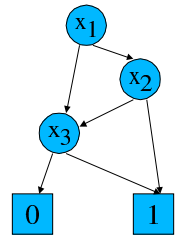
Binary Decision Diagrams



Binary Decision Diagrams

- Anwendungen in der kombinatorischen und sequentiellen Verifikation
- bieten eine kanonische Darstellung
  - Konstruktion des BDDs für Spezifikation und für Implementierung
  - Vergleich der BDDs in linearer (konstanter) Zeit
- bieten effiziente Algorithmen zur Manipulation
  - Boolesche Operationen, Quantifizierung, etc.

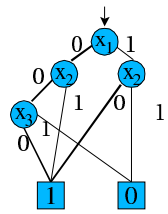
In diesem Kapitel...

- Definition von BDDs
- Algorithmen auf BDDs
  - Konjunktion, etc.
  - Symbolische Simulation
  - Implementierung von BDDs
- Optimieren der Variablenordnung
  - verschiedene Verfahren
- Verallgemeinerungen
  - KFDDs, WLDDs, LTBDDs

Frei verfügbare BDD-Packages

- CUDD-Package: Fabio Somenzi
  http://vlsi.colorado.edu/~fabio
  University of Colorado at Boulder.
- CAL-Package: R.K. Ranjan and J. Sanghavi
  http://www-cad.eecs.berkeley.edu/Research/cal_bdd/
  University of California, Berkeley.

Beispiel



beschreibt die Boolesche Funktion

$f(x_1, x_2, x_3) = x_1' x_2' x_3' + x_1' x_2 + x_1 x_2'$.

BDDs beruhen auf der Shannon-Expansion:

$f(x_1, ..., x_i, ..., x_n) = x_i' f_{xi=0}(x_1, ..., x_n) + x_i f_{xi=1}(x_1, ..., x_n)$.

Definition von BDDs: Syntax

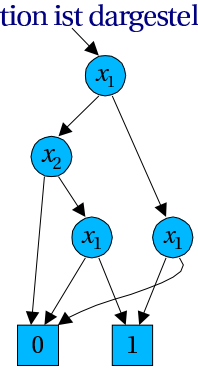
- Ein *binäres Entscheidungsdiagramm* (engl. Binary Decision Diagram, BDD) über einer Variablenmenge $X_n=\{x_1, ..., x_n\}$ ist ein gerichteter, azyklischer Graph $G = (V, E)$ mit Wurzel $v \in V$, wobei
  - G genau eine Wurzel und mindestens eine Senke hat
  - V aus terminalen und nichtterminalen Knoten besteht
  - jeder nichtterminale Knoten $v \in V$ mit einer Variablen $index(v) \in X_n$ markiert ist und zwei direkte Nachfolger $low(v) \in V$ und $high(v) \in V$ hat;
  - jeder terminale Knoten $t \in V$ einen Booleschen Wert $index(t) \in \{0,1\}$ als Marke hat und Senke ist.

Definition von BDDs: Semantik

- Ist B ein BDD über X_n, v ein Knoten von B, so ist die Boolesche Funktion $f_v : X_n \rightarrow \{0,1\}$, die durch ***v*** dargestellt wird, folgendermaßen definiert:
  - Ist v terminaler Knoten, so ist f_v die konstante Boolesche Funktion $index(v)$.
  - Ist v ein nichtterminaler Knoten mit $index(v)=x$, so ist f_v gleich $f_v = x' f_{low}(v) + x \times f_{high}(v)$.
- Ist ***w die Wurzel von B***, so ist die Funktion $f_B: X_n \rightarrow \{0,1\}$, die durch ***B*** dargestellt wird, gleich f_w.

Beispiel

- Welche Funktion ist dargestellt?



Schwierigkeiten bei BDDs

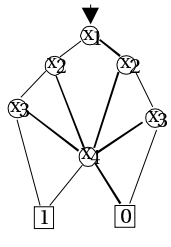
- Allgemeine BDDs gemäß der obigen Definition liefern keine kanonische Darstellung Boolescher Funktionen.
- Es ist unklar wie Operationen (*and, or, not, ...*) auf Booleschen Funktionen bei allgemeinen BDDs effizient realisiert werden können.
- Deshalb:
  Verwendung von **Einschränkungen:**
  - **Reduzierte geordnete Decision Diagrams (ROBDDs)**

Einschränkungen an BDDs

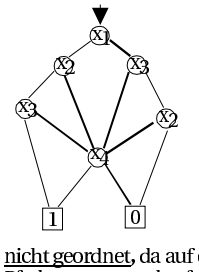
Definition [freie DD, FBDD]:
Ein BDD heißt **frei,** wenn auf jedem Pfad von der Wurzel zu einem terminalen Knoten jede Variable höchstens einmal als Markierung vorkommt.

Definition [ordered BDD, OBDD]
Ein FBDD über $\{x_1,...,x_n\}$ heißt **geordnet,** wenn es eine bijektive Abbildung $\pi:\{1,...,n\}\rightarrow\{x_1,...,x_n\}$ gibt, so dass für jede Kante (v,w) mit w nichtterminal die Ungleichung $\pi^{-1}(index(w)) > \pi^{-1}(index(v))$ gilt,
d.h. π gibt die Reihenfolge an, in der die Variablen auf jedem Pfad von der Wurzel zu einem Blatt abgefragt werden.

Beispiele für geordnete und nicht geordnete BDDs



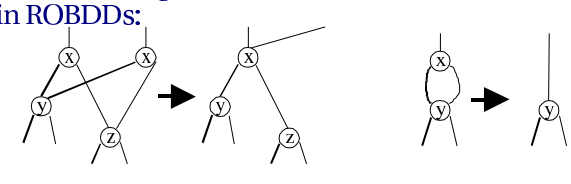
geordnet mit $\pi=(x_1,x_2,x_3,x_4)$



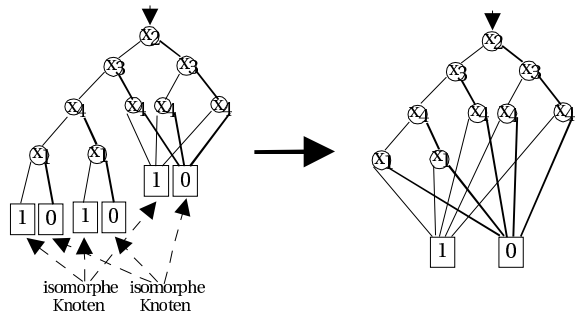
nicht geordnet, da auf einem Pfad x_2 vor x_3 und auf einem anderen Pfad x_2 nach x_3 steht

Reduzierte OBDDs (ROBDDs)

- Ein OBDD ist reduziert, wenn es
  - keine verschiedenen Knoten v und w gibt, die Wurzeln von isomorphen BDDs sind.
  - keinen nichtterminalen Knoten v mit low(v) = high(v) gibt.
- Reduktionsregeln zur Transformation von OBDDs in ROBDDs:

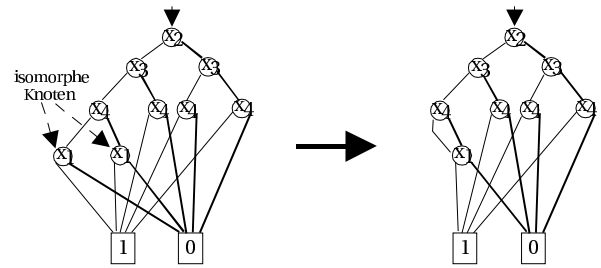


Reduzieren von OBDDs (1)

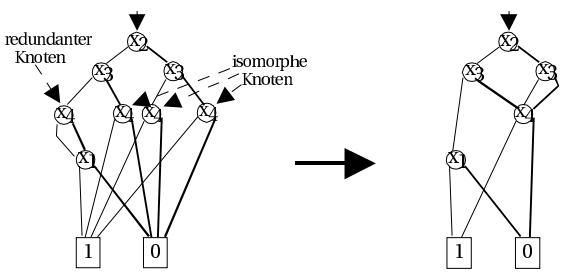


OBDDs können schichtweise (bottom up) reduziert werden

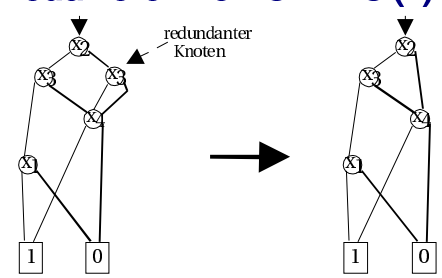
Reduzieren von OBDDs (2)



Reduzieren von OBDDs (3)



Reduzieren von OBDDs (4)



ROBDD

Anmerkung: Gängige Implementierungen von ROBDD-Paketen haben keine eigene Operation zum Reduzieren, sondern garantieren zu jedem Zeitpunkt, dass im gesamten System (evtl. aus mehreren ROBDDs bestehend) *nie* zwei identische Sub-Funktionen durch verschiedene ROBDD-Knoten repräsentiert werden

Kanonizität von ROBDDs

Satz (Bryant, 1986):
Sei π eine feste Variablenordnung.
Dann gibt es zu jeder Booleschen Funktion f (bis auf Isomorphie) genau einen ROBDD mit Variablenordnung π, der f beschreibt.

- Wesentliche Eigenschaft für die Verifikation: Sowohl für Spezifikation als auch für Implementierung werden die ROBDDs der entsprechenden Booleschen Funktionen (mit der selben Variablenordnung) berechnet.
- Äquivalenztest reduziert sich auf den Test, ob die entsprechenden ROBDDs isomorph sind.

Kanonizität von ROBDDs (2)

Beweis: Induktion nach Zahl n der Variablen, von denen f abhängt

Induktionsanfang: $n=0 \Rightarrow f$ ist konstant. O.B.d.A. sei $f = 0$.

Sei B ein ROBDD, der f repräsentiert und $k \geq 0$ innere Knoten hat.

Da $f_B = 0$, führt jeder Pfad von der Wurzel zu Terminal 0, d.h. Terminal 1 kommt in B nicht vor.

Wegen Reduziertheit gibt es *genau ein* 0-Terminal.

Annahme: $k \geq 1$.

Da B endlich und azyklisch ist, gibt es dann einen inneren Knoten, dessen beide Nachfolger Blätter, also Terminal 0 sind.

⇒ Widerspruch dazu, dass B reduziert ist!

Kanonizität von ROBDDs (3)

...

Induktionsvoraussetzung: Es gibt zu jeder Funktion mit < n Variablen (bis auf Isomorphie) genau einen ROBDD.

Induktionsschritt: *Annahme*: Zur Funktion f (mit n Variablen) gebe es 2 verschiedene, nicht isomorphe ROBDDs $B1$ und $B2$ mit Wurzeln $w1$ und $w2$ und $f_{B1} = f_{B2} = f$.

Sei $w1$ mit x_i beschriftet und $w2$ mit x_j beschriftet.

Fallunterscheidung $x_i \neq x_j$ und $x_i = x_j$.

Kanonizität von ROBDDs (4)

...

Fall 1: $x_i \neq x_j$. O.B.d.A. sei $\pi^{-1}(x_i) < \pi^{-1}(x_j)$.

$\Rightarrow f_{B2}$ ist unabhängig von $x_i \Rightarrow f = f_{B2}$ ist unabhängig von x_i.

Nach Def. gilt $f = f_{B1} = x_i' f_{low}(w1) + x_i f_{high}(w1)$ und somit $f_{xi}' = f_{low}(w1)$ und $f_{xi} = f_{high}(w1)$.

Da f unabhängig von x_i, gilt $f_{xi}' = f_{xi}$ und $f_{low}(w1) = f_{high}(w1)$

Nach Ind.vor. gibt es dazu (bis auf Isomorphie) genau einen ROBDD.

Wegen Reduziertheit von $B1$ ist deshalb $low(w1) = high(w1)$ und somit ist $w1$ ein redundanter Knoten. Widerspruch.

Kanonizität von ROBDDs (5)

...

Fall 2: $x_i = x_j$. D.h. *w1* und *w2* sind mit x_i beschriftet.

Analog zu obiger Argumentation:

$f_{low}(w1) = f_{xi'} = f_{low}(w2)$ und $f_{high}(w1) = f_{xi} = f_{high}(w2)$,
da $f = f_{B1} = f_{B2}$.

Damit nach Ind.vor.:
die ROBDDs mit Wurzeln *low*(*w1*) und *low*(*w2*) sind isomorph und
die ROBDDs mit Wurzeln *high*(*w1*) und *high*(*w2*) sind isomorph.

⇒ *B1* und *B2* sind isomorph.

Q.E.D. □

- Im Folgenden:
  - alle BDDs sind geordnet und reduziert
  - Variablenmenge ist X_n
- Annahme:
  Wir haben Funktionen f und g als BDD gegeben.
- Frage:
  Wie kann man damit rechnen, d.h. wie ist der BDD für
  $f \cdot g$, $f + g$, $f \oplus g$, etc.?

Konstruktion von BDDs

- **Bemerkung**: alle Booleschen Operatoren lassen sich durch den tenären Operator ITE(F, G, H) = F G + F' H zurückführen.
- Beispiel:
  AND(F, G) = ITE(F, G, 0),
  NOT(F) = ITE(F, 0, 1).
- ITE(F, G, H) = F G + F' H
  $= x\,(F\,G + F'H)_x + x'\,(F\,G + F'H)_{x'}$
  $= x\,(F_x\,G_x + (F_x)'\,H_x) + x'\,(F_{x'}\,G_{x'} + (F_{x'})'H_{x'})$
  $= x\,\mathrm{ITE}(F_x, G_x, H_x) + x'\,\mathrm{ITE}(F_{x'}, G_{x'}, H_{x'})$

Basisalgorithmus für ITE (1)

- ITE(F, G, H)
  $= x'\,\mathrm{ITE}(F_{x'}, G_{x'}, H_{x'}) + x\,\mathrm{ITE}(F_x, G_x, H_x)$
  - Berechne rekursiv ROBDDs für $\mathrm{ITE}(F_{X'},G_{X'},H_{X'})$ und $\mathrm{ITE}(F_X,G_X,H_X)$.
  - Falls $\mathrm{ITE}(F_X,G_X,H_X) = \mathrm{ITE}(F_{X'},G_{X'},H_{X'})$, so ist $\mathrm{ITE}(F,G,H) = \mathrm{ITE}(F_X,G_X,H_X) = \mathrm{ITE}(F_{X'},G_{X'},H_{X'})$.
  - Ansonsten: Generiere einen neuen Knoten v mit *index*(v)=x, dessen *low*-Kante auf $\mathrm{ITE}(F_{X'},G_{X'},H_{X'})$ zeigt und dessen *high*-Kante auf $\mathrm{ITE}(F_X,G_X,H_X)$ zeigt, (falls ein solcher Knoten nicht schon existiert).

Basisalgorithmus für ITE (2)

```
node* ITE(node* F, node* G, node* H) {
  /* Terminalfälle */
  if (F == 1 || G == H) return G;
  if (F == 0) return H;
  /* Rekursive Aufrufe */
  x = top_variable_of(F, G, H);
  low  = ITE(F_x', G_x', H_x');
  high = ITE(F_x, G_x, H_x);
  if (low == high) return low;
  R = unique_table_find_or_add(x, low, high);
  return R;
}
```

Die 3 Operatoren müssen die gleiche Variablen-ordnung haben

Exponentielle Laufzeit in der Anzahl der Variablen (zwei rekursive Aufrufe!)

Beschleunigung durch Cache

```
node* ITE(node* F, node* G, node* H) {
  /* Terminalfälle */

  if (computed_table_has_entry(F, G, H))
    return result;

  R = /* Berechne Ergebnis */;

  computed_table_insert(F, G, H, R);
  return R;
}
```

In der „computed table“ werden alle Resultate der bisherigen ITE-Operationen gespeichert
⇒
Anzahl der Aufrufe von ITE beschränkt durch die Anzahl der möglichen Tripel (F, G, H)

Laufzeit des ITE-Operators

- Sei |F| die Zahl der Knoten im BDD für F.
- **Lemma**:
  Die Laufzeit von ITE(F, G, H) ist in O(|F| |G| |H|).
- **Korollar**:
  Die Laufzeit des logischen AND(F, G) ist in O(|F| |G|).

Unique Table

- Knoten eines BDDs werden in einer Hashtabelle gespeichert
  - Schlüssel: Index und ausgehende Kanten (Zeiger)
  - Vor dem Anlegen eines neuen Knoten wird nachgesehen, ob es schon einen mit gleicher Funktion gibt.
  - Dadurch wird Kanonizität gewährleistet
- Üblicherweise werden für jeden Index (für jedes Level) separate Hashtabellen verwendet

Computed Table

- Cache für Operationen, um zu verhindern, dass die gleiche Operation mehrfach berechnet werden muss
- Es ist nicht sinnvoll, *alle* Ergebnisse zu speichern (Speicherbedarf)
- Üblicherweise ist die Größe der Computed Table (ziemlich) fest gewählt
  - *Hashing mit Verlust*: Soll ein Eintrag neu eingefügt werden und ist die entsprechende Hash-Adresse schon belegt, so wird der alte Eintrag überschrieben.

Implementierung

- Struktur für Knoten:

```
struct node {
  variable     var; // Variable
  int          ref; // Referenzzähler
  struct node *low, *high; // Kanten
  struct node *next; // für Unique Table
};
```

- *ref*: Zahl der eingehenden Kanten

Komplementkanten

- Jeder Kante bekommt ein zusätzliches Bit
  - Das unterste Bit der Zeiger kann verwendet werden, da bei üblichen Rechnerarchitekturen nur gerade Speicheradressen verwendet werden
- Ist das Bit gesetzt, so ist die dargestellte Funktion zu invertieren
- Es genügt ein Terminalknoten

Kanonizität bei Komplementkanten

- Die Darstellung mit Komplementkanten ist (zunächst) nicht kanonisch.
- Die Kanonizität kann wiederhergestellt werden durch die Einschränkung, dass *low*-Kanten nicht komplementiert sein dürfen

$$\overline{x_i} f_{x_i=0} + x_i f_{x_i=1} = \overline{\overline{x_i} \overline{f_{x_i=0}} + x_i \overline{f_{x_i=1}}}$$

Kofaktor-Berechnung

- Berechnung von $f_{x_i=0} = f(\ldots, x_{i-1}, 0, x_{i+1}, \ldots)$
- Verwendung einer Computed Table
- Kofaktor nach x_i eines Knotens v ist
  - die *low*-Kante, wenn der Knoten mit x_i markiert ist,
  - der Knoten ($index(v)$, $(f_{low})_{xi=0}$, $(f_{high})_{xi=0}$), sonst


```
node* cofactor(node* f, variable v) {
    /* Terminalfälle */
    if (f konstant) return f;
    x := index(f);
    if (level(x) > level(v)) return f;   // f hängt nicht von v ab
    /* Überprüfen der Computed Table */
    if (computed table has entry (f, v, result)) return result;
    if (x == v)       result = f_low;
          else {      res0 = cofactor(f_low, v);
                      res1 = cofactor(f_high, v);
                      result = unique_find_or_add(x, res0, res1);
    }
    insert (f, v, result) into computed_table;
    return result;
}
```


Überblick über Operationen

Operation	Komplexität
Auswertung	O(n)
Erfüllbarkeit	O(1)
Äquivalenz	O(1)
Negation*	O(1)
AND/OR/XOR	O(|F| |G|)
Kofaktor	O(|F|)
Quantifizierung	O(|F|²)

* mit Komplementkanten, sonst O(|F|)

Symbolische Simulation (1)

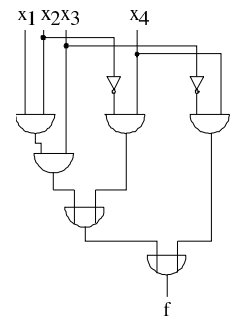
- Gegeben: Kombinatorische Schaltung
- Gesucht: BDD für die Boolesche Funktion, die die Schaltung realisiert

- Anwendungen:
  - Äquivalenzvergleich
    - Wenn ich für zwei Schaltungen den BDD konstruiert habe, kann ich in konstanter Zeit sagen, ob sie die gleiche Funktion berechnen
  - ... und viele andere

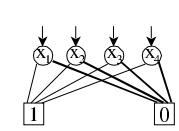
Symbolische Simulation (2)

- Beginne bei den Eingängen, erzeuge für jeden Eingang den BDD für die entsprechende Eingangsvariable.
- Durchlaufe die Schaltung in *topologischer Reihenfolge* und berechne mit Operationen die BDDs für die an den einzelnen Gattern berechneten Funktionen.
- Resultat: BDD der realisierten Booleschen Funktion
- *Symbolische Simulation* (im Gegensatz zur Simulation einzelner Eingangsbelegungen)

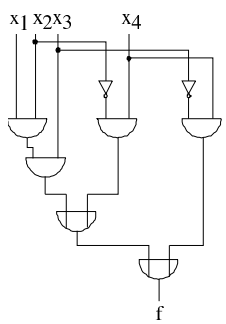
Symbolische Simulation (3)



1. Generiere BDDs für die Variablen x_1, ..., x_4: bdd_{x1}, ..., bdd_{x4}



Symbolische Simulation (4)



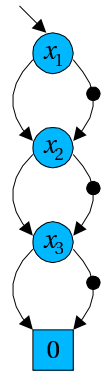
1. $bdd_{12} := AND(bdd_{x1}, bdd_{x2})$
2. $bdd_{123} := AND(bdd_{12}, bdd_{x3})$
3. $bdd_{2'} := NOT(bdd_{x2})$
4. $bdd_{2'4} := AND(bdd_{2'}, bdd_{x4})$
5. $bdd_{123+2'4} := OR(bdd_{123}, bdd_{2'4})$
6. $bdd_{3'} := NOT(bdd_{x3})$
7. $bdd_{3'4} := AND(bdd_{3'}, bdd_{x4})$
8. $bdd_f := AND(bdd_{123+2'4}, bdd_{3'4})$

Die Größe des BDDs

- Die Größe des BDDs einer Funktion hängt von der Funktion ab
  - Es gibt Funktionen, für die der BDD lineare Größe hat
    - Beispiel: Die XOR-Funktion
  - Es gibt Funktionen, für die der BDD exponentiell groß ist
    - Beispiel: Die Multipliziererfunktion
  - „Fast alle“ BDDs haben exponentielle Größe
    - Folgt aus Satz von Shannon (Fast alle Booleschen Funktionen haben eine Komplexität (= Anzahl von 2-Input-Gattern in optimaler Realisierung) > $2^n/n$) und der Tatsache, dass man BDDs als Multiplexerschaltkreise interpretieren kann

Die XOR-Funktion

- $f(x_1, x_2, x_3) = x_1 \oplus x_2 \oplus x_3$



Hidden Weighted Bit-Funktion

- Betrachte $\mathrm{hwb}_n(x_1,\ldots,x_n)=\begin{cases} 0 & s=0 \\ x_s & s>0 \end{cases}$

  mit $s=\sum_{i=1}^{n} x_i \in \mathbb{N}_0$

- Wie groß ist der BDD dafür unter der besten Variablenordnung?

Definition: Transfermenge

- Sei σ eine Belegung der Variablen aus $X_n := \{ x_1 \ldots x_n \}$ und $I \subseteq X_n$.
- σ lässt sich aufteilen in eine Belegung σ_I der Variablen in I und in eine Belegung $\sigma_{Xn\setminus I}$ der anderen Variablen.
- Eine Menge $\mathbf{F}(I)$ von Belegungen der Variablen aus I heißt **Transfermenge** (***fooling set***) für eine Boolesche Funktion f, wenn es für $\sigma_I, \sigma_I' \in \mathbf{F}(I)$ mit $\sigma_I \neq \sigma_I'$ eine Belegung σ_J der Variablen aus $\sigma_{Xn\setminus I}$ gibt, so dass $f(\sigma_I, \sigma_J) \neq f(\sigma_I', \sigma_J)$.

Definition: Balancierte Partition

- Sei $\omega \in (0, 1)$.
- Sei (L, R) eine Partitionierung von X_n, d.h. $L \cup R = X_n$ und $L \cap R = \emptyset$.
- Die Partition ist **balanciert**, wenn $\lfloor \omega \cdot |X_n| \rfloor \leq |L| \leq \lceil \omega \cdot |X_n| \rceil$.
  - Man kann sich auch auf eine Menge $Y \subseteq X_n$ von *Schlüsselvariablen* beschränken. Dazu muss man in obiger Definition $|X_n|$ durch Y ersetzen.

Untere Schranke für die Größe

Satz (Bryant, 1991):
Sei $f : B^n \to B$ und $\omega \in (0, 1)$.
Gibt es für jede balancierte Partition (L, R) bezüglich ω eine *Transfermenge* $\mathbf{F}(L)$ der Größe c, so hat jede BDD-Repräsentation von f mindestens c-2 Knoten.

- Sind die Variablen L im oberen Teil des BDDs, muss jede Belegung von $\mathbf{F}(L)$ zu einem neuen BDD-Knoten zeigen.
- Für jede Variablenordnung gibt es eine balancierte Partition (L, R) bezüglich
- Beweis: Drechsler, Becker, Graph. Funktionsdarstellung.

Untere Schranke für HWB_n

Satz (Bryant, 1991):
Ein BDD für HWB_n benötigt $\Omega(2^{n/10})$ Knoten.

Beweis:
Sei n ein Vielfaches von 10 (o. B. d. A.).
Sei (L, R) eine balancierte Partition bzgl. $\omega = 0{,}6$.
Zu zeigen: $\exists$ Transfermenge $\mathbf{F}(L)$ der Größe $\Omega(2^{(0,1 \cdot n)})$.
Seien X_H und X_L Teilmengen von X_n mit

$$X_H = \{x_{0,9n}, \ldots, x_{0,5n+1}\} \qquad X_L = \{x_{0,5n}, \ldots, x_{0,1n+1}\}$$

Da $|L| = 0{,}6\,n$ und $|X_H| + |X_L| = 0{,}8\,n$, muss entweder
$|X_H \cap L| \geq 0{,}2\,n$ oder $|X_L \cap L| \geq 0{,}2\,n$ sein.
Es gibt ein W mit $|W| = 0{,}2\,n$ und $W \subseteq X_H \cap L$ oder $W \subseteq X_L \cap L$.

Untere Schranke für HWB_n (2)

Fall 1: $W \subseteq X_H \cap L$.
$\mathbf{F}(L)$ wird so definiert, dass
$\sigma \in \mathbf{F}(L) \Leftrightarrow \sigma$ setzt die Hälfte der Variablen in W und alle Variablen aus $L - W$ auf 1.
In L sind dann genau $0{,}5\,n$ Variablen mit Wert 1.
$\mathbf{F}(L)$ ist Transfermenge: Seien $\sigma_1, \sigma_2 \in \mathbf{F}(L)$, $\sigma_1 \neq \sigma_2$.
Aus Def. von $\mathbf{F}(L)$ folgt: $\exists\, x_i \in W$: $\sigma_1(x_i) \neq \sigma_2(x_i)$.
Wegen $W \subseteq X_H$ ist $0{,}5n < i \leq 0{,}9n$.
Sei σ_R eine Belegung der Variablen aus R, die $(i - 0.5n)$ Variablen auf 1 setzt.
Da $0.5n$ Variablen aus L mit 1 belegt sind, gilt
$HWB_n(\sigma_1 \circ \sigma_R) = \sigma_1(x_i) \neq \sigma_2(x_i) = HWB_n(\sigma_2 \circ \sigma_R)$.
Die Größe von $\mathbf{F}(L)$ ist $\binom{0.2n}{0.1n} = \Omega(2^{\frac{n}{10}})$

Untere Schranke für HWB_n (3)

Fall 2: $W \subseteq X_L \cap L$.
$\mathbf{F}(L)$ wird so definiert, dass
$\sigma \in \mathbf{F}(L) \Leftrightarrow \sigma$ setzt die Hälfte der Variablen in W und alle Variablen aus $L - W$ auf 0.
In L sind dann genau $0{,}1\,n$ Variablen mit Wert 1.
$\mathbf{F}(L)$ ist Transfermenge: Seien $\sigma_1, \sigma_2 \in \mathbf{F}(L)$, $\sigma_1 \neq \sigma_2$.
Aus Def. von $\mathbf{F}(L)$ folgt: $\exists\, x_i \in W$: $\sigma_1(x_i) \neq \sigma_2(x_i)$.
Wegen $W \subseteq X_L$ ist $0{,}1n < i \leq 0{,}5n$.
Sei σ_R eine Belegung der Variablen aus R, die $(i - 0.1n)$ Variablen auf 1 setzt.
Da $0.1n$ Variablen aus L mit 1 belegt sind, gilt
$HWB_n(\sigma_1 \circ \sigma_R) = \sigma_1(x_i) \neq \sigma_2(x_i) = HWB_n(\sigma_2 \circ \sigma_R)$.
Die Größe von $\mathbf{F}(L)$ ist $\binom{0.2n}{0.1n} = \Omega(2^{\frac{n}{10}})$ □

Exponentielle Größe

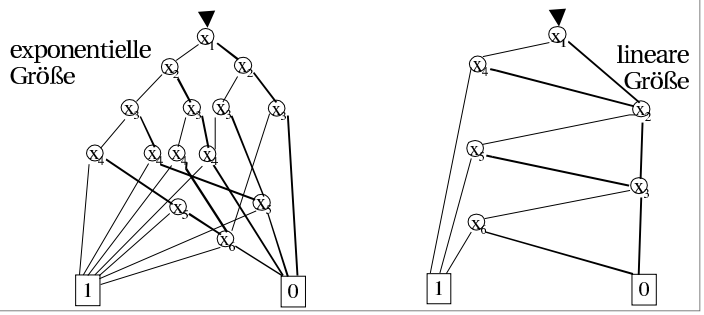
- Mit der gleichen Beweistechnik lässt sich beweisen, dass die Multipliziererfunktion

$$f(x_1,\ldots,x_n,y_1,\ldots,y_n) = bin((\sum_{i=1}^{n} x_i 2^{i-1}) * (\sum_{i=1}^{n} y_i 2^{i-1}))$$

  exponentiell groß ist.
  - Genauer: Um das (n-1)-te Ergebnisbit darzustellen, werden $\Omega(2^{(\frac{n}{8})})$ Knoten benötigt.
  - Im Gegensatz zur HWB hat diese eine hohe *praktische Relevanz*!

Einfluss der Variablenordnung

- Die Größe eine BDDs kann stark von der verwendeten Variablenordnung abhängen
  Beispiel: $f(x_1,\ldots,x_{2n}) = x_1x_{n+1} + x_2x_{n+2} + \ldots + x_nx_{2n}$



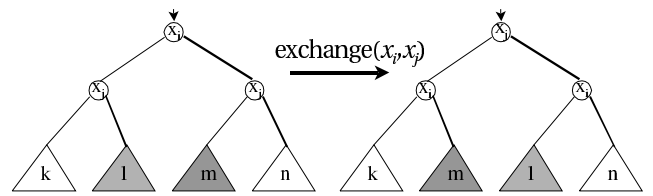
Verbessern der Variablenordnung

Satz (Bollig, Savicky, Wegener, 1994):
Das Problem, zu einem gegebenen BDD mit Variablenordnung π eine Variablenordnung π' zu finden, bei der die BDD-Größe minimal ist, ist NP-vollständig.

- Man unterscheidet
  - exakte Verfahren
  - initiale Verfahren (Ordnung wird aus Darstellung des Schaltkreises bestimmt)
  - dynamische Verfahren
    (Ordnung wird während der Berechnung verbessert)

Vertauschen benachbarter Variablen

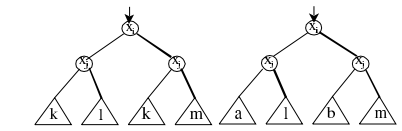
- Basisoperation für viele Verfahren, die mit einem BDD starten.



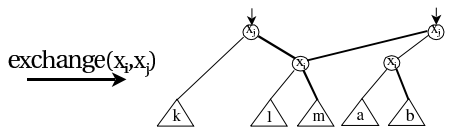
- Variablentausch ist eine lokale Operation und betrifft nur die Knoten, die mit den beiden Variablen beschriftet sind.

Vertauschen benachbarter Variablen (2)

- Die Größe kann sich verändern



mit *k, l, m, a, b* paarweise verschieden.



Vertauschen benachbarter Variablen (3)

- Es brauchen nur die Knoten der beiden betroffenen Variablen betrachtet werden
- ***Aber:*** Knoten sind in einer Hashtabelle (Unique Table) gespeichert
- ***Deshalb:*** Für jede Variable wird eine separate Unique Table verwendet
  - Bei *n* Variablen sind es nun *n* Unique Tables
  - Schlüssel der Hashtabelle braucht die Variable nicht mehr zu enthalten
  - Größen der Hashtabellen müssen dynamisch sein

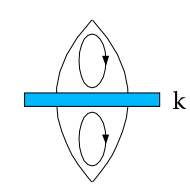
Exaktes Minimieren

- Es gibt Anwendungen, bei denen eine geringe Größe des BDDs sehr wichtig ist
  - Logik-Synthese:
    Der BDD wird direkt auf einen Multiplexer-Schaltkreis abgebildet
  - Evaluierung von Heuristiken
- Trivialer Ansatz:
  Konstruiere den BDD für alle n! Ordnungen

Exaktes Minimieren (2)

Satz (Friedman, Supowit, 1987):
Sei $f: B^n \rightarrow B$, $I \subseteq X_n$, $k = |I|$ und $x_i \in I$.
Dann gibt es $c \geq 0$, so dass $\#nodes_{xi}(f, \pi) = c$
für jedes $\pi \in \Pi(I)$ mit $\pi(k) = x_i$,
wobei $\Pi(I) = \{ \pi : \pi(x_j) \in I \quad \forall x_j \in I \}$.

- Knoten in Level *k* bleiben unverändert bei Änderungen im oberen / unteren Bereich
- Folgt aus Lokalität des Tausches benachbarter Variablen



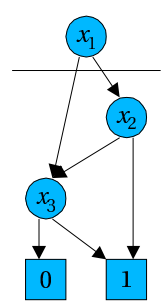
Exaktes Minimieren (3)

- Dynamische Programmierung
- Der BDD wird von oben her aufgebaut
- Sei $I \subseteq X_n$ mit $|I| = k$.
  - Angenommen, die optimale Ordnung sei für alle $I' \subset I$ mit $|I'| = k$-1 bekannt.
  - Dann kann die optimale Ordnung für I berechnet werden, indem für alle $x_i \in I$
    - die Variablen aus $I \setminus \{x_i\}$ in die oberen Level gebracht werden
    - Variable x_i nach Level k gebracht wird


```
compute_optimal_ordering(BDD f)
  insert (I = ∅, min_cost_∅ = 0, π_∅ = current order) into table;
  for (k = 1; k ≤ n; k++) {     // k : current level
    next_table = ∅;
    for each I ∈ table {
      set_permutation(π_I);
      for each x_i ∈ X_n {
        shift x_i to level k;
        I' = I ∪ { x_i }; π = current order;
        cost = min_cost_I + #nodes_xi(f, π);
        if (I' ∉ next_table or cost < min_cost_I')
          store (I', cost, π) into next_table;
    } }
    table = next_table;
  }
  set_permutation(best π);
```

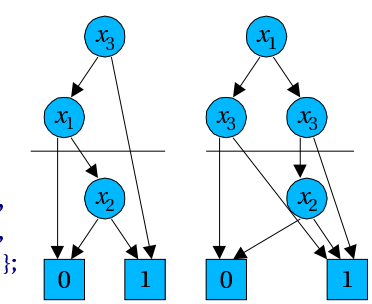

Beispiel: $f = x_1 x_2 + x_3$

- $table := \{ (I = \emptyset,$
  $min_cost_\emptyset = 0,$
  $\pi_\emptyset = (x_1, x_2, x_3)) \};$
- $k := 1;$
- $next_table := \{$
  $(\{ x_1 \}, 1, (x_1, x_2, x_3)),$
  $(\{ x_2 \}, 1, (x_2, x_1, x_3)),$
  $(\{ x_3 \}, 1, (x_3, x_2, x_1)) \};$



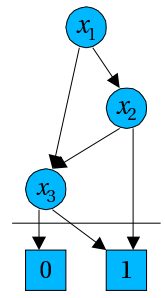
Beispiel: $f = x_1 x_2 + x_3$

- $table := \{$
  $(\{ x_1 \}, 1, (x_1, x_2, x_3)),$
  $(\{ x_2 \}, 1, (x_2, x_1, x_3)),$
  $(\{ x_3 \}, 1, (x_3, x_2, x_1)) \};$
- $k := 2;$
- $next_table := \{$
  $(\{ x_1, x_2 \}, 2, (x_1, x_2, x_3)),$
  $(\{ x_1, x_3 \}, 2, (x_3, x_1, x_2)),$
  $(\{ x_2, x_3 \}, 2, (x_3, x_2, x_1)) \};$



Beispiel: f = $x_1 x_2 + x_3$

- *table* := {
  ({ x_1, x_2 }, 2, (x_1, x_2, x_3)),
  ({ x_1, x_3 }, 2, (x_3, x_1, x_2)),
  ({ x_2, x_3 }, 2, (x_3, x_2, x_1)) };
- $k := 3$;
- *next_table* := {
  ({ x_1, x_2, x_3 }, 3, (x_1, x_2, x_3)) };



Exaktes Minimieren (5)

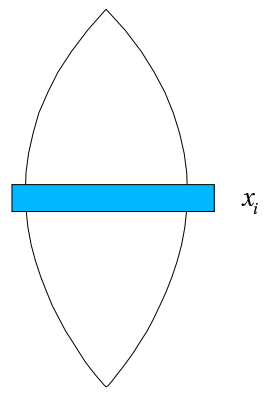
- Möglichkeiten der Verbesserung
  - Verwendung unterer Schranken
    - etwa Bryant, 1991
  - Verwendung von Symmetrien
    - Wenn $f(\ldots x \ldots y \ldots) = f(\ldots y \ldots x \ldots)$, dann kann o. B. d. A. angenommen werden, dass x vor y in der optimalen Ordnung steht.
- Für Funktionen mit bis zu ~25 Variablen anwendbar

Heuristische Verfahren

- Sifting (Rudell, 1993)
  - lower bound sifting (Drechsler, Günther, 1999)
- Group Sifting (Panda, Somenzi, 1995)
- Block-restricted Sifting (Meinel, Slobodová, 1997)
- Window Permutation
  - Anwendung des exakten Verfahrens auf „Fenster“ geringer Größe
  - Die „Fenster“ werden über den BDD verschoben
- Evolutionärer Algorithmus

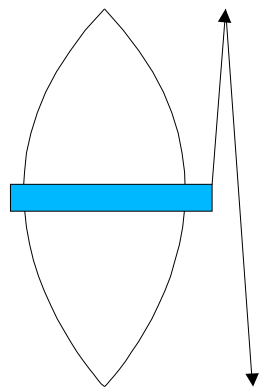
Sifting

- x_i := Variable des breitesten Levels



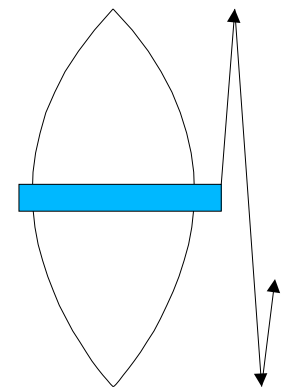
Sifting

- x_i := Variable des breitesten Levels
- Verschieben des Levels an alle Positionen



Sifting

- x_i := Variable des breitesten Levels
- Verschieben des Levels an alle Positionen
- Verschieben an die „beste“ Position
- Wiederhole dies für die restlichen Variablen



Sifting (2)

- Laufzeit: O(n^2) Vertauschungen von Variablen
  - Bei Funktionen mit >100 Variablen zu viel
- Verbesserungen:
  - Verschieben zum näheren Ende zuerst
  - Eine Variable wird nicht weiter in eine Richtung bewegt, wenn eine Maximalgröße überschritten ist
    - Beispiel: 1.2-faches der momentanen Größe
  - Verwendung unterer Schranken

Lower Bound Sifting

- SiftingDown(k : level to sift)

```
lb = compute_lower_bound(k);
best = size_of_BDD();
while (k < n and lb ≤ best) {
    level_exchange(k, k+1);
    lb = compute_lower_bound(k);
    if (size_of_BDD() < best)
        best = size_of_BDD();
    k = k + 1;
}
```

Varianten von BDDs

- Bit-Ebene: $f: B^n \rightarrow B^m$
  - Andere Dekompositionstypen (FDDs, KFDDs)
  - Weniger restriktive Darstellungen (Freie BDDs, Read-k-times BDDs)
  - Transformationen (LTBDDs, ...)
  - Operationsknoten (Parity-BDDs, BEDs, ...)
- Wort-Ebene: $f: B^n \rightarrow Z^n$
  - MTBDDs, BMDs, *BMDs, K*BMDs, ...
  - Die Multiplikation kann mit *BMDs linearer Größe dargestellt werden

Functional DDs (FDDs)

- positive Davio-Dekomposition:
  $f(x_1, ..., x_i, ..., x_n) = f_{xi=0} \oplus x_i (f_{xi=0} \oplus f_{xi=1})$
- negative Davio-Dekomposition:
  $f(x_1, ..., x_i, ..., x_n) = f_{xi=1} \oplus x_i' (f_{xi=0} \oplus f_{xi=1})$
- Manipulation von FDDs
  - XOR-Operation bleibt polynomiell
  - AND, OR exponentiell im worst case
- Minimierung
  - Variablentausch kann analog zu BDDs erfolgen (⇨ Übungsaufgabe)

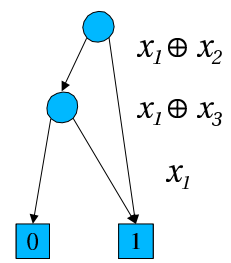
Kronecker Functional DDs (KFDDs)

- alle drei Dekompositionstypen
- Manipulation von KFDDs
  - wie bei FDDs
- Minimierung
  - Wahl der Variablenordnung und der Dekompositionstypen

Linear Transformierte BDDs

- Knoten haben statt *einer* Variablen die Parität einer Menge von Variablen
- Dies entspricht einer *linearen Transformation* $f'(x) = f(A(x))$

$f = (x_1 \oplus x_2) + (x_1 \oplus x_3)$



Linear Transformierte BDDs (2)

- Kanonische Darstellung
  - Für eine feste Variablenordnung *und* Transformation
- Ein exponentieller Unterschied ist möglich
  - Es gibt eine Funktion, die mit einem LTBDD linearer Größe dargestellt werden kann, für die aber jeder BDD exponentiell groß ist.
- Verändern der Transformation ist effizient
  - Ersetzen von x durch $x \oplus y$ für in der Variablenordnung benachbarte Variablen ist lokale Operation

Linear Transformierte BDDs (3)

- Techniken zur Minimierung:
  - Linear Sifting (Meinel, Somenzi, Theobald, 1997)
    - Erweiterung von Sifting um einen linearen Operator
    - auch hier können untere Schranken verwendet werden
  - Exakte Minimierung
    - Nur für sehr kleine Funktionen praktikabel
  - Window Optimization Algorithmus
  - Genetischer Algorithmus

Operatoren auf LTBDDs

- ITE-Operator kann wie für BDDs implementiert werden
  - $\mathrm{ITE}(f \circ A, g \circ A, h \circ A) = \mathrm{ITE}(f, g, h) \circ A$
- Es gibt effiziente Algorithmen für
  - Kofaktor-Berechnung
  - Quantifizierung
  - Erfüllbarkeitsprobleme

Word-Level DDs

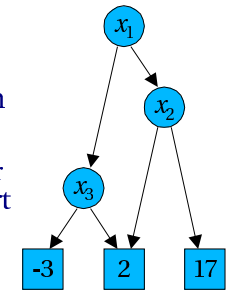
- Bit-level DDs können die **Multiplikation** nicht mit polynomieller Größe darstellen
  - Außer ...
- Arithmetische Schaltungen können auch durch **pseudo-Boolesche Funktionen** beschrieben werden
- Boolesche Werte werden mit 0, 1 identifiziert
  - Inverter: $f(x) = 1 - x$.

Dekompositionstypen für WLDDs

- Shannon: $(1 - x_i)\, f_{xi=0} + x_i f_{xi=1}$
- positiv Davio: $f_{xi=0} + x_i\,(f_{xi=1} - f_{xi=0})$
  - $f_{low} := f_{xi=0}$ heißt „konstanter Teil“
  - $f_{high} := f_{xi=1} - f_{xi=0}$ heißt „linearer Teil“
- negativ Davio: $f_{xi=1} + (1 - x_i)\,(f_{xi=0} - f_{xi=1})$

Multi-Terminal BDDs

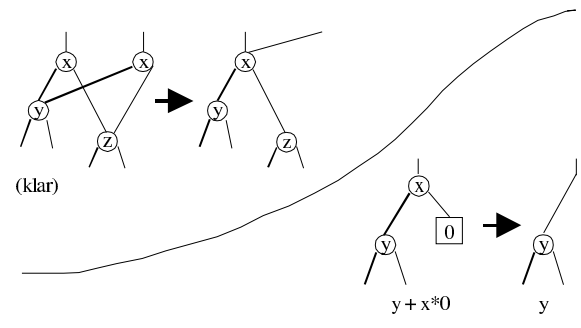
- Nur Shannon-Dekomposition
- Ganzzahlige Terminalknoten
- Kanonische Darstellung
- Minimierung und Manipulation ähnlich zu BDDs
  - Terminalknoten können in einer weiteren Hashtabelle gespeichert werden



Binary Moment Diagrams

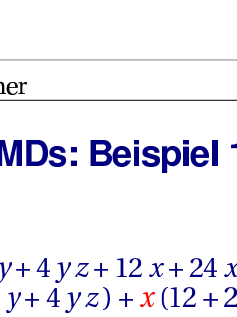
- Nur positiv Davio-Dekomposition
- Kanonische Darstellung
- Können Multiplizierer in quadratischer Größe darstellen

Reduktionsregeln für BMDs



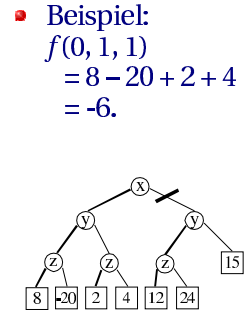
BMDs: Beispiel 1

- $f(x, y, z)$
  $= 8 - 20\,z + 2\,y + 4\,y\,z + 12\,x + 24\,x\,z + 15\,x\,y$
  $= (8 - 20\,z + 2\,y + 4\,y\,z) + x\,(12 + 24\,z + 15\,y)$
  $= ((8 - 20\,z) + y\,(2 + 4\,z)) + x\,((12 + 24\,z) + 15\,y)$



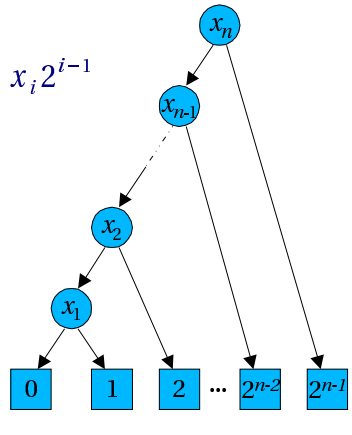
Auswertung von BMDs

- Summe aller *aktiven Pfade*
- Eine Kante ist „aktiv“, falls es
  - eine *low*-Kante ist
  - eine *high*-Kante ist und die Belegung der Variablen wahr ist
- Ein aktiver Pfad ist ein Pfad von der Wurzel zu einem terminalen Knoten, der nur aus aktiven Kanten besteht.
- Beispiel:
  $f(0, 1, 1)$
  $= 8 - 20 + 2 + 4$
  $= -6.$



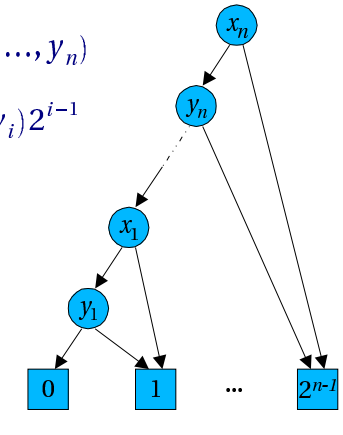
BMDs: Beispiel 2

$$f(x_1, \ldots, x_n) = \sum_{i=1}^{n} x_i 2^{i-1}$$



BMDs: Beispiel 3

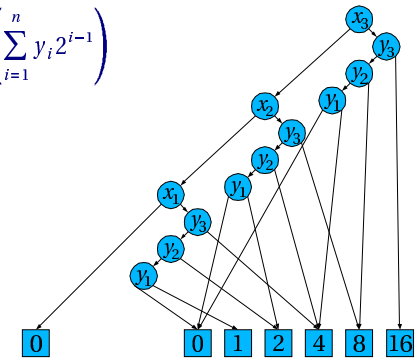
$$f(x_1, \ldots, x_n, y_1, \ldots, y_n) = \sum_{i=1}^{n} (x_i + y_i) 2^{i-1}$$



BMDs: Multiplizierer

$$f(x_1,\ldots,x_n,y_1,\ldots,y_n) = \left(\sum_{i=1}^{n} x_i 2^{i-1}\right) * \left(\sum_{i=1}^{n} y_i 2^{i-1}\right)$$

n=3:



Die Multiplikation lässt sich durch einen BMD mit n^2 inneren Knoten darstellen.

BMDs: Addition

- Gegeben: BMDs für f und g.
  Gesucht: BMD für $(f + g)$.
- $f + g = (f_{low} + x_i \cdot f_{high}) + (g_{low} + x_i \cdot g_{high})$
  $= (f_{low} + g_{low}) + x_i \cdot (f_{high} + g_{high})$

```
node* ADD(node* F, node* G) {
  check terminal cases
  x = top_variable_of(F, G);
  low = ADD(F_x', G_x');
  high = ADD(F_x, G_x);
  R = unique_table_find_or_add(x, low, high);
  return R;
}
```

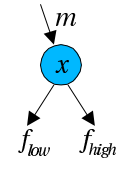

BMDs: Operationen

- Addition zweier BMDs F und G hat eine Laufzeit von O(|F| |G|)
  - wenn die Computed Table perfekt funktioniert
- Multiplikation zweier BMDs F und G kann exponentiellen Aufwand erfordern

Operation	BDD	BMD
Auswertung	O(n)	O(|F|)
Erfüllbarkeit	O(1)	O(1)
Äquivalenz	O(1)	O(1)
AND	O(|F| |G|)	exp.
ADD	--	O(|F| |G|)
MULT	--	exp.

Multiplicative Binary Moment Diagrams

- Bei *BMDs werden zusätzlich multiplikative Faktoren ≠ 0 an den Kanten zugelassen.



Die eingehende Kante des Knotens repräsentiert die Funktion $m * (f_{low} + x * f_{high})$.

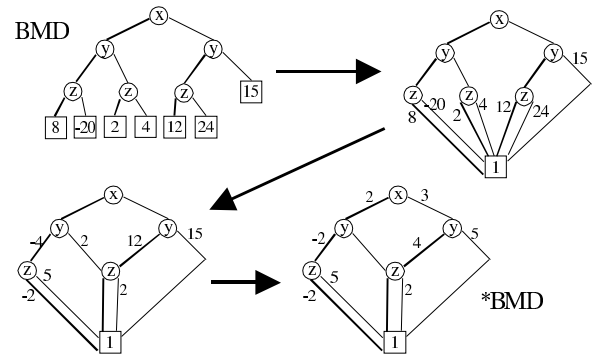
- Kanonizität?

Kanonizität von *BMDs

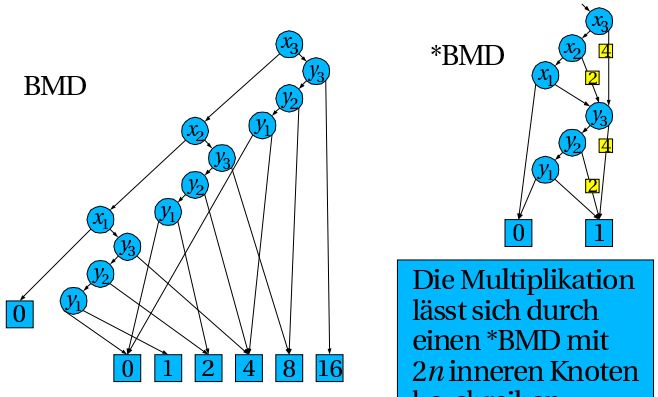
- *BMDs sind eine kanonische Darstellung, wenn
  (1) es als Terminalknoten höchstens 0 und 1 gibt,
  (2) alle multiplikativen Faktoren ungleich 0 sind,
  (3) der multiplikative Faktor von *high*-Kanten > 0 ist,
  (4) der größte gemeinsame Teiler der ausgehenden Kanten eines Knotens 1 ist.

*BMDs: Beispiel 1

- $f = 8 - 20\,z + 2\,y + 4\,yz + 12\,x + 24\,xz + 15\,xy$



Multiplikation mit *BMDs



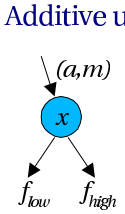
Die Multiplikation lässt sich durch einen *BMD mit $2n$ inneren Knoten beschreiben.

Verifikation mit *BMDs

- Vorteile:
  - weiterhin kanonische Darstellung
  - lineare Größe für alle „wichtigen“ arithmetischen Schaltungen
  - Für Boolesche Funktionen verhalten sich *BMDs in der Praxis ähnlich wie BDDs
- Nachteile:
  - Algorithmen zur Synthese können exponentiellen Aufwand haben
  - Multiplizierer kann nur auf Wort-Ebene dargestellt werden (Gruppierung der Ausgänge)

Kronecker Multiplicative BMDs

- Alle drei Dekompositionstypen
- Additive und multiplikative Kantengewichte



Die eingehende Kante des Knotens repräsentiert die Funktion

$a + m * ((1\text{-}x)\, f_{low} + x * f_{high})$ (Shannon)

$a + m * (f_{low} + x * f_{high})$ (pos. Davio)

$a + m * (f_{low} + (1 - x) * f_{high})$ (neg. Davio)

Vergleich BDDs vs. BMDs (1)

- BDDs: reine Bit-Level-Datenstruktur
  - Schaltung muss auf Bit-Ebene gegeben sein.
  - Nur Boolesche Operationen können auf BDDs berechnet werden
- *BMDs: Word-Level-Datenstruktur
  - Schaltung kann auf Word-Level gegeben sein, Operationen auf ganzen Zahlen können verwendet werden
  - Additionen und Multiplikationen können direkt auf *BMDs durchgeführt werden.

Vergleich BDDs vs. BMDs (2)

- BDDs
  - Multiplizierer benötigen exponentielle Größe
  - Boolesche Operationen benötigen in polynomiale Zeit
  - Vorwärtsaufbau
  - Standard
- *BMDs
  - Manchmal kann man exponentielle Blow-ups vermeiden.
  - (Boolesche) Operationen benötigen exponentielle Zeit
  - Rückwärtsaufbau
  - haben sich (noch???) nicht durchgesetzt