und jede Kante e wird die Zahl α_v bzw. α_e der erfüllenden Belegungen für die von v bzw. von e repräsentierten Funktion berechnet. Das Ergebnis hängt von der Anzahl der Variablen ab, so hat z. B. $f(x_1, x_2) = x_1 x_2$ eine erfüllende Belegung, $f(x_1, x_2, x_3) = x_1 x_2$ jedoch zwei.

Vereinbarung:

Index des Funktionsknotens: 0.
Index des terminalen Knotens (bei n Variablen): $n + 1$.

Ist v terminal, dann ist $\alpha_v = 2^n$.

Für alle unkomplementierten Kanten $e = (v, w)$ gilt: $\alpha_e = \alpha_w$.
Für alle komplementierten Kanten gilt: $\alpha_e = 2^n - \alpha_w$.

Für alle Knoten v mit den beiden ausgehenden Kanten t und e gilt: $\alpha_v = \frac{\alpha_t}{2} + \frac{\alpha_e}{2}$, da f_T und f_E unabhängig von Index(v) sind und beide Anteile disjunkt sind.

Die Anzahl der erfüllenden Belegungen von f ist α_v, falls v der zugehörige Funktionsknoten ist.

```
SatHowMany(v[4],n[5])
{
      if(v ist terminal) return 2^n;
      if(v ist tabelliert) return Tabellenwert;
      countT = SatHowMany(v→T,n);
      countE = SatHowMany(v→E,n);
      if(v→E ist kompl.) countE = 2^n-countE;
      count = (countT + countE) / 2;
      insert(v,count) in Tabelle;
      return count;
}
```

Komplexitätsanalyse: Ohne Tabellierung hat der Algorithmus eine Komplexität von $O(2^n)$, mit Tabellierung $O(|G|)$ (Unter der Annahme, daß Addition und Multiplikation in der notwendigen Präzision in konstanter Zeit möglich sind).

Bemerkung 3.2.1: Bei Verwendung von 64 Bit Floating Point Arithmetik ist die Berechnung nur bis zu etwa 2000 Variablen hinreichend präzise.

[4] Topknoten
[5] Anzahl der Variablen

3.2.3 Finde billigste erfüllende Belegung

Wir ordnen der Variable x_i die nichtnegativen „Kosten“ c_i zu:

$$x = (x_1, \ldots, x_n);\ c(x) = \sum_{x_i=1}^{n} c_i$$

Die Kosten einer Variablenbelegung sind also die Summe der Kosten aller mit 1 belegten Variablen.

Beispiel 3.2.2: Sei $f(x_1, x_2, x_3) = x_1\overline{x_2} + x_2x_3$. Dann ist $x_1\overline{x_2}x_3$ eine erfüllende Belegung. Die Kosten $c(x_1, \overline{x_2}, x_3) = c_1 + c_3$ sind nicht minimal, da $x_1\overline{x_2}\,\overline{x_3}$ ebenfalls eine erfüllende Belegung mit den Kosten $c_1 \leq c_1 + c_3$ ist.

Sei $f \in \mathbb{B}_n$ als OBDD G gegeben, dann kann die billigste erfüllende Belegung in linearer Zeit $O(|G|)$ gefunden werden (vgl. Shortest-Path-Algorithmus für DAGs mit bounded outdegree 2).

Definition 3.2.3: Sei G OBDD und e Kante von G.

$$length(e) = \begin{cases} 0 \text{ falls } e \text{ else-Kante} \\ c_i \text{ falls } e \text{ then-Kante mit Ursprung in } x_i \end{cases}$$

Die Länge der else-Kante eines Knotens von G ist 0 und die Länge der then-Kante eines Knotens von G mit Variable x_i ist c_i.

Satz 3.2.4 *Sei G OBDD für $f(x_1, \ldots, x_m)$. Dann ist $A = (a_1, \ldots, a_m)$ eine Minimalkostenbelegung für $x_1, \ldots, x_n$ für die gilt $f(a_1, \ldots, a_m) = 1$ und A ist gegeben durch den kürzesten Pfad gerader Komplementierungparität von der Wurzel von G zum terminalen Knoten.*

Beweis: Jeder Pfad gerader Parität von der Quelle zum terminalen Knoten repräsentiert eine Menge erfüllender Belegungen. Bewerten wir alle Variablen, die in diesem Pfad nicht auftauchen, mit 0, so erhalten wir eine Variablenbelegung minimaler Kosten für diesen Pfad (Es kann mehrere Variablenbelegungen mit minimlen Kosten geben). Sei A eine kostenminimale Zuweisung, die mit dem kürzesten Pfad p in G assoziiert ist.

Zu zeigen: A ist minimal.

Annahme: Es existiert ein a' mit $f(a') = 1$ und $c(a') < c(a_p)$.
$\Rightarrow a'$ korrespondiert mit einem Weg p', der von der Quelle zur Senke führt.
$\Rightarrow c(a') \geq length(p') \geq length(p) = c(a_p)$
Widerspruch! □

3.3 Komposition von Funktionen

Berechnungsziel:
$f|_{x_i=g} = f(x_1, \ldots, x_{i-1}, g, x_{i+1}, \ldots, x_n) \quad f, g \in \mathbb{B}_n$
Wir benutzen die Shannon-Entwicklung $f = x_i f_{x_i} + \overline{x_i} f_{\overline{x_i}}$:

$\Rightarrow f|_{x_i=g} = g \cdot f_{x_i} + \overline{g} \cdot f_{\overline{x_i}}$
$\Rightarrow f|_{x_i=g} = ite(g, f_{x_i}, f_{\overline{x_i}})$
(Die Kofaktoren f_{x_i} und $f_{\overline{x_i}}$ können später berechnet werden.)

Effizienter Berechungsweg
Idee: Wir bilden Kofaktoren bzgl. der Topvariable (einfach zu berechnen).
Sei t der Index der Topvariable von f:

1.Fall $t > i$**:** f hängt nicht von x_i ab $\Rightarrow f|_{x_i=g} = f$.

2.Fall $t = i$**:** x_i ist die Topvariable von f, Kofaktoren können direkt ausgegeben werden.

3.Fall $t < i$**:** Wir entwickeln bzgl. x_t:

$$\begin{aligned} f|_{x_i=g} &= x_t(gf_{x_i} + \overline{g}f_{\overline{x_i}})_{x_t} + \overline{x_t}(gf_{x_i} + \overline{g}f_{\overline{x_i}})_{\overline{x_t}} \\ &= x_t \cdot (g_{x_t} \cdot (f_{x_t})_{x_i} + \overline{g_{x_t}}(f_{x_t})_{\overline{x_i}}) + \overline{x_t} \cdot (g_{\overline{x_t}} \cdot (f_{\overline{x_t}})_{x_i} + \overline{g_{\overline{x_t}}}(f_{\overline{x_t}})_{\overline{x_i}}) \end{aligned}$$

(Kommutativität der Kofaktorenbildung).

Die rekursive Anwendung bis $t \geq i$ liefert das gewünschte Ergebnis durch die Berechnung der Kofaktoren von Topvariablen.

3.4 Quantifizierung

Definition 3.4.1: Sei $f \in \mathbb{B}_n$.

$\mathbb{B}_{n-1} \ni (\exists_{x_i} f) := f_{x_i} + f_{\overline{x_i}}$ heißt *existentielle Quantifizierung von f bzgl. x_i*,
$\mathbb{B}_{n-1} \ni (\forall_{x_i} f) := f_{x_i} \cdot f_{\overline{x_i}}$ heißt *universelle Quantifizierung von f bzgl. x_i*.

$\exists f := \exists_{x_1} \ldots \exists_{x_n} f$
$\forall f := \forall_{x_1} \ldots \forall_{x_n} f$

Es gilt: $(\exists_{x_i} f)(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n) = 1 \Leftrightarrow \exists x_i : f(x_1, \ldots, x_n) = 1$
$(\forall_{x_i} f)(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n) = 1 \Leftrightarrow \forall x_i : f(x_1, \ldots, x_n) = 1.$

Beispiel 3.4.2: $f = \bar{x}\bar{y}z + x\bar{z} + xy$
Wir wollen nun f nach z quantifizieren. Es ist $f_z = \bar{x}\bar{y} + xy$ und $f_{\bar{z}} = x + xy = x$.
Daher gilt: $\exists_z f = x + \bar{x}\bar{y} + xy = x + \bar{x}\bar{y} = x + \bar{y}$
$\forall_z f = x(\bar{x}\bar{y} + xy) = xy.$

Betrachte $\exists_{x_i} f, \forall_{x_i} f$ als Funktionen von $\mathbb{B}_n$. Dann gilt: $\forall_{x_i} f \leq f \leq \exists_{x_i} f$. Genauer:

$\forall_{x_i} f$: größte von x_i unabhängige Funktion in f.
$\exists_{x_i} f$: kleinste von x_i unabhängige Funktion, die f enthält.

Die Quantifizierung ist monoton: $f \leq g \Rightarrow \exists_{x_i} f \leq \exists_{x_i} g$ und $\forall_{x_i} f \leq \forall x_i g$.

Beweis:

$$\left.\begin{aligned} f &= xf_x + \overline{x}f_{\overline{x}} \\ g &= xg_x + \overline{x}g_{\overline{x}} \end{aligned}\right\} \; f \leq g \;\Rightarrow\; f_x \leq g_x \cdot f_{\bar{x}} \leq g_{\bar{x}}$$

□

Eigenschaften:

- Quantifizierungen des selben Typs sind kommutativ:
  $\exists_{x_1}\exists_{x_2} f := \exists_{x_2}(\exists_{x_1} f) = \exists_{x_1}(\exists_{x_2} f)$
  $\forall_{x_1}\forall_{x_2} f := \forall_{x_2}(\forall_{x_1} f) = \forall_{x_1}(\forall_{x_2} f)$
  Aber: Quantifizierungen unterschiedlichen Typs sind nicht kommutativ!

- Distributivität:
  $\exists_x(f+g) := \exists_x f + \exists_x g$
  $\forall_x(f \cdot g) := \forall_x f \cdot \forall_x g$

- $\forall_x(f+g) \geq \forall_x f + \forall_x g$
  $\exists_x(f \cdot g) \leq \exists_x f \cdot \exists_x g$

- Komplementierung:
  $\exists_x(\overline{f}) := \overline{(\forall_x f)}$
  $\forall_x(\overline{f}) := \overline{(\exists_x f)}$

- Ein rekursiver Algorithmus zur Quantifizierung kann genauso abgeleitet werden, wie bei der Kofaktorenbildung. Es gilt: $\exists_c f = x\exists_c f_x + \overline{x}\exists_c f_{\overline{x}}$.

  Beweis: $\exists_c f = \exists_c(x f_x + \overline{x} f_{\overline{x}}) = \exists_c(x f_x) + \exists_c(\overline{x} f_{\overline{x}}) = x\exists_c f_x + \overline{x}\exists_c f_{\overline{x}}$ □

Beispiel 3.4.3: Berechne $\exists_z f$ für $f = \bar{x}\bar{y}z + x\bar{z} + xy$ mit $x \leq y \leq z$:

$$\begin{aligned}
\exists_z f &= x \cdot \exists_z f_x + \bar{x} \cdot \exists_z f_{\bar{x}} \\
&= x \cdot (y \cdot \exists_z f_{xy} + \bar{y}\exists_z f_{x\bar{y}}) + \bar{x} \cdot (y\exists_z f_{\bar{x}y} + \bar{y}\exists_z f_{\bar{x}\bar{y}}) \\
&= x \cdot (y\exists_z 1 + \bar{y}\exists_z \bar{z}) + \bar{x}(y\exists_z 0 + \bar{y}\exists_z z) \\
&= x \cdot 1 + \bar{x} \cdot \bar{y} \;=\; x + \bar{y}.
\end{aligned}$$

Bemerkung 3.4.4: Für eine als DNF gegebene Funktion ist die existentielle Quantifizierung einfach: Man entfernt einfach die quantifizierte Variable aus dem Term. Das gleiche gilt für die universelle Quantifizierung, wenn die Funktion als KNF gegeben ist.

3.5 Berechnung verallgemeinerter Kofaktoren

3.5.1 Orthogonale Entwicklung von Schaltfunktionen

Einfache Kofaktoren werden gemäß der Boole-Shannon-Entwicklung gebildet:

$$f = x_i f|_{x_i=1} + \overline{x_i} f|_{x_i=0}.$$

Die Erweiterung auf Monome basiert auf der Kommutativität der Kofaktoren:

$$(f_x)_y = (f_y)_x = f_{xy},$$

kann also als Sequenz von Kofaktoren für einfache Variablen betrachtet werden. Andererseits ist ein Monom auch eine Funktion.

Frage: Kann ein Kofaktor auch bzgl. einer beliebigen Funktion definiert werden?

Antwort: Ja. Die generalisierte Form des Kofaktors hat große Bedeutung bei der Verifikation endlicher Automaten.

Wir wollen die Shannon-Entwicklung bzgl. orthonormaler Mengen verallgemeinern.

Definition 3.5.1: Eine Menge $\{b_1, \ldots, b_r\}, b_i \in \mathbb{B}_n$ heißt *orthonormal*, falls

1. $\sum_{i=1}^{r} b_i = 1$ und
2. $b_i b_j = 0 \ \forall i \neq j$.

Beispiel 3.5.2: $\{x_i, \overline{x_i}\}$ ist eine orthonormale Menge und wird in der Shannon-Entwicklung verwendet. Ebenso ist für alle $g \in \mathbb{B}^n$ die Menge $\{g, \overline{g}\}$ orthonormal. Wir können $f \in \mathbb{B}_n$ bezüglich einer beliebigen orthonormalen Menge $(b_1, \ldots, b_r)$ entwicklen:

$$f(x_1, \ldots, x_n) = \sum_{j=1}^{r} f_j(x_1, \ldots, x_n) \cdot b_j(x_1, \ldots, x_n).$$

Satz 3.5.3 *Sei $\{b_1, \ldots, b_r\}$ eine orthonormale Menge und $b_j, f \in \mathbb{B}_n$, so gilt:*

$$f = \sum_{j=1}^{r} f_j \cdot b_j \Leftrightarrow f \cdot b_j = f_j \cdot b_j, \quad \forall j.$$

Beweis:
„$\Rightarrow$“

$$f = \sum_{i=1}^{r} f_i b_i \ \Rightarrow \ f b_j = (\sum_{i=1}^{r} f_i b_i) b_j = \sum_{i=1}^{r} f_i b_i b_j = f_j b_j \ (\text{da } b_i b_j = 0 \ \forall i \neq j).$$

„$\Leftarrow$“ Gelte

$$f_j b_j = f b_j \quad \forall j \ \Rightarrow \ \sum_{i=1}^{r} f_i b_i = \sum_{i=1}^{r} f b_i = f \sum_{i=1}^{r} b_i = f.$$

□

Bemerkung 3.5.4: Die Koeffizienten sind durch Satz 3.5.3 zwar nicht eindeutig bestimmt, aber es werden Restriktionen bzgl. ihres Verhaltens angegeben. Sie sind also nur unvollständig spezifiziert. Man kann zeigen, daß jede Funktion $h \in [f b_j, f + \overline{b_j}]$ die in Satz 3.5.3 aufgestellten Bedingungen erfüllt:

$h_j = f b_j \Rightarrow h_j b_j = f b_j b_j = f b_j$ und
$h_j = f + \overline{b_j} \Rightarrow h_j b_j = (f + \overline{b_j}) b_j = f b_j + b_j \overline{b_j} = f b_j$

Interpretation: f_j stimmt mit f überall dort überein, wo $b_j = 1$ ist, ansonsten ist es „don't care“.

Übung: Shannon-Entwicklung mit verallgemeinerten Kofaktoren!

Das Ergebnis kann ausgenutzt werden, um kompakte Repräsentationen von Koeffizienten zu erhalten. Wir zeigen nun, daß viele Eigenschaften einfacher Kofaktoren auch für verallgemeinerte Kofaktoren gelten.

Satz 3.5.5 *Sei* $\{b_1, \ldots, b_r\}$ *orthonormale Menge und seien* $f = \sum_{j=1}^{r} f_j b_j$ *und* $g = \sum_{j=1}^{r} g_j b_j$ *Boolesche Funktionen. Dann gilt:*

1. $f + g = \sum_{j=1}^{r} (f_j + g_j) b_j$
2. $fg = \sum_{j=1}^{r} (f_j g_j) b_j$
3. $\overline{f} = \sum_{j=1}^{r} \overline{f_j} b_j$.

Beweis:

1. Trivial, folgt aus der Distributivität.
2. Gilt wegen $b_i b_j = 0 \Leftrightarrow (i \neq j)$.
3. Definition des Komplements: $f \cdot \overline{f} = 0$ und $f + \overline{f} = 1$. Einsetzen:

$$f \cdot \overline{f} = (\sum_{i=1}^{r} f_i b_i) \cdot (\sum_{i=1}^{r} \overline{f_i} b_i) = \sum_{i=1}^{r} (f_i \overline{f_i}) b_i = 0$$

$$f + \overline{f} = (\sum_{i=1}^{r} f_i b_i) + (\sum_{i=1}^{r} \overline{f_i} b_i) = \sum_{i=1}^{r} (f_i + \overline{f_i}) b_i = 1.$$

□

3.5.2 Constrain Operator

Wir können jetzt den Kofaktor einer Funktion bzgl. einer anderen Funktion als orthonormale Erweiterung angeben:

Definition 3.5.6: Seien $f, g \in \mathbb{B}_n$ und $f = g f_g + \overline{g} f_{\overline{g}}$ sei Entwicklung von f bzgl. $\{g, \overline{g}\}$. f_g heißt *positiver verallgemeinerter Kofaktor* und $f_{\overline{g}}$ heißt *negativer verallgemeinerter Kofaktor*. Verallgemeinerte Kofaktoren sind nicht eindeutig bestimmt! f_g ist fixiert, falls $g = 1$, ansonsten ist es frei wählbar.

Da Kofaktoren nur unvollständig (nicht eindeutig) spezifiziert sind, stellt sich das Problem, den besten Kofaktor bzgl. eines bestimmten Kriteriums zu finden. Als Kriterium bietet sich die Größe der Repräsentationsform an, für OBDDs also die Anzahl der Knoten. Die Wertemenge von f_g ist überall dort fixiert, wo $g = 1$ ist, aber wie wählen wir die Werte von f_g für $g = 0$?

Idee: (Coudert, Berthert, Madre '89) Einführung „Constrain"-Operators der kompakt darstellbare verallgemeinerte Kofaktoren liefert. Strategie:

- Bilde jeden Minterm der off-Menge von g auf einen Minterm von g ab.
- Benutze diese Abbildung, um f_g an den Stellen $g = 0$ zu definieren.
- Die Abbildung von g ist abhängig von g und folgender Abstandsdefinition:

Definition 3.5.7: Seien $x_1, \ldots, x_n$ Boolesche Variablen mit der Ordnung $x_1 \leq x_2 \leq \ldots \leq x_n$ und seien $r = (r_1, \ldots, r_n)$ sowie $s = (s_1, \ldots, s_n)$ zwei Minterme. Der Abstand zwischen r und s ist definiert durch:

$$\| r - s \| = \sum_{i=1}^{n} |r_i - s_i| 2^{n-i}.$$

Die Definition bezieht sich auf den Unterschied der Pfade, die durch die beiden Minterme beschrieben werden innerhalb eines OBDD der Ordnung $x_1 \leq \ldots \leq x_n$.

Definition 3.5.8: Seien $f, g \in \mathbb{B}_n$. $f \downarrow g$ (*„f constrained by g"*) ist folgendermaßen definiert:

$$f \downarrow g(r) = \begin{cases} f(r) \text{ falls } g(r) = 1 \\ f(s) \text{ falls } g(r) = 0 \text{ und } g(s) = 1 \text{ und } \|r - s\| \text{ minimal} \\ 0 \text{ sonst} \end{cases}$$

$f \downarrow g$ ist verallgemeinerter Kofaktor.

Bemerkung 3.5.9: $s \neq t \Rightarrow \| r - s \| \neq \| r - t \|$. Daher ist s in Definition 3.5.7 eindeutig bestimmt.

Der Algorithmus zur Berechnung des Constrain Operators $f \downarrow g$ basiert auf rekursiver Anwendung auf die BDDs von f und g, sowie auf folgendem Satz.

Satz 3.5.10 *Seien $f, g, h \in \mathbb{B}_n$ und sei $\hat{f}_g = h(f_h)_{g_h} + \bar{h}(f_{\bar{h}})_{g_{\bar{h}}}$ und $\hat{f}_{\bar{g}} = h(f_h)_{\bar{g}_h} + \bar{h}(f_{\bar{h}})_{\bar{g}_{\bar{h}}}$. Dann gilt:*

$$f = g \cdot \hat{f}_g + \bar{g} \cdot \hat{f}_{\bar{g}}.$$

Beweis: Wir müssen zeigen, daß $\hat{f}_g$ und $\hat{f}_{\bar{g}}$ Satz 3.5.3 erfüllen. Betrachte $\hat{f}_g$:

$$\begin{aligned} \hat{f}_g \cdot g &= g \cdot h \cdot (f_h)_{g_h} + g \cdot \bar{h}(f_{\bar{h}})_{g_{\bar{h}}} \\ &= h \cdot g_h (f_h)_{g_h} + \bar{h} \cdot g_{\bar{h}} (f_{\bar{h}})_{g_{\bar{h}}} \\ &= h \cdot g_h \cdot f_h + \bar{h} \cdot g_{\bar{h}} \cdot f_{\bar{h}} \\ &= h \cdot g_h \cdot f + \bar{h} \cdot g_{\bar{h}} \cdot f_h \\ &= f(h_{g_h} + \bar{h}_{g_{\bar{h}}}) = f \cdot g \end{aligned}$$

$\hat{f}_{\bar{g}}$ analog. Hierbei wurde ausgenutzt, daß $g \cdot h = g_h \cdot h$ und $g_h \cdot f_h = g_h(f_h)_{g_h}$. □

Achtung: Die näherliegende Entwicklung $f_g = h(f_g)_h + \bar{h}(f_g)_{\bar{h}}$ ist nicht für eine Rekursion geeignet. Es gilt nicht $(f_g)_h = (f_h)_{g_h}$, da die Funktionen unvollständig spezifiziert sind!

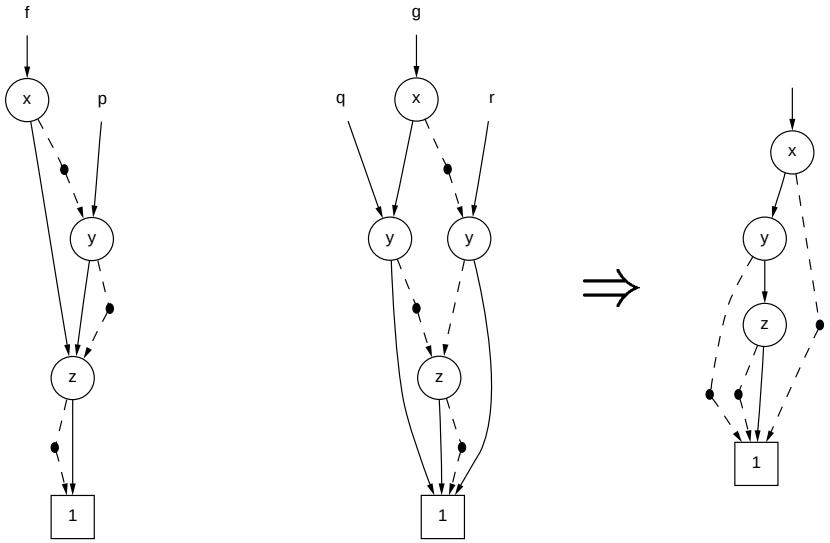
Algorithmus:

constrain(f, g)
{

if$(g = 1$ or $f = 0)$ return f;	$(g = 1$ oder $f\, const \Rightarrow f \downarrow g = f)$
else if$(f = g)$ return 1;	$(f = g \Rightarrow (\forall s\, g(s) = 1 \Rightarrow f(s) = 1) \Rightarrow f \downarrow g = 1)$
else if$(f = \overline{g})$ return 0;	$(f = \bar{g} \Rightarrow f \downarrow g = 0$ analog$)$
else if$(g = 0)$ return 0;	$(g = 0 \Rightarrow f \downarrow g = 0$ (wahlfrei))
Sei x Topvariable in $\{f, g\}$	$(*)$
else if $(g_x = 0)$ return constrain$(f_{\overline{x}}, g_{\overline{x}})$	
else if $(g_{\bar{x}} = 0)$ return constrain(f_x, g_x)	
else	
return ite$(x$,constrain(f_x, g_x),constrain$(f_{\overline{x}}, g_{\overline{x}}))$; }	

$(*)$ $g_x = 0$: Sei c Monom bereits festgelegter Variablen.Gesucht: Minterm s von g; $g(s) = 1$ mit $\| s = c\bar{x} \|$ minimal. Sei $r = c\bar{x}t$ und $\hat{r} = cxt$. Ist nun $g(\hat{r}) = g(cxt) = 1$, dann ist $s = \hat{r}$, sonst ist s zu $\hat{r}$ nächstgelegener Minterm von g.

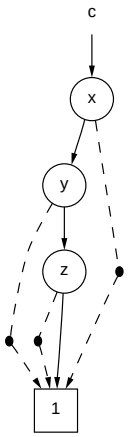
Beispiel 3.5.11:



$$
\begin{aligned}
f\downarrow g &= (x,\, f_x\downarrow g_x,\, f_{\bar{x}}\downarrow g_{\bar{x}}) \\
&= (x,\, z\downarrow g_x,\, \bar{p}\downarrow\bar{r}) \\
&= (x,\, z\downarrow q,\, (y,\, p_{\bar{y}}\downarrow r_{\bar{y}},\, p_{\bar{y}}\downarrow r_{\bar{y}})) \\
&= (x,\, (y,\, z\downarrow 1,\, z\downarrow\bar{z}),\, (y,\, z\downarrow 0,\, \bar{z}\downarrow z)) \\
&= (x,\, (y,\, z,\, 0),\, (y,\, 0,\, 0)) \\
&= (x,\, (y,\, z,\, 0),\, 0)
\end{aligned}
$$

Bemerkung 3.5.12:

- Eine Performanceverbesserung kann durch die Verwendung einer Computed-Tabelle als Cache wie bei der *ite*-Operation erreicht werden. Als Schlüssel werden die beiden Argumente (f, g) verwendet.
- constrain(f, c), wobei c Monom ist, liefert gewöhnliche Kofaktoren von f bzgl. c. Beispiel: $c = xyz$



- Im Allgemeinen ist $f\downarrow g$ kleiner als f, wenn aber g sehr groß ist, dann kann $f\downarrow g$ auch größer sein als f, z. B. wenn g von vielen Variablen abhängt, von denen f nicht abhängt. Diese müssen dann in $f\downarrow g$ eingeführt werden.

  Ausweg: Ausquantifizieren der Variablen aus g, die in f nicht vorkommen.

3.5.3 Restrict Operator

Wir benötigen zunächst folgendes Lemma:

Lemma 3.5.13: Seien $f, g, h \in \mathbb{B}_n$ gegeben mit $h \geq g$, dann gilt: $gf_g = gf_h$.

Beweis: Klar wegen $g = 1 \Rightarrow h = 1$ (und $g = 1 \Rightarrow f_g = f \Rightarrow f_h = f$). Der Kofaktor bzgl. g ist gleich dem Kofaktor bzgl. h. □

Speziell gilt Lemma 3.5.13 auch für den Constrain Operator und für $h = \exists_{x_i} g$. Wir können also alle Variablen aus g, die in f nicht vorkommen, existentiell quantifizieren, bevor wir $f \downarrow g$ berechnen. Wenn h das Ergebnis der Quantifizierungen ist, dann ist $f \downarrow h$ ein gültiger Kofaktor von f bzgl. g. Wir integrieren also die Quantifizierung mit in die Constrain Berechnung und erhalten:

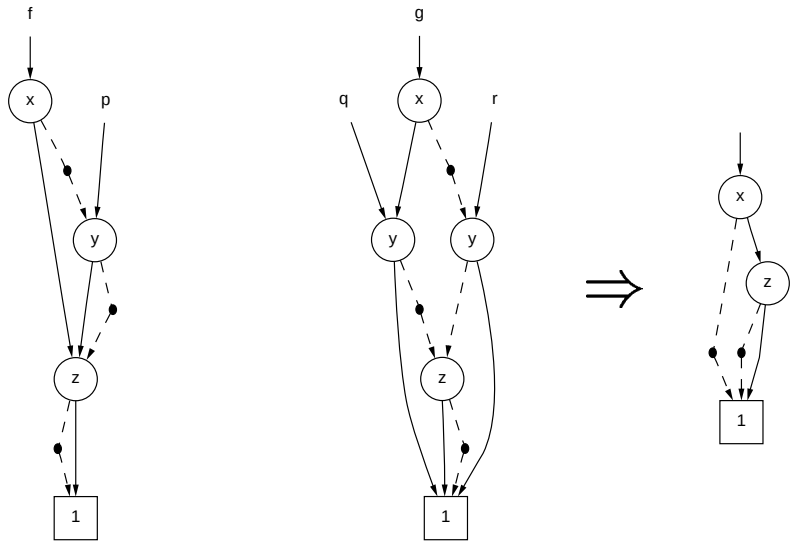
Bezeichnung: $f \Downarrow g$ „*Restrict-Operator*“

Algorithmus: Immer wenn die Topvariable x_t von g einen niedrigeren Index hat als die Topvariable von f, gibt die Prozedur $f \Downarrow (\exists_{x_t} g)$ zurück.

restrict(f, g)\{
    if$(g = 1$ or $f\, const)$ return f;
    else if$(f = g)$ return 1;
    else if$(f = \overline{g})$ return 0;
    else if$(g = 0)$ return 0;
    Sei x Topvariable von $\{f, g\}$
    else if$(g_x = 0)$ return restrict$(f_{\overline{x}}, g_{\overline{x}})$
    else if$(g_{\overline{x}} = 0)$ return restrict(f_x, g_x)
    else if(x ist nicht Topvariable von f)
        return restrict(f,ite$(g_x, 1, g_{\overline{x}})$); (*existenzielle Quantifizierung*)
    else return ite(x,restrict(f_x, g_x),restrict$(f_{\overline{x}}, g_{\overline{x}})$);
\}

Bemerkung 3.5.14: Restrict kann kleinere OBDDs produzieren als „erst Quantifizieren, dann Constrain“. Außerdem wird bei Restrict die Quantifizierung von Variablen möglich, die nur in bestimmten Kofaktoren von f nicht vorkommen. Da aber in jedem Rekursionsschritt die Durchführung der Quantifizierung erfolgen kann, ist Restrict langsamer. Wegen der existentiellen Quantifizierung ist Restrict nicht distributiv bzgl. der Produktbildung.

Beispiel 3.5.15: Vergleiche Beispiel 3.5.11.



$$\begin{aligned} f \Downarrow g &= (x,\, f_x \Downarrow g_x),\, (f_{\bar{x}} \Downarrow g_{\bar{x}}) \\ &= (x,\, z \Downarrow q),\, (\bar{p} \Downarrow \bar{r})\ ^{(*)} \\ &= (x,\, z \Downarrow (\exists_y q),\, 0) \\ &= (x,\, z \Downarrow 1,\, 0) \\ &= (x,\, z,\, 0) \,=\, x \cdot z \end{aligned}$$

(*) $\bar{p} \Downarrow \bar{r} = 0$ wie bei Constrain, da $\bar{p}$ und $\bar{r}$ von den gleichen Variablen abhängig sind, d. h. es gibt in r keine Variablen, die in p nicht vorkommen

$f \Downarrow g$ hat weniger Knoten als $f \downarrow g$, da wir es vermeiden konnten, einen Knoten y in den positiven Kofaktor bzgl. x einzuführen.

3.6 Mengenoperationen und Relationen

Definition 3.6.1: Sei $\mathbb{B} = \{0,1\}$, $S \subseteq \mathbb{B}^n$ eine Punktmenge im Booleschen Raum $\mathbb{B}^n$. Wir definieren $\chi_S : \mathbb{B}^n \to \mathbb{B}$ als *charakteristische Funktion* von S mit:

$$\forall x \in \mathbb{B}^n : x \in S \;\Leftrightarrow\; \chi_S(x) = 1.$$

Wir erweitern die Definition der charakteristischen Funkton auf Teilmengen beliebiger endlicher Mengen Q durch Einführung einer *injektiven Abbildung* ϵ von Q nach $\mathbb{B}^n$ (binäre Kodierung):

$$\epsilon : Q \to \mathbb{B}^n,\ n \geq \log_2 |Q| \ \text{(da injektiv)}.$$

Die *Bildmenge* von $P \subseteq Q$ gemäß ϵ ist $A \subseteq \mathbb{B}^n$:

$$A = \{a \in \mathbb{B}^n : \exists p \in P, a = \epsilon(p)\}.$$

Wir definieren die charakteristische Funktion von P gemäß ϵ als charakteristische Funktion von A:

$$\chi_P^{\epsilon} = \chi_A$$

Wir können jetzt jeder endlichen Menge eine charakteristische Funktion zuordnen. Alle mengentheoretischen Operationen können direkt auf charakteristischen Funktionen ausgeführt werden.

Sei $S_1, S_2 \subseteq \mathbb{B}^n$:

$$\begin{aligned} \chi_{S_1 \cup S_2} &= \chi_{S_1} + \chi_{S_2} \\ \chi_{S_1 \cap S_2} &= \chi_{S_1} \cdot \chi_{S_2} \\ \chi_{\overline{S_1}} &= \overline{\chi_{S_1}}. \end{aligned}$$

Für beliebige endliche Mengen $A_1, A_2 \subseteq Q$ und die Kodierung $\epsilon : Q \to \mathbb{B}^n$:

$$\begin{aligned} \chi_{A_1 \cup A_2} &= \chi_{A_1} + \chi_{A_2} \\ \chi_{A_1 \cap A_2} &= \chi_{A_1} \cdot \chi_{A_2} \\ \chi_{\overline{A_1}} &= \overline{\chi_{A_1}} \cdot \chi_Q \;\hat{=}\; \chi_{\overline{A_1} \cap Q}. \end{aligned}$$

Charakteristische Funktionen stellen eine kompakte Darstellung von endlichen Mengen dar und lassen die effiziente Ausführung von mengentheoretischen Funktionen zu.

Beispiel 3.6.2: Charakteristische Funktion

$$\chi_S(x_1, \ldots, x_{500}) = x_{100} + x_{200} + x_{300} + x_{400} + x_{500}.$$

Wir können zeigen, daß $|S| = 15 \cdot 2^{496}$ ist. Die OBDD-Darstellung der charakteristischen Funktion hat jedoch nur 5 Knoten. Die Speicherung als Bitvektor o. ä. würde wesentlich mehr Speicherplatz benötigen. Mengenoperationen lassen sich also leichter auf den OBDDs der charakteristischen Funktion ausführen, als auf der eigentlichen Menge.

Definition 3.6.3: Wir betrachten jetzt Relationen $R \subseteq Q_1 \times Q_2$:

$$\chi_R(x_1, \ldots, x_n, y_1, \ldots, y_m) \leq \chi_{Q_1}(x_1, \ldots, x_n) \cdot \chi_{Q_2}(y_1, \ldots, y_m).$$

Diese Definition kann auf n-stellige Relationen verallgemeinert werden.

Beispiel 3.6.4: $Q_1 = \{2, 4, 7\}$, $Q_2 = \{1, 4, 5, 6\}$
$R = \{(q_1, q_2) \in Q_1 \times Q_2 : q_1 > q_2\} = \{(2,1), (4,1), (7,1)\,(7,4), (7,5), (7,6)\}$
Kodierung: $\epsilon_1 : Q_1 \to \mathbb{B}^2$; $\epsilon_2 : Q_2 \to \mathbb{B}^2$

$$\begin{array}{lll} \epsilon_1(2) = & \epsilon_2(1) = 00 \\ \epsilon_1(4) = & \epsilon_2(4) = 01 \\ \epsilon_1(7) = & \epsilon_2(5) = 10 \\ & \epsilon_2(6) = 11 \end{array}$$

Dann folgt: $\chi_{Q_1}(x_1, x_2) = \overline{x_1} + \overline{x_2}$, $\chi_{Q_2}(y_1, y_2) = 1$

$$\chi_R(x_1, x_2, y_1, y_2) = \overline{x_1 x_2 y_1 y_2} + \overline{x_1} x_2 \overline{y_1 y_2} + x_1 \overline{x_2 y_1 y_2} + x_1 \overline{x_2 y_1} y_2 \quad +$$

$$x_1\,\overline{x_2}\,y_1\,\overline{y_2} + x_1\,\overline{x_2}\,y_1\,y_2 = \overline{x_1}\,\overline{y_1}\,\overline{y_2} + x_1\,\overline{x_2}$$

Frage: Wenn $R \subseteq Q_1 \times Q_2$ gegeben ist, welche Elemente aus Q_1 sind in Relationen mit einem Element q aus Q_2?

$$P = \{q_1 \in Q : (q_1, q) \in R\}$$

Gegeben: χ_r, $\chi_{\{q\}} = \hat{y}_1 \hat{y}_2 \cdots \hat{y}_m$, $\hat{y}_i \in \{y_i, \overline{y_i}\}$
$\Rightarrow \chi_P(x_1, \ldots, x_n) = \exists y_1, \ldots, y_m(\chi_R(x_1, \ldots, x_n, y_1, \ldots, y_m) \cdot \hat{y}_1 \hat{y}_2 \cdots \hat{y}_m)$

Beispiel 3.6.5: (Fortsetzung)
Welche Elemente aus Q_1 stehen in Relation mit dem Element $1 \in Q_2$?

$$\chi_{P_1}(x_1, x_2) = \exists y_1, y_2(x_1\overline{x_2} + \overline{x_1}\,\overline{y_1}\,\overline{y_2}) \cdot \overline{y_1}\,\overline{y_2} = x_1\overline{x_2} + \overline{x_1} = \overline{x_2} + \overline{x_1}$$

d. h. $P_1 = \{7, 4, 2\}$. Tatsächlich gilt: $7, 4, 2 > 1$.

Frage: Welche Elemente aus Q_1 stehen mit mind. einem Element aus Q_2 in Relation?

$$Def(R) = \{q_1 \in Q_1 : \exists q_2 \in Q_2 : (q_1, q_2) \in R)\}$$

$\Rightarrow \chi_{Def(R)}(x_1, \ldots, x_n) = \exists y_1, \ldots, y_m(\chi_R(x_1, \ldots, x_n, y_1, \ldots, y_m))$

Beispiel 3.6.6: (Fortsetzung)
Welche Elemente aus Q_1 stehen mit mindestens einem Element aus Q_2 in Relation?

$$\chi_{Def(R)} = \exists y_1 y_2(x_1\,\overline{x_2} + \overline{x_1}\,\overline{y_1}\,\overline{y_2}) = x_1\,\overline{x_2} + \overline{x_1} = \overline{x_1} + \overline{x_2} = \chi_{Q_1}$$

Frage (Verallgemeinerung): Welche Elemente aus Q_1 stehen mit mindestens einem Element aus $P_2 \subseteq Q_2$ in Relation? (Vorher: $P_2 = \{q\}$ oder $P_2 = Q_2$)

$$P_1 = \{q_1 \in Q_1 : \exists q_2 \in P_2 : (q_1, q_2) \in R\}$$

Mit χ_R und χ_{P_2} gegeben:

$$\chi_{P_1}(x_1, \ldots, x_n) = \exists y_1 \cdots y_m(\chi_R(x_1, \ldots, x_n, y_1, \ldots, y_m) \cdot \chi_{P_2}(y_1, \ldots, y_m))$$

Beispiel 3.6.7: (Fortsetzung)
Welche Elemente aus Q_1 stehen in Relation mit den geradzahligen Elementen aus Q_2?

$$\exists y_1 y_2((\overline{x_1}\,\overline{y_1}\,\overline{y_2} + x_1\,\overline{x_2}) \cdot y_2) = x_1\,\overline{x_2} = \chi_{\{7\}}$$

Bemerkung 3.6.8: Wenn $|P_2| = 1$, kann χ_{P_1} berechnet werden durch Kofaktorbildung von χ_R bzgl. χ_{P_2}.

Es kann auch das Bild von R

$$Im(R) = \{q_2 \in M_2 : \exists m_1 \in M_1 : (m_1, m_2) \in R\}$$

berechnet werden durch:

$$\chi_{Im(R)}(y_1, \ldots, y_m) = \exists x_1 \cdots x_n \chi_R(x_1, \ldots, x_n, y_1, \ldots, y_m)$$

Beispiel 3.6.9: (Fortsetzung)

$$\chi_{Im(R)}(y_1, y_2) = \exists x_1 x_2\,(\overline{x_1}\,\overline{y_1}\,\overline{y_2} + x_1\,\overline{x_2}) = \exists x_2\,(\overline{y_1}\,\overline{y_2} + \overline{x_2}) = 1$$

Tatsächlich besitzt jedes Element aus M_2 ein Urbild in M_1.

Kapitel 4

Optimierung der Variablenordnung

4.1 Einführung

Wir wissen bereits:

- Für eine vorgegebene Variablenordnung ist die (reduzierte) OBDD-Darstellung einer Schaltfunktion eindeutig bestimmt (Kanonizität).
- Für verschiedene Variablenordnungen können OBDD-Darstellungen für eine Schaltfunktion verschiedene Größen haben.

Die Problematik der Variablenordnung wird an folgenden Beispielen gut sichtbar:

Beispiel 4.1.1: Es gibt Funktionen, die für jede Variablenordnung lineare OBDD-Größe besitzen, z. B. $OR(x_1, \ldots, x_n)$, $AND(x_1, \ldots, x_n)$, $PAR(x_1, \ldots, x_n)$.

Beispiel 4.1.2: Andererseits gibt es Funktionen, die für jede Variablenordnung exponentielle OBDD-Größe besitzen, z. B. $HWB(x_1, \ldots, x_n)$ (Hidden Weighted Bit Function)

$$HWB(x_1, \ldots, x_n) = \begin{cases} x_w \text{ mit } w = \sum_{i=1}^{n} x_i & \sum_{i=1}^{n} x_i \neq 0 \\ 0 & \sum_{i=1}^{n} x_i = 0 \end{cases}$$

Satz 4.1.3 *Für jede Ordnung hat die OBDD-Darstellung von* $HWB(x_1, \ldots, x_n)$ *eine Größe von* $\Omega(1.14^n)$ *(Bryant).*

Desweiteren gibt es Funktionen, die bzgl. einer Variablenordnung lineare OBDD-Größe haben und bzgl. einer anderen Variablenordnung exponentielle OBDD-Größe, z. B. die Addition zweier Binärzahlen.

Zur Bestimmung einer optimalen Variablenordnung können folgende Techniken angewandt werden:

Erschöpfende Suche: Die Brute-Force Methode liefert ein optimales Ergebnis. Das Problem der optimalen Variablenordnung ist allerdings NP-hart.

Heuristiken: Diese werden z. B. auf eine Schaltkreisbeschreibung angewandt und sind dabei abhängig von der Topologie des Schaltkreises. Heuristiken liefern in der Regel nicht die optimale Variablenordnung und sind nicht für alle Schaltkreise gleich gut geeignet.

Dynamische Techniken: Eine dynamische Verbesserung der Variablenordung kann z. B. durch schrittweises Vertauschen von Variablen mit deren Nachbarn innerhalb des OBDDs erfolgen (siehe Sifting Algorithmus). Basierend auf einer Heuristik wird für die einzelnen Inputvariablen nacheinander eine möglichst gute Position in der Variablenordnung bestimmt. Daher führen diese dynamischen Optimierungsverfahren lediglich in lokale Minima.

4.2 Bestimmung einer optimalen Variablenordnung

4.2.1 Komplexitätsbetrachtung

Gegeben: Schaltkreis S auf $X = \{x_1, \ldots, x_n\}$ sowie Permutation π auf $I = \{1, \ldots, n\}$.

Problem: Ist π eine optimale Variablenordnung für die durch S berechnete Funktion?

Satz 4.2.1 *Das o.g. Problem ist NP-hart*

Beweis: Reduktion von SAT.

Sei $CNF(y)$ eine konjunktive Normalform auf den Variablen $y_1, \ldots, y_n$.
Sei $f = (x_1x_2 \vee x_3x_4) \wedge CNF(y)$.
Sei $\pi = (x_1, x_3, x_2, x_4, y_1, \ldots, y_n)$.

Falls $CNF(y)$ nicht erfüllbar ist, dann ist $f \equiv 0$ und π ist optimal.
Falls $CNF(y)$ erfüllbar ist, dann ist $\pi' = (x_1, x_2, x_3, x_4, y_1, \ldots, y_n)$ um mindestens einen Knoten besser als $\pi \Rightarrow \pi$ ist nicht optimal.

Damit ist die Entscheidung, ob π optimal ist, äquivalent zur Entscheidung, ob $CNF(y)$ erfüllbar ist. □

4.2.2 Algorithmus zur Bestimmung einer optimaler Variablenordungen

Einfachste Methode: **Brute Force**. Es werden alle $n!$ möglichen Ordnungen der n Variablen ausprobiert. Dieses Verfahren hat bei naiver Implementation eine Zeitkomplexität von $O(n!2^n)$. *Friedman/Supowit* beschrieben 1990 eine Methode zur Bestimmung einer optimalen Variablenordnung mit Zeitkomplexität $O(n^2 3^n)$, die von *Sieling/Wegener* durch Verwendung von Hashing und Bucketsort auf $O(n3^n)$ reduziert werden konnte. *Ishiura,*

Sawada und *Yajima* vereinfachten das Verfahren von Friedman/Supowit durch dynamisches Umordnen von OBDDs und formulierten eine Methode nach dem *Branch and Bound* Verfahren.

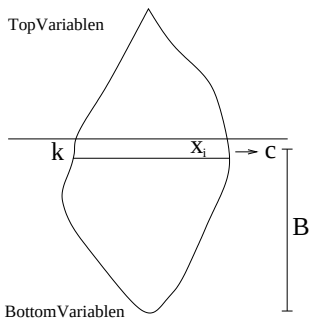
Beobachtung: Zwei verschiedene Permutationen der Inputvariablen mit gemeinsamen Suffix führen zu OBDDs mit identischem Rumpf

Vereinbarungen: Sei $f \in \mathcal{B}_n$ und $N = \{1, \ldots, n\}$ die Menge der Variablenindizes.
Π sei eine Permutation von N.
Der OBDD für f mit der Ordnung Π sei $OBDD(f, \Pi)$.
Sei $B \subseteq N$, $\Pi(B) = \{\Pi | \Pi[n-j+1] \in B,\ j = 1 \ldots |B|\}$.
Variablen mit Index in B sind Bottomvariablen, Variablen mit Index in $N - B$ sind Topvariablen.
Sei $N_i(f, \Pi)$ die Anzahl der Knoten ν mit Variablenbelegung x_i in $OBDD(f, \Pi)$.

Der folgende Satz zeigt, daß man mit einem horizontalen Schnitt im OBDD die Bottomvariablen von den Topvariablen trennen kann. Dabei beeinflußt eine Änderung der Ordnung der Topvariablen nicht die Azahl der Knoten im darunterliegenden OBDD (und umgekehrt).

Satz 4.2.2 *(Struktursatz)*
Sei $B \subseteq N$, $k = n - |B| + 1$, x_i Variable mit $i \in B$. Dann existiert eine Konstante c mit:
$\forall \Pi \in \Pi(B)$*:* $\Pi(k) = i \ \Rightarrow\ N_i(f, \Pi) = c$.

Beweis: Sei k die Position der ersten Bottomvariable. Jeder Knoten mit der Variablen x_i ist mit mindestens einer Variablenbelegung $a_1, \ldots, a_{i-1}$ der Variablen $x_1, \ldots, x_{i-1}$ erreichbar. Umgekehrt führt jede solche Belegung zu einem Knoten mit einer Variablen aus $\{x_i, \ldots, x_n\}$ oder zu einer Senke.
Die Markierung dieser Knoten ist genau dann gleich x_i, wenn der Kofaktor $f_{x_1=a_1,\ldots,x_{i-1}=a_{i-1}}$ von x_i abhängt. Alle mit x_i markierten Knoten berechnen wiederum verschiedene Funktionen (Kanonizität); d. h. die Zahl der mit x_i markierten Knoten ist gleich der Zahl der Restriktionen $f_{x_1=a_1,\ldots,x_{i-1}=a_{i-1}}$ von f, die von x_i abhängig sind. Da die Kofaktorbildung kommutativ bzgl. der Reihenfolge ist, ist die resultierende Funktion unabhängig von der Variablenordung der Topvariablen und daher für alle $\Pi \in \Pi(B)$ identisch.



$\Rightarrow$ Die Anzahl der eindeutigen Kofaktoren, bzgl. aller möglichen Belegungen der Top-Variablen, die von x_i abhängig sind, ergibt die Konstante c. □

Der Struktursatz kommt bei komplexitätstheoretischen Betrachtungen von OBDDs zur Anwendung. Zur Bestimmung einer optimalen Variablenordung kann er folgendermaßen verwendet werden:

Algorithmus-Skizze (induktiv):
Für alle $(s-1)$-elementigen Teilmengen $B \subseteq N$ mit $|B| = s-1$, sei die optimale Größe des jeweils durch die Bottomvariablen definierten Teil des OBDDs für f bekannt. Anfangsbedingung: Für $s=0$ (Senke) ist die optimale Größe 1.

Für ein gegebenes $B \subseteq N$, $|B| = s$ können wir ein optimales OBDD folgendermaßen bestimmen:

Betrachte jede Variable, deren Index in B als erste Bottom-Variable x_i , $i \in B$, $\Pi[k] = i$ ist. Für x_i als erste Bottom-Variable folgt $B' = B - \{i\}$. Die minimale Größe des unteren Teils des OBDD mit x_i als erster Bottomvariablen ergibt sich aus der zuvor berechneten minimalen Größe für B' plus der Anzahl der Knoten, die mit x_i belegt sind. Der Struktursatz garantiert dabei, daß die Anzahl nicht von der Ordnung in B' abhängt.

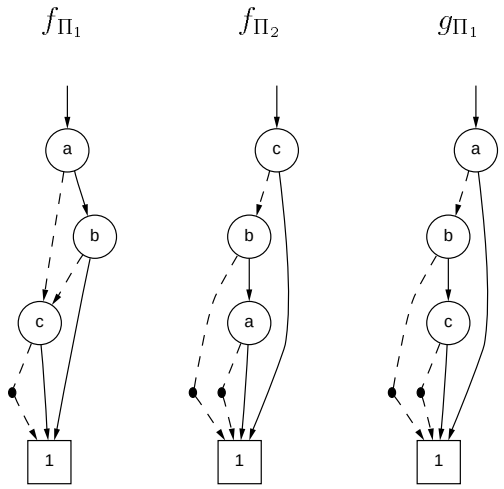
Nun kann s schrittweise erhöht werden, bis eine optimale Ordung für alle Variablen gefunden ist. Wir betrachten also jede Teilmenge $B \subseteq \{1, \ldots, n\}$ geordnet nach steigender Kardinalität. Die Knoten, die mit x_i markiert sind, können einfach im jeweiligen OBDD gezählt werden.

Gedanken zu einer effizienten Implementierung:
Es ist nicht nötig, alle OBDDs für $B \subseteq N$ neu zu konstruieren. Daher betrachten wir folgende Fragen:

- Wie leitet man OBDDs für eine gegebene Variablenordnung von einem OBDD mit einer anderen Variablenordnung ab?
- Wie kann man gleichzeitig OBDDs unterschiedlicher Ordnung im gleichen Graphen speichern?

Annahme: Die Ordnung Π_2 kann aus der Ordnung Π_1 abgeleitet werden durch Vertauschung von x_i und x_j. Dann sind $OBDD(f, \Pi_2)$ und $OBDD(g, \Pi_1)$ mit $g = f|_{x_i=x_j, x_j=x_i}$ isomorph.

Beispiel 4.2.3: $f = ab + c$, $\Pi_1 : a \leq b \leq c$, $\Pi_2 : c \leq b \leq a$, $g = cb + a = f|_{a=c, c=a}$:



Dies erlaubt, OBDDs unterschiedlicher Ordnung als unterschiedliche Funktionen zu betrachten.

Wir können einen OBDD von einem anderen durch Erweiterung bzgl. der orthonormalen Menge $\{x_i x_j,\ x_i\overline{x_j},\ \overline{x_i}x_j,\ \overline{x_i x_j}\}$ ableiten:

$$f = x_i x_j f_{x_i x_j} + x_i\overline{x_j} f_{x_i\overline{x_j}} + \overline{x_i}x_j f_{\overline{x_i}x_j} + \overline{x_i}\,\overline{x_j} f_{\overline{x_i}\,\overline{x_j}}$$

g ist darstellbar als:

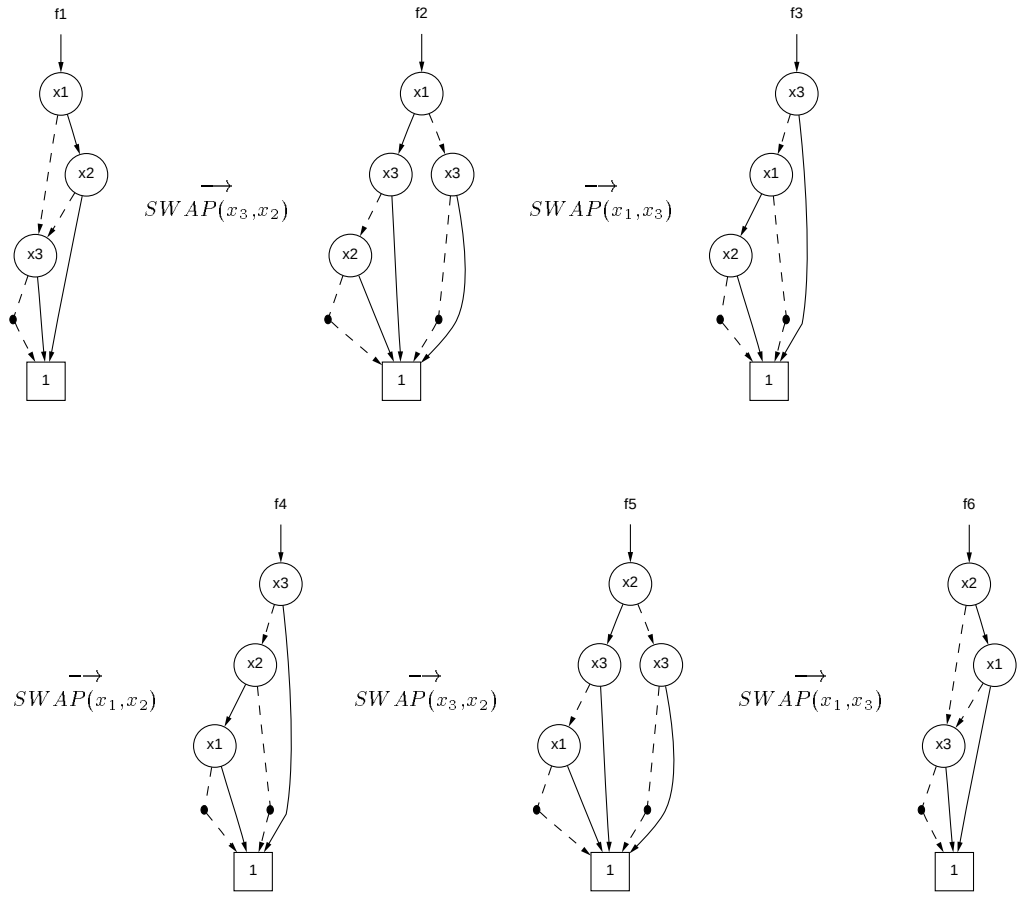
$$g = x_i x_j f_{x_i x_j} + \overline{x_i}x_j f_{x_i\overline{x_j}} + x_i\overline{x_j} f_{\overline{x_i}x_j} + \overline{x_i}\,\overline{x_j} f_{\overline{x_i}\,\overline{x_j}}.$$

g wird von f abgeleitet durch die Berechnung von vier Kofaktoren und die Ausführung von drei ITE-Operationen. Allgemein läßt sich jede Permutation der Variablenmenge durch Folgen von paarweisen Vertauschungen erreichen. Ein Algorithmus zur Bestimmung der optimalen Variablenordnung läßt sich folgendermaßen definieren:

1. Starte mit Π_0 und baue $OBDD(f, \Pi_0)$ auf.
2. Durch paarweises Vertauschen erhalten wir $n-1$ OBDDs mit jeweils einer anderen Variablen am unteren Ende (entspricht allen $B \subseteq N$ mit $|B| \leq 1$).
3. Bilde danach alle $\binom{n}{2}$ Teilmengen B mit $|B| = 2$, $B = \{x_i, x_j\}$.
4. Betrachte zwei OBDDs mit x_i und x_j als Bottomvariable. Dann ist $B' = \{x_i\}$ bzw. $B' = \{x_j\}$.
5. Wenn x_i Bottom-Variable ist und x_j die nächsthöhere Variable, dann zähle alle Knoten, die mit x_j belegt sind dazu. Wenn x_j innerhalb des OBDDs an einer höheren Position steht und x_k die bzgl. x_i nächsthöhere Variable ist, dann führe $SWAP(x_j, x_k)$ durch.
6. Verfahre ebenso im OBDD mit x_j als Bottomvariablen.
7. Behalte die OBDDs mit der geringsten Knotenanzahl für das gewählte B und lösche die übrigen OBDDs.
8. Wiederhole dies für alle $|B| = s$ mit $\binom{n}{s}$ Teilmengen. Dabei werden jeweils maximal s SWAP-Operationen benötigt.

Folgendes Beispiel soll die Funktionsweise des Algorithmus demonstrieren und nicht dessen Effizienz.

Beispiel 4.2.4: $f = x_1 x_2 + x_3,\ \Pi_0 : x_1 < x_2 < x_3$



Zuerst werden alle OBDDs für einelementige Teilmengen $B \subseteq N$, $|B| = 1$ berechnet. Die Kosten sind für alle OBDDS f_1, f_2 und f_4 gleich 2 (Die Senke plus Knoten dessen Variablenindex in B ist).

Betrachte nun die zweielementigen Teilmengen $B \subseteq N$, $|B| = 2$.
Sei als erstes $B = \{x_2, x_3\}$. f_1 und f_2 stellen die zugehörigen OBDDs dar.
Die Kosten betragen $cost(f_1, (x_2, x_3))) = 3$ und $cost(f_2, (x_3, x_2))) = 4$. Daher wird f_1 für $B = \{x_2, x_3\}$ ausgewählt.

Jetzt betrachten wir $B = \{x_1, x_2\}$, d.h. f_3 und f_4.
Die Kosten belaufen sich auf $cost(f_3, (x_2, x_1))) = 3$ und $cost(f_4, (x_1, x_2))) = 3$.

Die letzte zweielementige Teilmenge $B = \{x_1, x_3\}$ wird durch f_5 und f_6 beschrieben. Die Kosten belaufen sich auf $cost(f_5, (x_3, x_1))) = 4$ und $cost(f_6, (x_1, x_3))) = 3$. Ausgewählt werden f_1, f_3, und f_6.

Zur Bestimmung der OBDDs für $B = \{x_1, x_2, x_3\} = N$ müssen keine weiteren SWAPs durchgeführt werden, da die einzige Variable, die nicht in B' ist, an oberster Stelle steht. Die Kosten belaufen sich wiederum bei allen übrigen OBDDs auf 3, so daß jede dieser Lösungen eine optimale Variablenordnung bezeichnet.

Beobachtung: Wir können den Algorithmus verbessern.

- Wenn wir ein OBDD erhalten, in dem es für jede Variable exakt einen Knoten gibt, kann der Algorithmus sofort abbrechen, da eine optimale Lösung gefunden wurde.
- Es gibt Funktionen, die bzgl. bestimmter Variablen Symmetrien aufweisen. Die Größe eines OBDDs ändert sich nicht, wenn symmetrische Variablen permutiert werden. Daher sollte der Tausch symmetrischer Variablen vermieden werden.
- Der Algorithmus zählt implizit alle Variablenordnungen auf. Daher können wir ein *Branch and Bound* Verfahren anwenden:
  - obere Grenze: kleinste berechnete OBDD-Größe für Teilmenge $B \subseteq N$
  - untere Grenze: berechnete OBDD-Größe plus geschätzte minimale Zahl Knoten, um OBDD zu vervollständigen.
  - Sei die Anzahl der Knoten mit dem Index der ersten Bottomvariable gleich t, dann würden mindestens $t-1$ Knoten benötigt, um den OBDD zu vervollständigen. D.h. wenn $|B| + (t-1) \geq$ obere Grenze, dann verwerfe $|B|$ und das zugehörige OBDD.
  - Die Effizienz des *Branch and Bound* Verfahrens hängt von der ursprünglichen Variablenordnung ab.

4.3 Heuristiken zur Bestimmung guter Variablenordnungen

Motivation: Finde eine gute Ordnung für die primären Inputs bei der Konstruktion eines OBDDs für einen kombinatorischen Schaltkreis. Diese kann als Startpunkt für ein *exaktes Verfahren* zum Finden der optimalen Variablenordnung oder für ein *iteratives Verfahren* zur Verbesserung der Variablenordnung genutzt werden. Die Qualität der Ausgangsordnung beeinflußt die Konvergenz beider Verfahren.

4.3.1 Single-Output Funktionen

Betrachte: Der Schaltkreis sei als DNF gegeben.

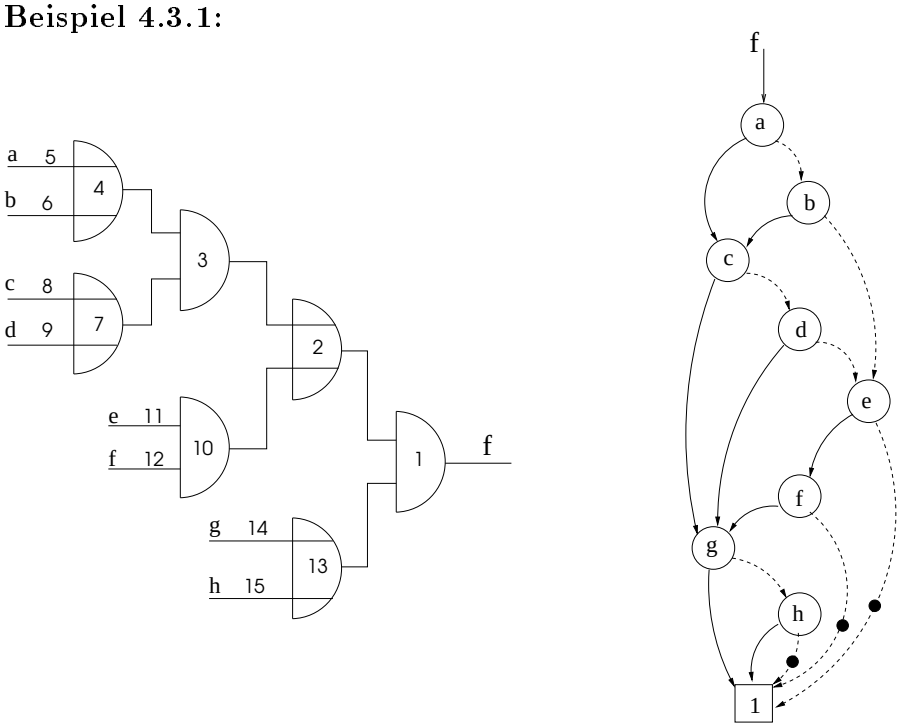
Idee: Nehme die *wichtigste* Variable der DNF als Topvariable des OBDD. Diese Variable kann z. B. die in den Mintermen am häufigsten vorkommende Variable sein. Dadurch wird die Anzahl der Minterme in den jeweiligen Kofaktoren minimiert, was ebenfalls zu kleineren OBDDs führt. Die *wichtigste* Variable hat den größten Einfluß auf den Funktionswert und sollte daher möglichst frühzeitig im OBDD geprüft werden.

$\Rightarrow$ Ordne die Variablen nach ihrer *Wichtigkeit*.

Betrachte: Der Schaltkreis sei in Multi-Level-Form gegeben.

Idee: Wir betrachten den einfachsten Fall. Für *fanout-freie* Schaltkreise (Fanout jedes Gatters gleich 1), bestehend aus AND, OR und NOT Gattern, kann ein OBDD mit nur je einem Knoten pro Variable konstruiert werden, indem man eine Variablenordnung mittels *DFS* Durchlauf des Schaltkreisgraphen ermittelt. Die Variablenordnung entspricht dabei der Reihenfolge der im *DFS*-Durchlauf besuchten Blätter.

Beispiel 4.3.1:



Allgemein:

- Der Schaltkreis wird rekursiv vom Output ab in *DFS-order* durchmustert.
- Die Inputs werden dabei in der Reihenfolge eingeordnet, wie sie angetroffen werden.
- *DFS-Heuristiken* unterscheiden sich in der Reihenfolge, in der die Vorgänger eines Gatters besucht werden.

Fanin-Heuristik nach Malik:

- Kenngröße: Tiefe des Teilschaltkreises, für den das betrachtete Gatter den (einzigen) Augang darstellt.

DFS-Heuristik im SIS-Paket:

- Als zusätzliche Kenngröße werden Informationen außerhalb des transitiven Fanin herangezogen.
- Zu jedem Gatter werden bestimmt:

$$Level(v) := \begin{cases} 0 & \text{falls } v \text{ Output} \\ 1 + \max\{Level(w) | w \text{ ist Nachfolger von } v\} & \text{sonst} \end{cases}$$

$$MaxLeaf(v) := \begin{cases} Level(v) & \text{falls } v \text{ Input} \\ 1 + \max\{MaxLeaf(w) | w \text{ ist Vorgänger von } v\} & \text{sonst} \end{cases}$$

- Die Anordnung erfolgt primär nach absteigendem *MaxLeaf* und sekundär nach aufsteigendem *Level*.

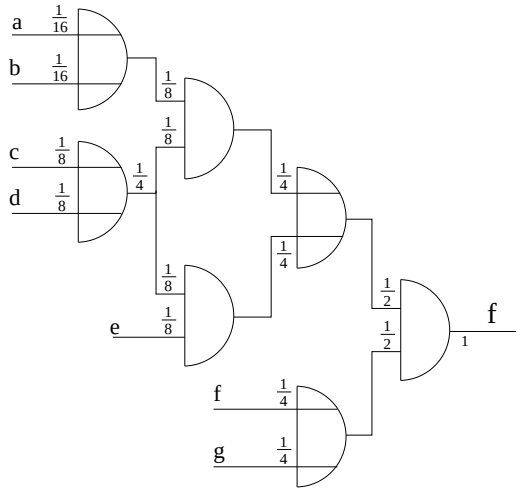
Weight-Propagation Methode
Wir nehmen die Variablen im OBDD nach oben, die die Funktion am stärksten beeinflußen. Dazu gewichten wir Ein- und Ausgänge jedes einzelnen Knoten im Netzwerk folgendermaßen:

- Der Output erhält das Gewicht 1
- Jeder Input eines Gatters v mit n Eingängen erhält $\frac{1}{n}$ des Gewichts am Output des Gatters, so daß gilt:

$$weight(n_i) := \sum_{j}^{n} weight(n_j) \ , \ n_j \text{ ist Eingang von } n_i$$

- Fahre fort, bis alle Inputs eine Gewichtung erhalten haben.
- Der Input mit dem größten Gewicht im ganzen Netzwerk wird die erste Variable in der Ordnung.
- Entferne den Input aus der Schaltkreisnetzliste.
- Starte die Berechnung der Gewichtung erneut für den vereinfachten Schaltkreis.
- Der Algorithmus endet, wenn alle Inputs angeordnet sind.

Beispiel 4.3.2:



4.3.2 Multiple-Output Funktionen

Idee:

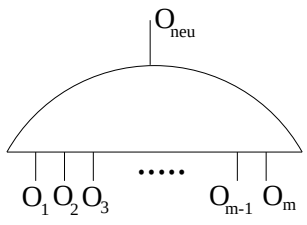
- Ordne die zu jedem Output gehörigen Variablen einzeln an.
- In welcher Reihenfolge sollen dann die Outputs betrachtet werden?

Betrachte: Addierer

- Wir wollen das *most significant bit* des Inputs an vorderster Stelle in der Variablenordnung. Dazu müßen wir zuerst das *most significant bit* des Outputs bestimmen (entspricht dem Output mit dem tiefsten Teilschaltkreis).
- Ordne die Outputs f_i so an, daß jeweils die Anzahl der zur Ordnung zugefügten Variablen minimal ist.
- Seien $f_1, \ldots, f_m \in \{0,1\}^k \to \{0,1\}$, $k \leq n, S_i = \{x_{i_1}, \ldots, x_{i_k}\}$ für $i = 1, \ldots, m$ die zu f_i gehörige Variablenmenge , $S_i \subseteq \{x_1, \ldots, x_n\}$.
  Minimiere $\sum_{i=1}^{m} | \bigcup_{j=1}^{i} S_j |$

Alternative aus der Praxis

Plaziere ein Gatter über alle m Outputs mit m Eingängen und wende dann eine Methode für Single-Output-Funktion an.



Bei Schaltkreisen mit *bounded fanin* konstruiere man einen vollständigen Baum von Gattern über den Outputs, so daß die Tiefeninformation bis auf einen Offset erhalten bleibt.

Keine der vorgestellten Heuristiken eignet sich für alle Schaltkreise gleich gut, d.h. eine Heuristik kann für einen Schaltkreis eine sehr gute Variablenordnung liefern, für einen anderen jedoch eine sehr schlechte.

Frage: Wie kann man die Güte einer Heuristik für einen bestimmten Schaltkreis bestimmen?

Zur Abschätzung der Güte einer bestimmten Heuristik in Bezug auf einen bestimmten Schaltkreis muß man nicht das zum Schaltkreis äquivalente OBDD vollständig aufbauen, sondern kann folgendermaßen verfahren:

- Aufbau eines *partiellen* OBDDs, d.h. man gibt für die maximale Größe des OBDD einen bestimmten fixen Wert vor:
- Der *ite*-Algorithmus wird dahingehend modifiziert, daß ein Knoten nur dann angelegt wird, wenn das festgelegte Limit nicht überschritten wird.
- Würde das festgesetzte Limit überschritten, wird der Funktion mit *don't care* belegt.
- Die Güte der Heuristik wird bewertet nach der Größe der jeweiligen *don't-care*-Menge.

4.4 Iterative Verbesserung der Variablenordnung

Der bereits genannt exakte Algorithmus zum Auffinden der optimalen Variablenordnung durchsucht implizit alle möglichen Variablenordnungen durch fortgesetztes Vertauschen von zwei benachbarten Variablen (SWAP). Diese Idee kann für eine Heuristik genutzt werden, die die Anzahl der durchmusterten Permutationen gegenüber der exakten Methode erheblich einschränkt. Wir erhalten eine *lokale Suchoperation*, wobei die Grundordnung solange durch Tauschen von Gruppen von Variablen verändert wird, bis eine *Terminalbedingung* erfüllt ist.

Variationsmöglichkeiten:

- Unterschiedliche Regeln zum Ableiten einer neuen Variablenordnung aus einer alten.
- Wann wird eine neue Ordnung akzeptiert (Terminalbedingung) ?

4.4.1 Möglichkeiten zum Vertauschen von Variablen

Paarweiser Tausch benachbarter Variablen (SWAP(i-1,i)):
$abcdef \implies bacdef$

Permutation von Gruppen p benachbarter Variablen:
Sei $p = 3$ $abcdef$, permutiere Gruppe bcd: $acbdef$, $acdbef$, $adcbef$, $adbcef$, $abdcef$
$p!$ Permutationen und $p! - 1$ SWAPs

Paarweiser Tausch beliebiger Variablen (EXCHANGE(i,j)):
$abcdef \implies aecdbf$

Verschieben einer Variablen (JUMP(i,j)): Variable x_i springt an Position j (sodaß die ehemaligen Variablen x_k an die neue Position $k+1$ verschoben werden für $j \leq k < i$). Die anderen Variablen bleiben an ihrer ursprünglichen Position.
$i > j$: JUMP-UP
$i < j$: JUMP-DOWN

Sifting: Suche für eine Variable die optimale Position, während die relative Position der anderen Variablen unverändert bleibt. Nachdem alle Variablen an ihre jeweils optimale Position geschoben wurden erreichen wir ein lokales Minimum.

Group-Sifting: Bewege Gruppen von Variablen durch die Ordnung, während der Rest der Variablen fixiert bleibt, bis ein lokales Minimum erreicht wird.

Bemerkung 4.4.1:

- Der paarweise Tausch benachbarter Variablen (SWAP) ist eine lokale Operation.
- EXCHANGE läßt sich über eine Folge von SWAPs realisieren.
- JUMP kann ebenfalls durch Folgen von SWAPs realisiert werden.
- EXCHANGE kann durch zwei JUMPs simuliert werden.
- Durch fortlaufende SWAPs kann jede beliebige Variablenordnung erreicht werden.
- Als Strategie für eine Optimierung kann jede bekannte Strategie zur kombinatorischen Suche angewandt werden, z. B. :
  - Greedy Methode
  - Simulated Annealing
  - Evolutionäre Algorithmen.

Satz 4.4.2 *Seien $s_1, \ldots, s_n$ die Knotenzahlen auf den Ebenen eines OBDD der Ordnung Π. Sei Π' die durch JUMP-UP(i,j) entstehende Ordnung. Dann ist die Größe des äquivalenten OBDD der Ordnung Π' beschränkt durch:*

a) $2(s_1 + \ldots + s_{i-1}) + 1 + s_{i_1} + \ldots + s_n + 2$

b) $s_1 + \ldots + s_{j-1} + 3(s_j + \ldots + s_{i-1}) + s_{i+1} + \ldots + s_n + 2$

Beweis: Übungsaufgabe □

Korollar 4.4.3: Für SWAP$(i-1, i)$ werden die Schranken zu

a) $2(s_1 + \ldots + s_{i-1}) + 1 + s_{i+1} + \ldots + s_n + 2$

b) $s_1 + \ldots + s_{i-2} + 3s_{i-1} + s_i + \ldots + s_n + 2$

Satz 4.4.4 *Für JUMP-DOWN(i, j) gelten die Schranken*

a) $s_1 + \ldots + s_{i-1} + (s_{i+i} + \ldots + s_n + 2)^2$

b) $s_1 + \ldots + s_n + (2^{j-i+1} - 2)s_i$

Korollar 4.4.5: Sei G ein OBDD und G^* das durch eine lokale Veränderung entstehende reduzierte OBDD. Dann gilt für:

- SWAP	:	$\|G\|/2 \leq \|G^*\| \leq 2\|G\|$
- EXCHANGE	:	$\|G\|^{1/2} \leq \|G^*\| \leq \|G\|^2$
- JUMP-UP	:	$\|G\|^{1/2} \leq \|G^*\| \leq 2\|G\|$
- JUMP-DOWN	:	$\|G\|/2 \leq \|G^*\| \leq \|G\|^2$

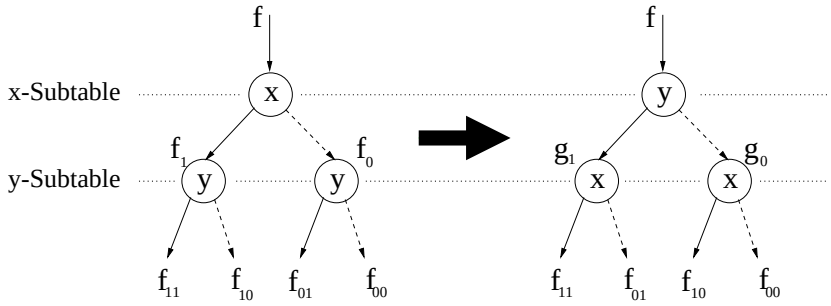
Bemerkung 4.4.6: Die inversen Operationen zu SWAP und EXCHANGE sind wiederum SWAP und EXCHANGE. Für JUMP-UP ist die inverse Operation JUMP-DOWN und umgekehrt.

Gedanken zu einer effizienten Implementierung:

- Die Unique Table muß als Menge von Subtables organisiert sein (Ein Subtable pro Variable). Damit wird gewährleistet, daß der Zugriff auf alle Knoten x_i unabhängig ist von der Gesamtgröße des OBDDs.

- SWAP(x_i, x_j) ist eine lokale Operation.
  Seien x, y benachbarte Variablen mit $x < y$. f sei eine Boolesche Funktion, die von x, y abhängt

$$\begin{array}{rcccl} f & = & xf_1 + \overline{x}f_0 & = & x(yf_{11} + \overline{y}f_{10}) + \overline{x}(yf_{01} + \overline{y}f_{00}) \\ f & = & yg_1 + \overline{y}g_0 & = & y(xf_{11} + \overline{x}f_{01}) + \overline{y}(xf_{10} + \overline{x}f_{00}) \end{array}$$



- Knoten x wird mit Label y belegt, $f_{11}, f_{10}, f_{01}, f_{00}$ können wiederverwendet werden, Knoten für g_0 und g_1 müssen neu angelegt werden.

- Ehemalige x-Knoten müssen beibehalten werden, da andere Knorten auf diese verweisen (sonst müssen Verweise geändert werden).

- Dekrementiere Referenzzähler für f_0 und f_1. Wenn alle Knoten im x-Subtable abgearbeitet sind, lösche alle Knoten aus y-Table mit Referenzzähler gleich 0. Nicht alle y-Knoten dürfen gelöscht werden, da Referenzen aus anderen Schichten auf diese zeigen können. Diese y-Knoten müssen in den x-Subtable eingehängt werden.

- Wenn $f_{11} = f_{01}$, dann wird g_1 nicht erzeugt.
  Wenn $f_{00} = f_{10}$, dann wird g_0 nicht erzeugt.
  Wenn vorher $f_{11} \neq f_{10}$ und $f_{01} \neq f_{00}$, wird die Anzahl der Knoten dekrementiert.

4.4.2 Static Reordering und Dynamic Reordering

Static Reordering
Der OBDD für die Boolesche Funktion f wird mit einer vorgewählten Variablenordnung aufgebaut. Erst dann beginnt der Reordering Prozeß.

Dynamic Reordering
Reordering wird in den Synthese-Prozeß miteinbezogen.

- Wann ?

  - Wenn die Anzahl der Knoten im OBDD einen bestimmten Schwellwert überschreitet oder
  - während der Garbage-Collection.

- Wie ?
  - Halte Synthese an.
  - Permutiere die vorhandenen Variablen ($\rightarrow$ Sifting).
  - Fahre mit Synthese fort.

- Dynamischer Variablentausch ist oft sehr effizient, da ein Reordering nur dann durchgeführt wird, wenn es wirklich notwendig wird.

Bemerkung 4.4.7: Das Unterbrechen der Syntheseoperation setzt große Sorgfalt in der Implementation voraus. Die Anwendung von Sifting kann die Berechnungszeit während der OBDD-Synthese signifikant erhöhen.

Steigerung der Effizienz

- Suche nach Inputvariablen, die sich nicht gegenseitig beeinflußen und speichere diese, damit ineffiziente SWAPs vermieden werden können (*Interaction Matrix*).
  - Sei f eine Boolesche Funktion, die durch das OBDD P_f repräsentiert wird. f hänge nicht gleichzeitig von den Variablen x und y ab, d.h. in OBDD P_f existiert keine direkte Kante von einem Knoten x zu einem Knoten y. Um SWAP(x, y) durchzuführen müßten neue Variablen eingeführt werden.

- Identifiziere zusammenhängende Gruppen durch Prüfen auf Symmetrie
  - Sei $f(x_1, \ldots, x_n)$ symmetrisch in x_i, x_j, d.h. wenn $f_{x_i,\overline{x_j}} = f_{\overline{x_i},x_j}$ oder $f_{x_i,x_j} = f_{\overline{x_i},\overline{x_j}}$ ändert sich die Größe des OBDD für f bei SWAP(x_i, x_j) nicht.

Kapitel 5

Erreichbarkeitsanalyse und Verifikation sequentieller Schaltkreise

5.1 Vorbemerkung

Validation
Die Spezifikation des Systems stimmt mit der Entwurfsidee[1] überein.

Damit ist die Überprüfung, ob die Spezifikation bestimmte Eigenschaften erfüllt bzw. unerwünschte Eigenschaften nicht erfüllt, möglich.

Ergebnis: Das Vertrauen in die Spezifikation wird gehoben.

Verifikation
Die Spezifikation stimmt mit der Implementierung überein. Ein formaler Beweis der Übereinstimmung ist möglich (*formale Verifikation*).

Wegen der großen Ähnlichkeit der beiden Aufgaben wird oft nur von der *Verifikation* gesprochen.

Lösungsansätze:

Fehlersuchtechniken: Prototypentest, Simulation: *informale Verifikation.*
Die eigentliche Verifikation ist unvollständig.

Konstruktionstechniken: Korrektheitserhaltende Synthese und Transformation.
Aus einer gegebener Spezifikation bzw. Implementierung werden Implementierungen konstruiert, so daß die spezifizierten Eigenschaften erhalten bleiben.

[1] Die Entwurfsidee besteht aus gewissen Eigenschaften, die das System haben soll. Sie ist allerdings nicht formalisiert und daher ist auch kein formaler Beweis möglich.

Beweistechniken: *formale Verifikation.*

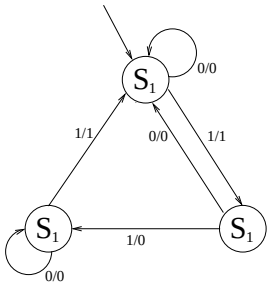
Die Spezifikation und die Implementierung werden miteinander verglichen. Es entsteht ein formaler Beweis der Übereinstimmung (d. h. bestimmte Eigenschaften werden erfüllt).

Methoden der formalen Verifikation

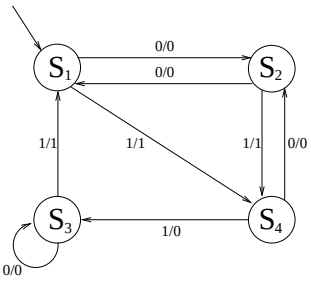
- Theorem-Beweiser: *Boyer, Moore*
  Das System wird als Axiomen-Menge beschrieben und die gewünschten Eigenschaften als Theoreme abgeleitet.
- Prozeßalgebra: *Milner*
- Temporale Logik: *Pnueli*
- Endliche Automaten:
  - Die zu verifizierenden Eigenschaften werden durch endliche Automaten beschrieben.
  - Das zu verifizierende System wird als eine Verbindung von endlichen Automaten repräsentiert.
  - Die entsprechenden Automaten können nun untersucht werden:
    * Der Äquivalenztest ist für deterministische Automaten in quadratischer Zeit bzgl. der Anzahl der Zustände lösbar.
    * Für nichtdeterministische Automaten ist das Problem PSPACE-vollständig.

5.2 Äquivalenztest für Finite State Machines (FSMs)

Beispiel 5.2.1: Sind die beiden FSMs äquivalent? ($\rightarrow$ gleiches Ein-/Ausgabeverhalten)



M_1



M_2

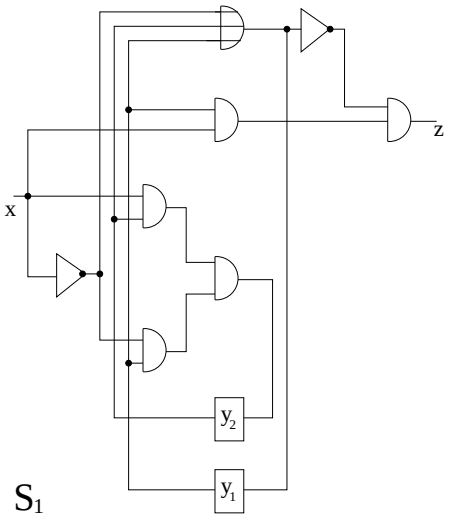
Wir kodieren die Zustände der FSMs binär und identifizieren die Maschine mit einem Schaltkreis, welcher die Ausgabefunktion der FSM berechnet. In diesem Beispiel geschieht die Kodierung durch die zwei Latches y_1 und y_2:

Zustand	y_1	y_2
S_1	0	0
S_2	0	1
S_3	1	0
S_4	1	1

Problem: Finden einer geeigneten Kodierung für die Zustände und die Eingaben.

Im folgenden repräsentiert x die Eingabe und z ist die entsprechende Ausgabe.

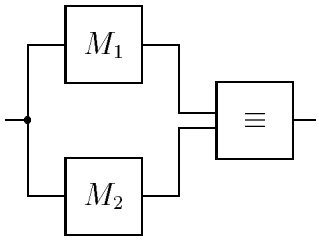
Vorsicht: Der Schaltkreis ist rückgekoppelt, da durch die Zustandsüberführungsfunktion δ der FSM der Folgezustand berechnet wird, der dann wiederum als Eingabe verwendet wird.



Aufgaben:

- Teste die Äquivalenz von M_1 und M_2.
- Ist S_1 eine korrekte Implementation von M_1?
- Ist S_1 eine korrekte Implementation von M_2?

Idee:
Man kann FSM und Schaltkreis als OBDD darstellen. Anschließend konstruiert man die *Produktmaschinen* $M_1 \times M_2, M_1 \times S_1$ und $M_2 \times S_1$



Nun untersucht man den *Zustandsüberführungsgraph* (*state transition graph, STG*). Das resultierende OBDD muß im Falle der Äquivalenz nur aus der 1-Senke bestehen.

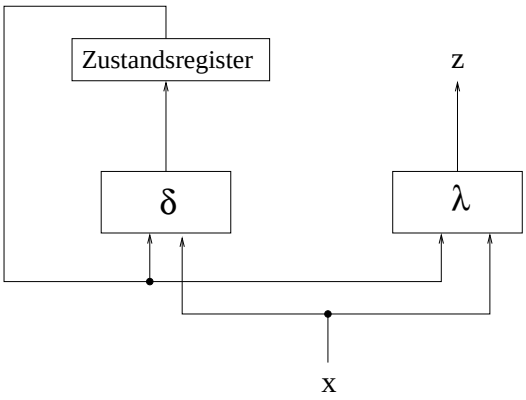
Bemerkung 5.2.2: Die Verifikation einer *globalen* Eigenschaft (Äquivalenztest) wird zurückgeführt auf die Verifikation *lokaler* Eigenschaften.

Grundlegend für diesen Test ist eine Erreichbarkeitsanalyse: *FSM-Traversal* (Vergleiche Erfüllbarkeit für kombinatorische Schaltkreise).

5.3 FSM Traversal

Definition 5.3.1: Eine Finite-State-Machine (**FSM**) ist ein 6-Tupel $(I, S, O, \delta, \lambda, S_0)$ mit:

I	...	Input-Alphabet
S	...	Zustandsmenge
O	...	Output-Alphabet
δ	...	next-state function (Überführungsfunktion)
λ	...	output function (Ausgabefunktion)
S_0	...	Initialzustand (*reset*)



Häufig ist die Betrachtung des Ausgabeverhaltens unwichtig. Daher werden vereinfachte FSMs (*endliche Akzeptoren*) betrachtet.

Definition 5.3.2: Ein endlicher Akzeptor ist ein 5-Tupel (I, S, δ, F, S_0) mit:

I	...	Input-Alphabet
S	...	Zustandsmenge
δ	...	next-state function
$F \subseteq S$	...	akzeptierende Zustände
S_0	...	Initialzustand (*reset*)

Diese Automaten akzeptieren oder verwerfen Eingaben.

Definition 5.3.3: Ein Zustand s ist genau dann *erreichbar* von Zustand r, wenn eine Zustandsfolge $s_1, \ldots, s_k$ und eine Eingabefolge $i_0, \ldots, i_k$ existiert, mit:

$$\begin{array}{l}
\delta(r, i_0) = s_1 \\
\delta(s_j, i_j) = s_{j+1} \quad 1 \leq j \leq k-1 \\
\delta(s_k, i_k) = s \\
s \text{ ist } \textit{einfach erreichbar} \text{ von } r \Leftrightarrow \exists i \in I : \delta(r, i) = s \\
s \text{ ist erreichbar von } \tilde{S} \subseteq S \Leftrightarrow \exists t \in \tilde{S}, i \in I\text{: } \delta(t, i) = s.
\end{array}$$

Die *Traversierung* eines FSM besteht in der Ermittlung aller erreichbaren Zustände.

5.3.1 Depth-First Traversal

Wir wollen alle Zustände ermitteln, die vom Initialzustand aus erreicht werden können.

Idee: Wir führen 2 Stacks (LIFO):

1. $S_1 = \{$schon untersuchte Zustände$\}$
2. $S_2 = \{$noch zu untersuchende Zustände$\}$

Ausgangssituation: $S_1 := \{S_0\} = S_2$
Berechnungsschritt: Wiederhole solange, bis $S_2 = \emptyset$:

- Wähle $s = \texttt{pop}(S_2)$ (gerade untersuchter Zustand) und berechne alle möglichen $\delta(s, i)$ mit $i \in I$, d. h. alle Zustände, die von s aus in einem Schritt erreicht werden können.
- $S_1 := S_1 \cup s$ und Gilt $\delta(s, i) \notin S_1$, dann ist $S_2 := \texttt{push}(\delta(s, i), S_2)$.
- LIFO Speicherorganisation garantiert *DFS* (FIFO $\rightarrow$ *BFS*).

5.3.2 Symbolisches Breadth-First Traversal

Vorteil: Es können mehrere Zustände gleichzeitig bearbeitet werden.

Zustandsmenge und Eingaberaum werden binär kodiert.
$\delta : S \times I \to S$, $\tilde{\delta} : 2^S \times I \to 2^S$ mit $\tilde{\delta}(S', i) = \bigcup_{s \in S'} \{\delta(s, i)\}$

Idee: Wir können $\tilde{\delta}$ als

- Vektor von OBDDs (ein OBDD für jede Variable) der Zustandskodierung repräsentieren (*Zustandsvariable*) oder
- wir bilden die Übergangsrelation zu $\tilde{\delta}$ (wird noch definiert).

Definition 5.3.4: Sei $f : \mathbb{B} \to \mathbb{B}^m$ und $C \subseteq \mathbb{B}^n$,
$Im(f) := \{u \in \mathbb{B}^m : \exists y \in \mathbb{B}^n \text{ mit } f(y) = u\}$ heißt *Image* (Bild) von f.
$Im(f, C) := \{u \in \mathbb{B}^m : \exists x \in C \text{ mit } f(x) = u\}$ heißt *Image von f constrained by C* (Bild von f eingeschränkt auf C).

Die Berechnung von $Im(f, C)$ heißt *Constrained Image Computation.*
Anwendung beim Traversal: C ist die Menge der aktuell erreichten Zustände;
$Im(f, C)$ ist die Menge der aus C in einem Schritt erreichbaren Zustände.

Bemerkung 5.3.5: Wir haben n Variablen: $n = m + p$, wobei m die Anzahl der Zustandsvariablen und p die Zahl der primären Eingaben ist.
$\Rightarrow C \subseteq \mathbb{B}_m$, d. h. wir betrachten eigentlich $C \times \mathbb{B}_p$.

Algorithmus (Pseudocode für Symbolic Breath First Traversal)

```
traverse(δ, S⁰){
    Reached = From = S⁰;
    do{
        To = Im(δ,From);
        New = To \ Reached;
        From = New;
        Reached = Reached ∪ New;
    } while(New≠∅);
    return Reached;
}
```

Bemerkung 5.3.6:

(1) Die Vereinigungs- und Differenzmengenbildung werden durch Manipulation der charakteristischen Funktionen realisiert.

(2) Die Zahl der Iterationen gibt die *sequentielle Tiefe* der FSM an, d. h. die Länge des längsten kürzesten Weges von der ein Zustand aus dem Startzustand erreicht wird.
$\Rightarrow$ *traverse* erreicht jeden Zustand in minimaler Zahl von Schritten.

5.4 Bildberechnung

5.4.1 Allgemeine Problemstellung

(1) Wir hatten gefordert: From = New
(d. h. die zuletzt erreichten Zustände bilden den Ausgangspunkt für die neue Berechnung.)

Wir hätten jede Menge New $\subseteq$ From $\subseteq$ New $\cup$ Reached als Ausgangspunkt für die Berechnung nehmen können (wobei nicht alle der o.a. Mengen zu einer effizienten Lösung führen.
Insbesondere wäre folgendes möglich:
From = To ($\subseteq$ Reached $\cup$ New) oder From = Reached $\cup$ New
Da wir alle Mengen in Form von OBDDs bearbeiten, sollte From so gewählt werden, daß die entsprechende OBDD-Darstellung möglichst klein wird.

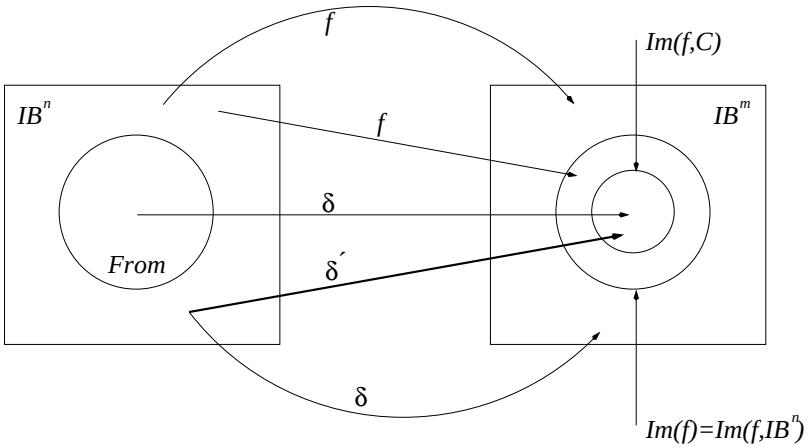
Weiter gilt: New $\subseteq$ (New)$_{\overline{\text{Reached}}} \subseteq$ New $\cup$ Reached.
$\Rightarrow$ Wir können From = New $\Downarrow \overline{\text{Reached}}$ wählen und so eine kompakte OBDD-Darstellung erreichen.

(2) **1. Ansatz:** Wir wollen Im(δ,From) berechnen.

Idee:

1) Wir suchen eine neue Funktion δ' mit Im(δ') = Im(δ,From).

2) Wir berechnen Im(δ').



Lemma 5.4.1: *Im(f,C) = Im(f$\downarrow$C) d. h. Constrain-Operator ist Image Restrictor.*

Beweis: Nach Definition von Constrain $\downarrow$ folgt:
$f = (f_1, \ldots, f_m)$, also $f \downarrow C = (f_1 \downarrow C, \ldots, f_m \downarrow C)$. $\Rightarrow f \downarrow C$ hängt nur von C ab. $\square$

Achtung:
Der Restrict-Operator ist kein Image Restrictor, denn $g \Downarrow h$ hängt nicht nur von h ab, sondern auch von g. Die Funktion g bestimmt, welche Variablen quantifiziert werden und nimmt so Einfluß.

2. Ansatz: Wir berechnen $Im(f, C)$ mit der Übergangsrelation

Definition 5.4.2: Sei $\delta = (\delta_1, \ldots, \delta_m)$ die Übergangsfunktion der FSM M.

$$T_M(y_1, \ldots, y_m, x_1, \ldots, x_n) = \prod_{i=1}^{m} (y_i \equiv \delta_i(x_1, \ldots, x_n))$$

heißt Übergangsrelation (*Transition Relation*) von M[2].

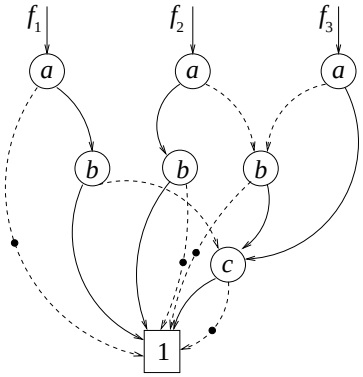
- Die Übergangsrelation beschreibt alle Zustandspaare, die im STG von M durch Kanten verbunden sind.
- Ist T_M gegeben, kann $Im(f, From)$ leicht berechnet werden:

$$Im(\delta, C) = \exists_{x_1,\ldots,x_n}(T_M(y_1, \ldots, y_m, x_1, \ldots, x_n) \cdot C(x_1, \ldots, x_n))$$

Wir betrachten das folgende Beispiel:

Beispiel 5.4.3:

$$f = (f_1, f_2, f_3), \quad \begin{array}{rcl} f_1 & = & a(b+c) \\ f_2 & = & b(a+c) \\ f_3 & = & c(a+b) \end{array}$$



a	b	c	f_1	f_2	f_3
0	0	0	0	0	0
0	0	1	0	0	0
0	1	0	0	0	0
0	1	1	0	1	1
1	0	0	0	0	0
1	0	1	1	0	1
1	1	0	1	1	0
1	1	1	1	1	1

$$Im(f) = \{000, 011, 101, 110, 111\}$$

5.4.2 Bildberechnung für die Übergangsfunktion

Sei $f = (f_1, \ldots, f_m)$ gegeben als Menge von OBDDs. Wir können o.B.d.A. annehmen, daß $C = 1$ (*unconstrained image computation*), ohne evtl. Anwendung des *Image Restrict* Operators.

[2] $n = m + p$, wobei m die Anzahl der Zustände und p die Anzahl der Eingabebits ist.

Bildberechnung durch Input Splitting

Idee: Wir berechnen $Im(f)$ mit der rekursiven Relation

$$Im(f) = Im(f_{x_i}) + Im(f_{\overline{x_i}})$$

Abbruch: f konstant.
Das berechnete Bild wird dargestellt als Vereinigung aller Ausgaben für $x_i = 1$ und für $x_i = 0$.

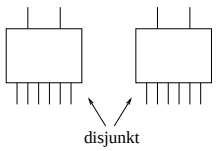
Für die charakteristische Funktion $\chi_{Im(f)}$ gilt:

$$\chi_{Im(f)} = \tilde{y}_1 \cdot \ldots \cdot \tilde{y}_m) \text{ mit } \tilde{y}_i = \begin{cases} y_i & \text{falls } f_i = 1 \\ \overline{y_i} & \text{falls } f_i = 0 \end{cases}$$

Eine Effizienzsteigerung kann durch eine geeignete Reihenfolge der Splitting Variablen und genauer Analyse der terminierenden Bedingung erreicht werden:

(a) Rekomposition in Richtung disjunkter Träger
Betrachte: $f = (f_1, f_2)$
Gilt $supp(f_1) \cap supp(f_2) = \emptyset$, dann ist $Im(f) = Im(f_1) \times Im(f_2)$
$\Rightarrow \chi_{Im(f)} = \chi_{Im(f_1)} \cdot \chi_{Im(f_2)}$ Eine Verallgemeinerung auf $f = (f_1, \ldots, f_m)$ ist möglich.

Beispiel 5.4.4: $Im(f_1, f_2) = \{00, 11\}$ $\quad$ $Im(f_3, f_4) = \{01, 10\}$



$$Im(f_1, f_2) = \{0001, 0010, 1101, 1110\}$$

Die Splitting Variablen werden jeweils mit dem Ziel ausgewählt, möglichst schnell Blöcke mit disjunktem Träger zu erhalten.

(b) Identifizierung identischer und komplementärer Komponenten
Betrachte: $f = (f_1, f_2)$; Sei $f_1 = f_2$ oder $f_1 = \overline{f_2}$
Annahme: f ist nicht konstant.
$\Rightarrow Im(f) = y_1 \equiv y_2$ bzw. $Im(f) = y_1 \oplus y_2$

Wir können alle identischen oder komplementären Komponenten bis auf eine streichen, dann das so vereinfachte Problem lösen und die fehlenden Komponenten später über Tests der Form $y_i \equiv y_j$ oder $y_i \oplus y_j$ wieder hinzufügen.

(c) Identifizierung identischer Teilprobleme
Übliche OBDD-Technik: Wir legen eine Tabelle mit bereits gelösten Teilproblemen an. Dieses Vorgehen ist sinnvoll, da im allgemeinen viele der verschiedenen Kofaktoren identisch sind.

Beispiel 5.4.5: Image Computation mit Input Splitting:
Sei $f = (f_1, f_2, f_3)$ aus Beispiel 5.4.1, wir wollen $Im(f_a)$ berechnen.
Es gilt:

$$\begin{aligned}(f_{1_a}) &= b + c\\ (f_{2_a}) &= b\\ (f_{3_a}) &= c\end{aligned}$$

Es wurde keine Abbruchbedingung erreicht. Wir führen Image Splitting bzgl. b durch:

$$\begin{aligned}(f_{1_{ab}}) &= 1 \qquad & (f_{1_{a\overline{b}}}) &= c\\ (f_{2_{ab}}) &= 1 \qquad & (f_{2_{a\overline{b}}}) &= 0\\ (f_{3_{ab}}) &= c \qquad & (f_{3_{a\overline{b}}}) &= c\end{aligned}$$

$$\begin{aligned}\Rightarrow Im(f_a) &= Im(f_{ab}) + Im(f_{a\overline{b}})\\ &= y_1y_2 + y_1y_3\overline{y_2} + \overline{y_1}\,\overline{y_3}\,\overline{y_2}\\ &= y_1y_2 + y_1y_3 + \overline{y_1}\,\overline{y_2}\,\overline{y_3}\end{aligned}$$

Jetzt ist $Im(f_{\overline{a}})$ zu berechnen:

$$\begin{aligned}&(f_{1_{\overline{a}}}) = 0\\ &(f_{2_{\overline{a}}}) = (f_{3_{\overline{a}}}) = bc\\ &\Rightarrow Im(f_{\overline{a}}) = \overline{y_1}(y_2 \equiv y_3) = \overline{y_1}y_2y_3 + \overline{y_1}\,\overline{y_2}\,\overline{y_3}\end{aligned}$$

Insgesamt erhält man:

$$Im(f) = y_1y_2 + y_1y_3 + y_2y_3 + \overline{y_1}\,\overline{y_2}\,\overline{y_3}$$

Bildberechnung durch Output Splitting

Idee:
Wir zerlegen das Problem bzgl. der Ausgabefunktion. Die zugrundeliegende Relation lautet dabei:

$$Im(f) = y_i \cdot Im((f_1, \ldots, f_{i-1}, f_{i+1}, \ldots, f_m), f_i) + \overline{y_i} \cdot Im((f_1, \ldots, f_{i-1}, f_{i+1}, \ldots, f_m), \overline{f_i}).$$

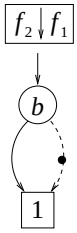
Danach ist eine rekursive Berechnung möglich, z. B. mit Hilfe des Constrain Operators.

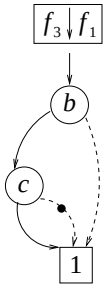
Abbruchkriterien:

Ist das Problem auf eine Komponente f_j reduziert, dann gilt:
Ist f_j nicht konstant, dann ist $Im(f) = 1$
Ist $f_j = 1$, dann ist $Im(f) = y_j$
Ist $f_j = 0$, dann ist $Im(f) = \overline{y_j}$

Beispiel 5.4.6: $f = (f_1, f_2, f_3)$ aus Beispiel 5.4.1., die Splittingreihenfolge sei f_1, f_2, f_3.

$$Im((f_2, f_3), f_1) = (\underbrace{f_2 \downarrow f_1}_{f_2^{(1)}}, \underbrace{f_3 \downarrow f_1}_{f_3^{(1)}})$$

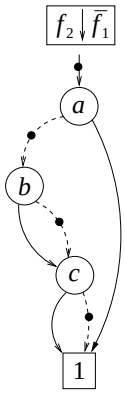


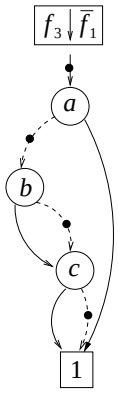


$$\begin{aligned} Im(f_3^{(1)} \downarrow f_2^{(1)}) &= Im(c) = 1 \\ Im(f_3^{(1)} \downarrow f_2^{(1)}) &= Im(1) = y_3 \end{aligned}$$

$$\Rightarrow Im((f_2, f_3), f_1) = y_2 + \overline{y_2} y_3 = y_2 + y_3$$

Analog: $Im((f_2, f_3), \overline{f_1}) = (y_2 \equiv y_3) = y_2 y_3 + \overline{y}_2 \overline{y}_3$
(identisches Unterproblem, nicht konstant)





Insgesamt:

$$\begin{aligned} Im(f) &= y_1(y_2 + y_3) + \overline{y_1}(y_2 y_3 + \overline{y}_2 \overline{y}_3) \\ &= y_1 y_2 + y_1 y_3 + y_2 y_3 + \overline{y}_1 \overline{y}_2 \overline{y_3} \end{aligned}$$

Bemerkung 5.4.7:

- Input- und Output-Splitting können kombiniert werden.
- Wir können analoge Methoden zur Effizienzsteigerung anwenden.
- Meistens ist Input Splitting besser als Output Splitting ($m \geq n$), obwohl Output-Splitting zu einer echten Zerlegung des Problems führt (disjunkt).

5.4.3 Bildberechnung für die Übergangsrelation

Wir wollen Im(δ,C) mit Hilfe der Übergangsrelation berechnen:

$$\exists_{x_1,\ldots,x_n}(T(y_1,\ldots,y_m,x_1,\ldots,x_n)\cdot C(x_1,\ldots,x_n))$$

C bezeichne die Einschränkung der present state Variablen. Ist keine Einschränkung gegeben, gilt also $C=1$, dann ist lediglich der Existenzquantor von T zu berechnen.

Beispiel 5.4.8: Für $C=1, \quad f=(f_1,f_2,f_3)$ aus Beispiel 5.4.1. gilt:

$$\begin{aligned} T(a,b,c,y_1,y_2,y_3) = \quad & (y_1ab + y_1ac + \overline{y}_1\overline{a} + \overline{y}_1\overline{b}\overline{c}) \\ & \cdot(y_2ab + y_2bc + \overline{y}_2\overline{b} + \overline{y}_2\overline{a}\,\overline{c}) \\ & \cdot(y_3ac + y_3bc + \overline{y}_3\overline{c} + \overline{y}_3\overline{a}\overline{b}) \end{aligned}$$

Wir wollen $Im(f_1,f_2,f_3) = \exists_{a,b,c}T(a,b,c,y_1,y_2,y_3)$ berechnen:

$$\begin{aligned} \exists_a T &= T_a + T_{\overline{a}} \\ &= (y_1b + y_1c + \overline{y}_1\overline{b}\overline{c})(y_2b + y_2bc + \overline{y}_2\overline{b})(y_3c + y_3bc + \overline{y}_3\overline{c}) \\ &\quad + \overline{y_1}(y_2bc + \overline{y}_2\overline{b} + \overline{y}_2\overline{c})(y_3bc + \overline{y}_3\overline{c} + \overline{y}_3\overline{b}) \\ \exists_{a,b} T &= (\exists_a T)_b + (\exists_a T)_{\overline{b}} + \ldots \end{aligned}$$

Die Berechnung ist aufgrund der vielen Kofaktoren zu aufwendig. In dieser Form ist der Berechnungsweg mit der Übergangsrelation weniger effizient als mit der Übergangsfunktion. Wir können die Methode aber verbessern, so daß sie konkurrenzfähig wird.

Verallgemeinerte Kofaktoren und die Übergangsrelation

Im folgenden benutzen wir $\diamond$ als generischen Operator.

Lemma 5.4.9: Seien $f:\mathbb{B}^{m+n}\to\mathbb{B}, \quad g:\mathbb{B}^n\to\mathbb{B}, \quad x=(x_1,\ldots,x_n)$ und $y=(y_1,\ldots,y_m)$ Sei $\diamond$ ein Operator mit:

(1) $(f\diamond g)g = f\cdot g$ (Eigenschaft für verallgemeinerten Kofaktor)

(2) $\exists_x(f(x,y)\diamond g(x)) \le \exists_x(f(x,y)\cdot g(x))$

Dann gilt: $\exists_x(f(x,y)\diamond g(x)) = \exists_x(f(x,y)\cdot g(x))$.

Beweis:

$\le$: trivial

$\ge$: $(f\diamond g)g = f\cdot g$ nach Vor. (1)
$\Rightarrow f\diamond g \ge f\cdot g$
da $\exists_x$ monoton gilt: $\exists_x(f(x,y)\diamond g(x)) \ge \exists_x(f(x,y)g(x))$

□

Als nächstes wollen wir wissen, ob wir das Lemma auf die uns bereits bekannten Kofaktoroperatoren anwenden können.

Betrachten wir den *Constrain* Operator als Abbildung:

$$(f \downarrow g)(r) = \begin{cases} f(r), \text{ falls } g(r) = 1 \\ f(s), \text{ falls } g(r) = 0 \end{cases}$$

wobei für s gilt: $g(s) = 1$ und $\|r - s\|$ ist minimal.

Sei μ_g eine Abbildung mit

$$\mu_g(r) = \begin{cases} r, \text{ falls } g(r) = 1 \\ s, \text{ falls } g(r) = 0 \end{cases}$$

Dann gilt: $f \downarrow g(x) = f(\mu_g(x))$.

Sei $f : \mathbb{B}^{n+m} \to \mathbb{B}, \quad g : \mathbb{B}^n \to \mathbb{B}$ und g hänge nicht von den letzten m Variablen von f ab.

Sei μ^x die Einschränkung von μ auf die ersten n Variablen.

Dann gilt: $\mu_g((x, y)) = (\mu_g^x(x), y)$.

Satz 5.4.10 *Seien $f : \mathbb{B}^{n+m} \to \mathbb{B}$ und $g : \mathbb{B}^n \to \mathbb{B}$ Boolesche Funktionen, $x = (x_1, \ldots, x_n)$ und $y = (y_1, \ldots, y_m)$.*
Dann gilt:

$$\exists_x(f(x, y)g(x)) = \exists_x(f(x, y) \downarrow g(x)).$$

Beweis: Es genügt zu zeigen, daß für beliebiges y folgendes gilt:

$$\exists_x(f(x, y) \downarrow g(x)) \Rightarrow \exists_x(f(x, y)g(x))$$

Sei $f(\tilde{x}, \tilde{y}) \downarrow g(\tilde{x}) = 1$, dann folgt $f(\mu_g^x(\tilde{x}), \tilde{y}) = 1$ und $g(\mu_g^x(\tilde{x})) = 1$
nach Definition von $f \downarrow g$ und μ_g^x.

Daher kann $\mu_g^x(\tilde{x})$ für x gewählt werden und Lemma 5.4.9 kommt zur Anwendung. □

Der *Restrict* Operator ($\Downarrow$) hat analoge Eigenschaften wie der *Constrain* Operator:

Sei $\tilde{g}$ mit $f \Downarrow g = f \downarrow \tilde{g}$ ($\tilde{g}$ entstand aus g durch Abstraktion von einigen Variablen aus Kofaktoren von g). Nach Definition von *Restrict* gilt: $\tilde{g} \geq g$ und $supp(\tilde{g}) \subseteq supp(g)$.

Lemma 5.4.11: Sei $f : \mathbb{B}^{n+m} \to \mathbb{B}, \quad g : \mathbb{B}^n \to \mathbb{B}, \quad x = (x_1, \ldots, x_n)$, und $y = (y_1, \ldots, y_m)$.
Sei ferner $\tilde{g}$ mit $f \Downarrow g = f \downarrow \tilde{g}$ und $a = \{a_1, \ldots, a_n\}$, $a_i \in \{0, 1\}$.
Ist $\tilde{g}(a) > g(a)$, dann existiert ein $b \in B^n$ mit:

$$\tilde{g}(b) = g(b) = 1 \text{ und } f(a) = f(b).$$

Beweis: Sei

$$\hat{x}_i = \begin{cases} x_i & \text{falls } a_i = 1 \\ \overline{x_i} & \text{falls } a_i = 0 \end{cases}$$

$\tilde{g}$ wird durch existentielle Abstraktion einiger Variablen aus Kofaktoren von g gewonnen.

Aus $\tilde{g}(a) > g(a)$ folgt, daß es eine Menge von Variablen gibt, die auf den durch a charakterisierten Weg liegen und existentiell ausquantifiziert wurden.

Diese Menge sei $J = \{x_{i_1}, \ldots, x_{i_r}\}$. Sei $K = \{x_1, \ldots, x_n\} - J$ und sei $c := \prod_{i \in K} \hat{x}_i$.

Es muß gelten: $a \leq c$.

Behauptung: f_c und $\tilde{g}_c$ sind konstant.

Es gilt: $g = 1$ für wenigstens einen Minterm aus c, sonst gilt: $\tilde{g}(a) = g_c(a) = 0$.

Daher existiert mindestens ein Minterm $b \leq c$ mit $f(a) = f(b)$ und $g(b) = 1$. □

Mit Hilfe von Lemma 5.4.11 können wir Satz 5.4.10 analog für den *Restrict* Operator zeigen.

Satz 5.4.12 *Sei* $f : \mathbb{B}^{n+m} \to \mathbb{B}, \quad g : \mathbb{B}^n \to \mathbb{B}, \quad x = (x_1, \ldots, x_n)$ *und* $y = (y_1, \ldots, y_m)$. *Dann gilt:*

$$\exists_x (f(x,y)g(x)) = \exists_x (f(x,y) \Downarrow g(x)).$$

Beweis: Sei $\tilde{g}$ eine Boolesche Funktion mit $(f \Downarrow g)(x,y) = (f \downarrow \tilde{g})(x,y)$. Dann gilt

$$(f \Downarrow g)(\tilde{x}, \tilde{y}) = 1 \Leftrightarrow (f \downarrow \tilde{g})(\tilde{x}, \tilde{y}) = 1.$$

Analog zu Satz 5.4.8 folgt

$$f(\mu^x_{\tilde{g}}(\tilde{x}), \tilde{y}) = 1 \text{ und } \tilde{g}(\mu^x_{\tilde{g}}(\tilde{x})) = 1.$$

Sei $a = (\mu^x_{\tilde{g}}(\tilde{x}), \tilde{y})$
1. Fall: $g(a) = 1$, trivial
2. Fall: $g(a) = 0$, $\exists b : \tilde{g}(b) = g(b) = 1$ und $f(b) = f(a) = 1$ (nach Lemma 5.4.9) □

Bemerkung 5.4.13: Der Constrain-Operator ist distributiv über der Produktbildung. Im folgenden bezeichne $\circ$ die Funktionskomposition.

Satz 5.4.14 *Sei* $f : \mathbb{B}^m \to \mathbb{B}, \quad g : \mathbb{B}^n \to \mathbb{B}^m$ *und* $h : \mathbb{B}^n \to \mathbb{B}$. *Dann gilt:*

$$(f \circ g) \downarrow h = f \circ (g \downarrow h).$$

Beweis:

$$\begin{aligned} (f \circ g)(x) \downarrow h(x) &= (f \circ g)(\mu_h(x)) \\ &= f \circ g(\mu_h(x)) \\ &= f \circ (g \downarrow h) \end{aligned}$$

□

Anwendung
Sei $f = g_1 \cdot g_2$, $g = (g_1, g_2)$ mit $g : \mathbb{B}^n \to \mathbb{B}^2$, $f : \mathbb{B}^2 \to \mathbb{B}$, dann gilt:

$$(f \circ g) \downarrow h = g_1 g_2 \downarrow h.$$

Ebenfalls gilt:

$$(f \circ g) \downarrow h = f \circ (g \downarrow h) = f \circ (g_1 \downarrow h, g_2 \downarrow h) = (g_1 \downarrow h)(g_2 \downarrow h)$$

$$\Rightarrow \exists_x[(\prod_{i=1}^{m}(y_i \equiv f_i(x_1, \ldots, x_n))) \cdot g(x_1, \ldots, x_n)] =$$

$$\exists_x[\prod_{i=1}^{m}(y_i \equiv f_i(x_1, \ldots, x_n) \downarrow g(x_1, \ldots, x_n))].$$

Achtung: Diese Aussage gilt nicht für den *Restrict*-Operator, da für diesen keine Distributivität bzgl. Produktbildung gilt.

Beispiel 5.4.15: Sei $g_1 = x_2, \quad g_2 = x_1\overline{x_2}$ und $h = \overline{x_1} + \overline{x_2}$. Dann gilt $(g_1 \cdot g_2) \Downarrow h = 0$, aber $(g_1 \Downarrow h) \cdot (g_2 \Downarrow h) = x_1 x_2$.

Bemerkung 5.4.16: Man kann zeigen:

$$\exists_x[(\prod_{i=1}^{m}(y_i \equiv f_i(x_1, \ldots, x_n))) \cdot g(x_1, \ldots, x_n)] =$$

$$\exists_x[(\prod_{i=1}^{m}(y_i \equiv f_i(x_1, \ldots, x_n))) \Downarrow g(x_1, \ldots, x_n)] =$$

$$\exists_x[(\prod_{i=1}^{m}(y_i \equiv f_i(x_1, \ldots, x_n)) \Downarrow g(x_1, \ldots, x_n)) \cdot g(x_1, \ldots, x_n)]$$

Wir können also **nicht** gleichzeitig *Restrict* auf die Produktterme anwenden und diese (wie bei *Constrain* eleminieren.

Partitionierte Bildberechnung

Problematisch bei der Bildberechnung mit der Übergangsrelation ist die Produktbildung:

$$\prod_{i=1}^{m}(y_i \equiv f_i).$$

Selbst wenn die einzelnen OBDDs für $y_i \equiv f_i$ klein sind, kann ihr Produkt die vorhandenen Ressourcen schnell übersteigen, obwohl das endgültige Ergebnis nach der Abstraktion wieder sehr klein sein kann. Eine effizientere Berechnung ist möglich mit:

Lemma 5.4.17: Sei $f = f(y_1, \ldots, y_m, x_1, \ldots, x_n) : \mathbb{B}^{m+n} \rightarrow \mathbb{B}$
und sei $g = g(y_1, \ldots, y_m, x_i, \ldots, x_n) : \mathbb{B}^{m+n-i+1} \rightarrow \mathbb{B}$, mit $1 < i \leq n$.
Dann gilt:

$$\exists_{x_1,\ldots,x_n}(f \cdot g) = \exists_{x_i,\ldots,x_n}(\exists_{x_1,\ldots x_{i-1}}(f) \cdot g).$$

Beispiel 5.4.18: Wir wollen $Im(f,g)$ berechnen mit $f = (f_1, f_2, f_3)$,

$$\begin{aligned} f_1 &= x_1 + x_2 \\ f_2 &= \overline{x_2} + x_3 \\ f_3 &= x_2 x_4 + \overline{x_3} \end{aligned}$$

und $g(x_1, x_2, x_3, x_4) = x_1 + x_2$.

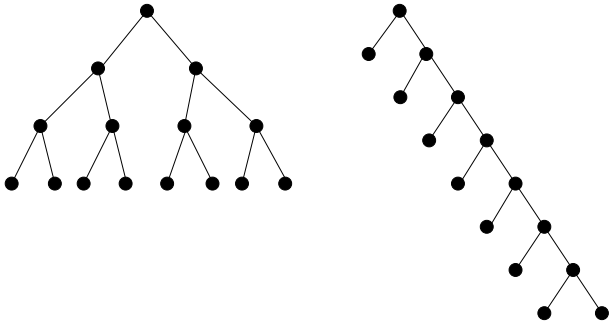
$$\begin{aligned} \mathrm{Im}(f,g) &= \exists_{x_1 x_2 x_3 x_4} \quad [(y_1(x_1 + x_2) + \overline{y}_1 \overline{x}_1 \overline{x}_2) \Downarrow (x_1 + x_2)) \\ &\qquad \cdot (y_2(\overline{x}_2 + x_3) + \overline{y}_2 x_2 \overline{x}_3) \Downarrow (x_1 + x_2)) \\ &\qquad \cdot ((y_3(x_2 x_4 + \overline{x}_3) + \overline{y}_3 x_3(\overline{x}_2 + \overline{x}_4) \Downarrow (x_1 + x_2)) \cdot (x_1 + x_2)] \\ &= \exists_{x_1 x_2 x_3 x_4} \quad [y_1(y_2(\overline{x}_2 + x_3) + \overline{y}_2 x_2 \overline{x}_3) \\ &\qquad \cdot y_3(x_2 x_4 + \overline{x}_3) + \overline{y}_3 x_3(\overline{x}_2 + \overline{x}_4) \cdot (x_1 + x_2)] \end{aligned}$$

Der zweite und der dritte Term der Übergangsrelation ist unabhängig von x_1. Daher quantifizieren wir x_1 aus $x_1 + x_2$ heraus, was die konstante Funktion 1 ergibt.

$$\begin{aligned} \mathrm{Im}(f,g) &= \exists_{x_2 x_3 x_4} \quad [y_1 \cdot (y_2(\overline{x}_2 + x_3) + \overline{y}_2 x_2 \overline{x}_3) \\ &\qquad \cdot (y_3(x_2 x_4 + \overline{x}_3) + \overline{y}_3 x_3(\overline{x}_2 + \overline{x}_4)) \\ &\qquad \cdot \underbrace{\exists_{x_1}(x_1 + x_2)}_{1}] \\ &= y_1 \exists_{x_2 x_3} \quad [\exists_{x_4}(y_3(x_2 x_4 + \overline{x}_3) + \overline{y}_3 x_3(\overline{x}_2 + \overline{x}_4)) \\ &\qquad \cdot (y_2(\overline{x}_2 + x_3) + \overline{y}_2 x_2 \overline{x}_3)] \\ &= y_1 \exists_{x_2 x_3} \quad [(y_3 \overline{x}_3 + \overline{y}_3 x_3 + y_3 x_2) \\ &\qquad \cdot (y_2(\overline{x}_2 + x_3) + \overline{y}_2 x_2 \overline{x}_3)] \\ &= y_1 \exists_{x_2 x_3} \quad [y_2 y_3 \overline{x}_2 \overline{x}_3 + \overline{y}_2 y_3 x_2 \overline{x}_3 + y_2 \overline{y}_3 \overline{x}_3 + y_2 y_3 x_2 x_3] \\ &= \qquad y_1(y_2 y_3 + \overline{y}_2 y_3 + y_2 \overline{y}_3) \\ &= \qquad y_1(y_2 + y_3). \end{aligned}$$

Wir wollen eine möglichst effiziente Berechnung unter Ausnutzung der Anwendung im vorherigen Abschnitt finden.

Wenn in jedem Schritt das Produkt zweier Faktoren gebildet wird, ergibt sich ein Binärbaum (Postorder-traversal). Dabei entsprechen dessen Blätter den Einzeltermen und dessen innere Knoten dem jeweiligen Produkt der beiden Söhne und der Ausquantifizierung aller Variablen im jeweiligen Ergebnis, die nicht im Rest des Baumes vorkommen. Es gibt viele verschiedene Bäume mit der gleichen Anzahl Blätter, z. B. die beiden Extreme des perfekt ausbalancierten Baumes und des total unbalancierten Baumes.



(1) Welcher Baum sichert eine effiziente Berechnung?
(Die Zahl der inneren Knoten hängt nur von der Zahl der Blätter (=Faktoren) ab)

(2) Wie werden Faktoren den Blättern zugeordnet?
Die Zuordnung sollte die Größe der Zwischenresultate minimieren:
Heuristiken:

- Ordnung wählen bzgl. Support der beteiligten Funktionen.
- So umordnen, daß möglichst viel frühzeitig quantifiziert werden kann.
- In jedem Knoten Kombination aus $\exists$ und AND berechnen ($\exists_c(f \cdot g)$)

Berechne: $\exists_c(f \cdot g)$ (AND-EXIST)
Eine rekursive Berechnung ist wie im Fall der Quantifizierung möglich:
Seien $f, g \in \mathbb{B}^n, \quad c = x_n\tilde{c}; \quad x_n$ ist die Topvariable von c und t der Index der Topvariable von $\{f, g\}$.
Abbruchbedingungen

1.) $c = 1$ oder $f \cdot g =$ konstant $\Rightarrow \exists_c(f \cdot g) = (f \cdot g)$

2.) $t > n$: In diesem Fall hängen f und g nicht von x_n ab. Daher gilt

$$\exists_c(f \cdot g) = \exists_{\tilde{c}}(f \cdot g).$$

$t = n$: Wegen $(f \cdot g) = ite(f, g, 0)$ gilt:

$$\begin{aligned} \exists_c(f \cdot g) &= \exists_{\tilde{c}}(\exists_{x_t} ite(x_t, ite(f_{x_t}, g_{x_t}, 0), ite(f_{\overline{x_t}}, g_{\overline{x_t}}, 0))) \\ &= \exists_{\tilde{c}}(ite(f_{x_t}, g_{x_t}, 0)) + \exists_{\tilde{c}}(ite(f_{\overline{x_t}}, g_{\overline{x_t}}, 0)) \\ &= ite(\exists_{\tilde{c}}(f_{x_t} \cdot g_{x_t}), 1, \exists_{\tilde{c}}(f_{\overline{x_t}} \cdot g_{\overline{x_t}})) \end{aligned}$$

Gilt $\exists_{\tilde{c}}(f_{x_t} g_{x_t}) = 1$, dann wird 1 zurückgegeben (Zweiter Aufruf nicht mehr nötig).

$t < n$: In diesem Fall erweitern wir nach x_t:

$$\begin{aligned} \exists_c(f \cdot g) &= \exists_c ite(x_t, ite(f_{x_t}, g_{x_t}, 0), ite(f_{\overline{x_t}}, g_{\overline{x_t}}, 0)) \\ &= ite(x_t, \exists_c(f_{x_t} \cdot g_{x_t}), \exists_c(f_{\overline{x_t}} \cdot g_{\overline{x_t}})) \end{aligned}$$

Wir können Ideen wie identische Subprobleme, Partitionierung oder Splitting auch in der Berechnung von $\exists_c(f \cdot g)$ verwenden.

Partitionierung arbeitet analog wie bei der Übergangsfunktion:

$$\exists_{xy}(f(x) \cdot g(y)) = \exists_x(f(x)) \cdot \exists_y(g(y)).$$

Identische Subprobleme :

Sei i-ter Term des Produkts der Form: $T_i = y_i A + \overline{y_i} B$ mit der Topvariablen y_i
Annahme: Es existieren keine weiteren y_k, $k \neq i$ in T_i.
(z. B. bei Beginn der Berechnung)

- T_i, T_j sind identisch, falls
  $A \cdot B = 0$ und $(T_j = y_j A + \overline{y_j} B$ oder $T_j = y_j \overline{B} + \overline{y_j}\overline{A})$
- T_i, T_j sind komplementär, falls
  $A \cdot B = 0$ und $(T_j = y_j \overline{A} + \overline{y}_j \overline{B}$ oder $T_j = y_j B + \overline{y}_j A)$;

$A \cdot B = 0$ kann mit ite-constant getestet werden;

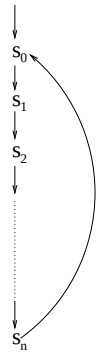
Sind T_i, T_j identisch, dann kann man $T_i \cdot T_j$ durch $y_i y_j + \overline{y}_i \overline{y}_j$ ersetzen.
Sind T_i, T_j komplementär, dann kann man $T_i \cdot T_j$ durch $y_i \overline{y}_j + \overline{y}_i y_j$ ersetzen.

5.5 Iteriertes Quadrieren

Mit breadth-first Travierserung können realistische FSMs mit 10^{120} Zuständen traversiert werden.

Ursache: Gleichzeitige Beurteilung vieler Zustände.

Probleme: FSM mit großer sequentieller Tiefe, z. B. Counter



Ausweg: Wir wenden die Technik des *iterierten Quadrierens* an.

Sei M eine FSM und $R(x, w, y)$ die Übergangsrelation von M.

Sei $(x, w, y) \in R$ falls Eingabe w aus Zustand x in Zustand y führt.

Behauptung: $T(x, y) = \exists_w R(x, w, y)$

$(s,t) \in T$ falls (s,t) eine Kante im *State Transition Graph* (STG) von M ist, d.h. falls eine Eingabe w existiert, die von s nach t führt.

Wir berechnen die *Transitive Hülle* $T^*(x,y)$ von $T(x,y)$.
$(s,t) \in T^*(x,y)$, falls der Weg von s nach t im STG von M existiert.

Sei S_0 Initialzustand.

$$Reached(y) := \exists_x(S_0(x) \cdot T^*(x,y)) + S_0(y),$$

d.h. wir müßen $T^*(x,y)$ berechnen.

Aus der Vorlesung *Informatik 2* ist bekannt:
T^* ist der kleinste Fixpunkt der Rekursion

$$T^*(x,y) = T(x,y) + \exists_z(T^*(x,z) \cdot T^*(z,y))$$

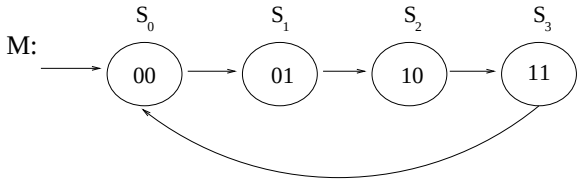
Wir wollen $T^*(x,y)$ berechnen:

$$\begin{array}{lcl} T_0^*(x,y) & := & T(x,y) \\ T_i^*(x,y) & := & T(x,y) + \exists_z(T_{i-1}^*(x,z) \cdot T_{i-1}^*(z,y)) \end{array}$$

Abbruch, falls $T_n^*(x,y) = T_{n-1}^*(x,y)$
$(s,t) \in T_i^*(x,y)$, falls ein Weg von s nach t im STG von M existiert mit der Länge $\leq 2^i$.

Bemerkung 5.5.1: Wir vermindern die Laufzeit des Verfahrens auf logarithmisch viele Schritte bzgl. der sequentiellen Tiefe.

Beispiel 5.5.2:



$$\begin{array}{lcl} T(x,y) & = & \overline{x_1}\,\overline{x_2}\,\overline{y_1}y_2 + \overline{x_1}\,x_2y_1\overline{y_2} + x_1\overline{x_2}\,y_1y_2 \\ T_0^*(x,y) & = & T(x,y) \\ T_1^*(x,y) & = & T(x,y) + \exists_{z_1z_2}[(\overline{x_1}\,\overline{x_2}\,\overline{z_1}z_2 + \overline{x_1}\,x_2z_1\overline{z_2} + x_1\overline{x_2}\,z_1z_2)\cdot \\ & & (\overline{z_1}\,\overline{z_2}\,\overline{y_1}y_2 + \overline{z_1}\,z_2y_1\overline{y_2} + z_1\overline{z_2}\,y_1y_2)] \\ & = & T(x,y) + \overline{x_1}\,\overline{x_2}\,y_1\overline{y_2} + \overline{x_1}\,x_2y_1y_2 \\ T_2^*(x,y) & = & T(x,y) + \exists_{z_1z_2}[(\overline{x_1}\,\overline{x_2}\,(\overline{z_1}\,z_2 + z_1\overline{z_2}) + \overline{x_1}\,x_2z_1 + x_1\overline{x_2}\,z_1z_2)\cdot \\ & & (\overline{z_1}\,\overline{z_2}\,(\overline{z_1}\,z_2 + y_1\overline{y_2}) + \overline{z_1}\,z_2y_1 + y_1\overline{z_2}\,y_1y_2)] \\ & = & T(x,y) + \overline{x_1}\,\overline{x_2}\,y_1 + \overline{x_1}\,\overline{x_2}\,y_1y_2 + \overline{x_1}\,x_2y_1 \\ & = & \overline{x_1}\,\overline{x_2}\,\overline{y_1}y_2 + \overline{x_1}\,x_2y_1\overline{y_2} + x_1\overline{x_2}\,y_1y_2 + \overline{x_1}\,\overline{x_2}\,y_1 + \overline{x_1}\,\overline{x_2}\,y_1y_2 \\ & = & \overline{x_1}\,\overline{x_2}(y_1 + y_2) + \overline{x_1}x_2y_1 + x_1\overline{x_2}\,y_1y_2 \end{array}$$

$$T_3^*(x,y) = T_2^*(x,y) = \overline{x_1}\,\overline{x_2}(y_1 + y_2) + \overline{x_1}x_2y_1 + x_1\overline{x_2}\,y_1y_2$$

Erreichbare Zustände:

$$
\begin{array}{lcl}
00 & \rightarrow & 1. = \{10, 11\} \\
00 & \rightarrow & 0. = \{01, 11\} \\
01 & \rightarrow & 1. = \{10, 11\} \\
10 & \rightarrow & 11
\end{array}
$$

Übungsaufgabe: $T^*(y, x) = \emptyset \Leftrightarrow$ Der STG ist zyklenfrei.

5.6 Approximative Erreichbarkeitsanalyse

Eine Erreichbarkeitsanalyse ist im Allgemeinen für einen Schaltkreis mit mehreren Hundert Latches sehr schwer. Oft reicht es aus, mit Ober- bzw. Untermengen erreichbarer Zustände zu rechnen.

(a) Berechnung einer Obermenge der erreichbaren Zustände:

Seien M, M' FSMs. M und M' sind äquivalent, falls für alle erreichbaren Zustände die gleiche Ausgabe produziert wird.

Gilt das auch für eine Obermenge der erreichbaren Zustände, dann sind M und M' erst recht äquivalent, d.h. der Äquivalenztest auf einer Obermenge der erreichbaren Zustände ist eine *konservative* Abschätzung.

- $\rightarrow$ Stimmen alle Ausgaben überein, so sind M und M' äquivalent.
- $\rightarrow$ Stimmen nicht alle Ausgaben überein, so muß überprüft werden, ob die jeweiligen nicht äquivalenten Zustände überhaupt erreichbar sind (*false negative*).

Das Ziel ist dabei, daß die gewählte Obermenge die Menge der erreichbaren Zustände möglichst gut approximiert.

(b) Berechnung einer Untermenge der erreichbaren Zustände:

Jeder Schritt der *forward* Traversierung liefert eine Approximation der Menge der erreichbaren Zustände (Oft kann die Traversierung aufgrund beschränkter Rechenzeit nicht abgeschloßen werden).

Erfolgt die Verifikation auf der Basis einer solchen Approximation, so bleibt diese Verifikation *partiell*, d.h liefert die Verifikation das Ergebnis *'FSM sind nicht äquivalent'*, so sind die beiden FSM tatsächlich nicht äquivalent.

Das Ergebnis *'FSM sind äquivalent'* ist nicht zuverlässig.

Bemerkung 5.6.1: Die partielle Verifikation ist ein sehr effizientes Verfahren zur Entdeckung von Entwurfsfehlern.

5.7 Erreichbarkeitsanalyse bei unvollständig spezifizierter FSM

Häufig tritt in der Praxis die folgende Situation auf: Die FSM M besitzt eine Zustandsübergangsfunktion bei der *Folgezustand* (*next state*) oder *Ausgabe* nicht definiert sind (*don't care*).

Wir beseitigen die *don't cares* durch Ersetzung mit einem neuen, bisher nicht vorkommenden Zustand d. Dann führt d für alle Eingaben zu d, d.h. die Ausgabe bleibt unspezifiziert. Alle zunächst unspezifizierten Übergänge führen auf d.

Bemerkung 5.7.1: Die Einführung des Zustands d ändert das Verhalten der FSM nicht.

Wir betrachten o.B.d.A. eine FSM mit partiell definierter Ausgabe.

Wir zerlegen die Ausgabefunktion λ von M in zwei Funktionen

$$\begin{array}{lll} \lambda, \xi : S \times I \to B^q & & \\ \xi_j(s,i) = 0 & \Rightarrow & \lambda_j(s,i) \text{ ergibt Ausgabe} \\ \xi_j(s,i) = 1 & \Rightarrow & \lambda_j(s,i) \text{ ist unspezifiziert.} \end{array}$$

Wir wollen die unvollständig spezifizierte FSM M_1 gegen die vollständig spezifizierte FSM M_2 verifizieren.

Für die Produktmaschine M_x gilt:

$$\lambda_x((s_1,s_2),i) = \prod_{k=1}^{q} (\xi_k(s_1,i) + (\lambda_{1k}(s_1,i) \equiv \lambda_{2k}(s,i)))$$

Es kann wie üblich weiter verfahren werden.

Achtung: Wir berechnen nicht mehr eine Äquivalenzrelation, sondern eine Überdeckung der Äquivalenzrelation ($Reached \leq \lambda_x$).

Sind beide Maschinen unvollständig spezifiziert, dann nehmen wir an, daß die Implementation (M_2) stärker spezifiziert ist als die Spezifikation (M_1) und checken zusätzlich $\xi_2 \leq \xi_1$.

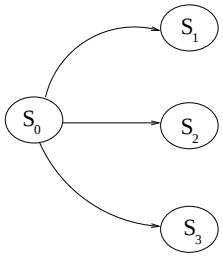
5.8 Erreichbarkeisanalyse für nondeterministische FSM

Bei der Verifikation von FSMs kommen auch nichtdeterministische FSMs vor:

- Modellierung asynchroner Schaltkreise (Verhalten oft nicht deterministisch berechenbar).
- Bei der Abstraktion von *unwichtigen* Details einer deterministischen Maschine kann oft eine starke Vereinfachung erreicht werden; oft geht dabei der Determinismus verloren

Daher müßen wir uns auch mit der Traversierung von nichtdeterministischen FSMs auseinandersetzen.

Nichtdeterministische Situation:



Eine nichtdeterministische FSM besitzt anstelle einer Zustandsübergangssfunktion eine Zustandübergangsrelation.

Wir können Algorithmen zur Traversierung, die auf der Übergangsrelation beruhen auch für nichtdeterministische FSM benutzen.

Achtung: Wir können die Transitionsrelation nicht schreiben als

$$\prod_{i=1}^{m} \delta_i(y_i, x_1 \ldots, x_n),$$

d. h. die Algorithmen, die auf partitionierter Bildberechnung beruhen sind in diesem Fall unbrauchbar.

Weitere Ideen: Wir überführen die nichtdeterministische FSM in eine deterministische durch Hinzunahme weiterer Variablen zur Auflösung der nichtdeterministischen Situationen.