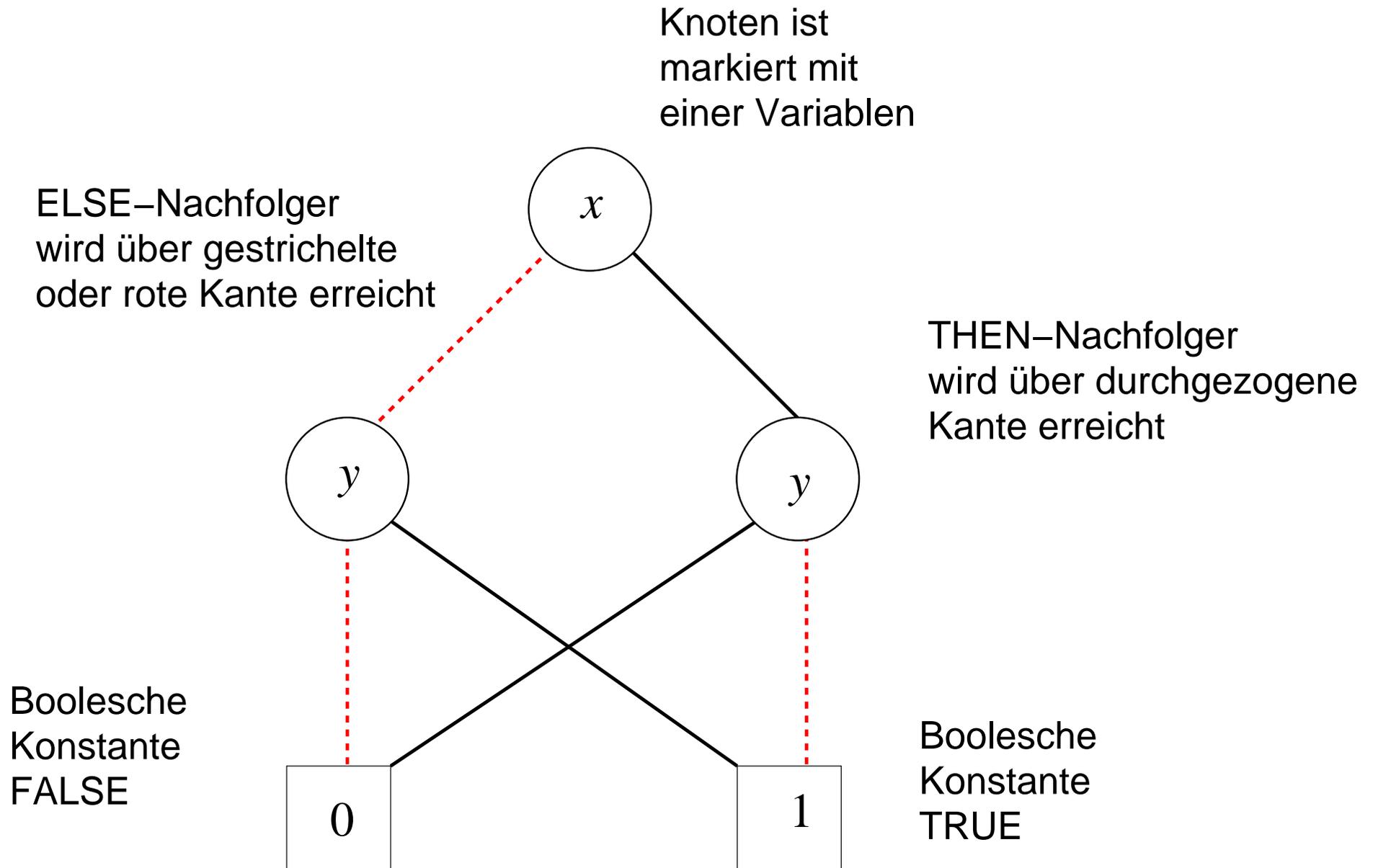


$$\bar{x} \cdot y \vee x \cdot \bar{y} \quad \equiv \quad (\bar{x} \vee \bar{y}) \cdot (x \vee y)$$

beide Formeln und AIGs beschreiben XOR von  $x$  und  $y$   
(Ausmultiplizieren der rechten Formel ergibt linke Formel)

i.Allg. gibt es mehrere AIGs für dieselbe boolesche Funktion

- neue dreistellige Basis-Operation ITE (if-then-else):  
Bedingung ist immer eine Variable
- gehen zurück auf Shannon (deshalb auch Shannon-Graphs)
- meist verwendete Version sind die ROBDDs  
Reduced Ordered Binary Decision Diagrams
- [Bryant86] hat Kanonizität von ROBDDs gezeigt:  
Jede Boolesche Funktion hat genau einen ROBDD

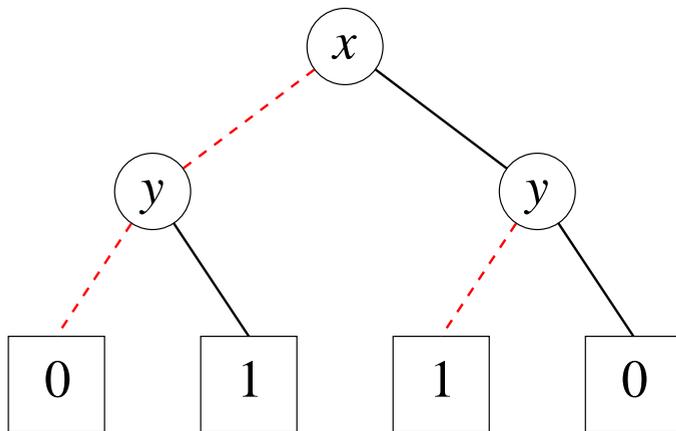


- innere Knoten sind ITE, Blätter sind boolesche Konstante
- Schreibweise  $ite(x, f_1, f_0)$  bedeutet *wenn  $x$  dann  $f_1$  ansonsten  $f_0$*   
(beachte: ELSE-Argument  $f_0$  kommt hinter  $f_1$  trotz umgekehrter Indizierung)
- Semantik  $eval$  produziert booleschen Ausdruck aus einem BDD

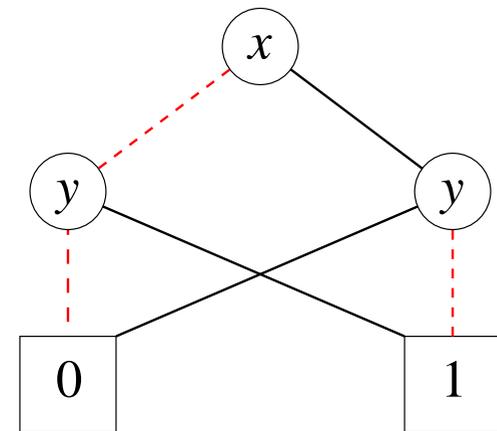
$$\begin{aligned}eval(0) &\equiv 0 \\eval(1) &\equiv 1 \\eval(ite(x, f_1, f_0)) &\equiv x \cdot eval(f_1) \vee \bar{x} \cdot eval(f_0)\end{aligned}$$

- BDDs sind auch wieder algebraisch reduzierte DAG's  
(max. Sharen von isomorphen Teil-Graphen wie bei AIGs)
- Negations-Kanten wie bei AIGs möglich

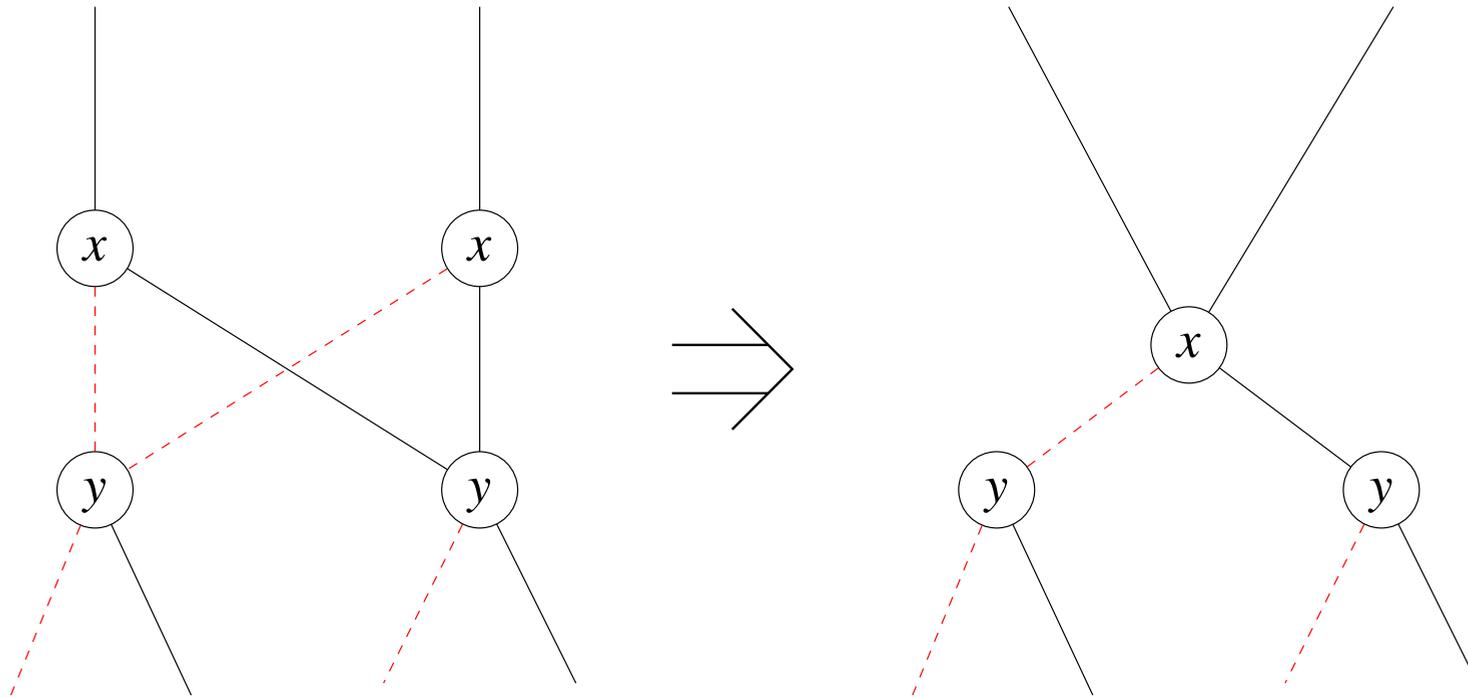
$x$	$y$	$x \oplus y$
0	0	0
0	1	1
1	0	1
1	1	0



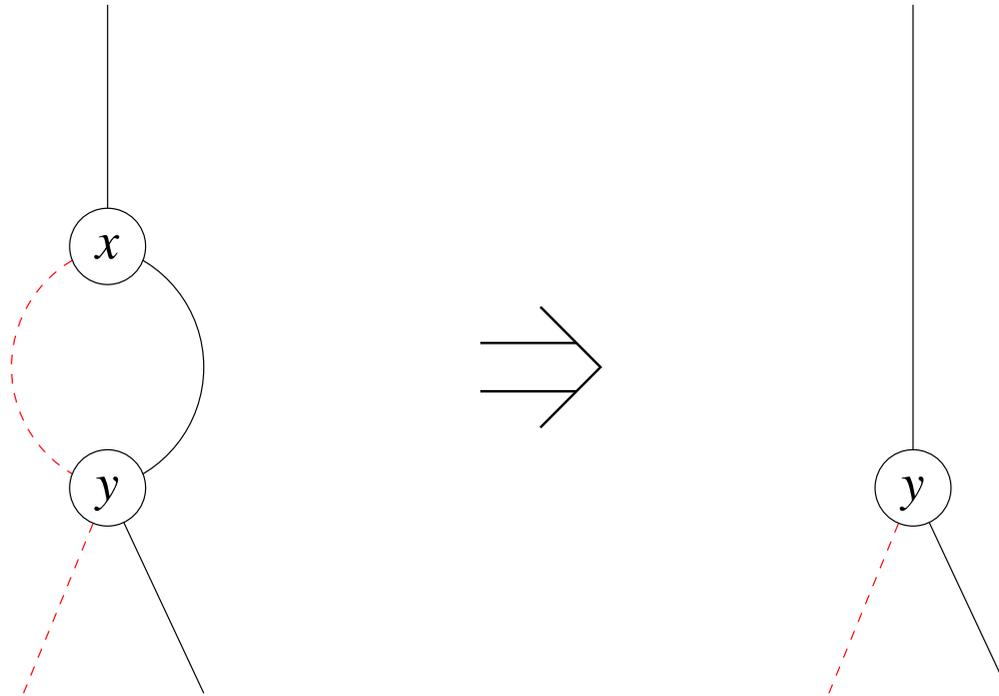
Entscheidungs-Baum



Entscheidungs-Diagramm  
(DAG)

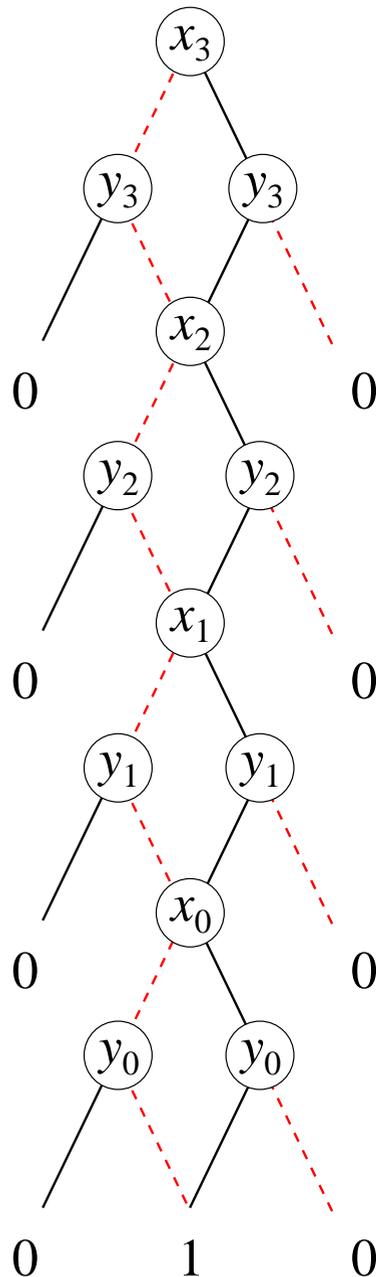


Maximales Sharen isomorpher Teil-Graphen



Elimination redundanter Innerer Knoten

- maximale algebraische und semantische Reduktion  
(das **Reduced** in ROBDDs)
- Variablen von der Wurzel zu den Blättern sind gleich geordnet  
(das **Ordered** in ROBDDs)
- diesen Annahmen machen BDDs kanonisch modulo Variablenordnung
- unterschiedliche Ordnungen führen i.Allg. zu unterschiedlichen BDDs
- Variablenordnung bestimmt essentiell die Grösse von BDDs
- **im weiteren meinen wir immer ROBDDs, wenn wir BDD sagen**



Boolesche Funktion:

$$\prod_{i=0}^{n-1} x_i = y_i$$

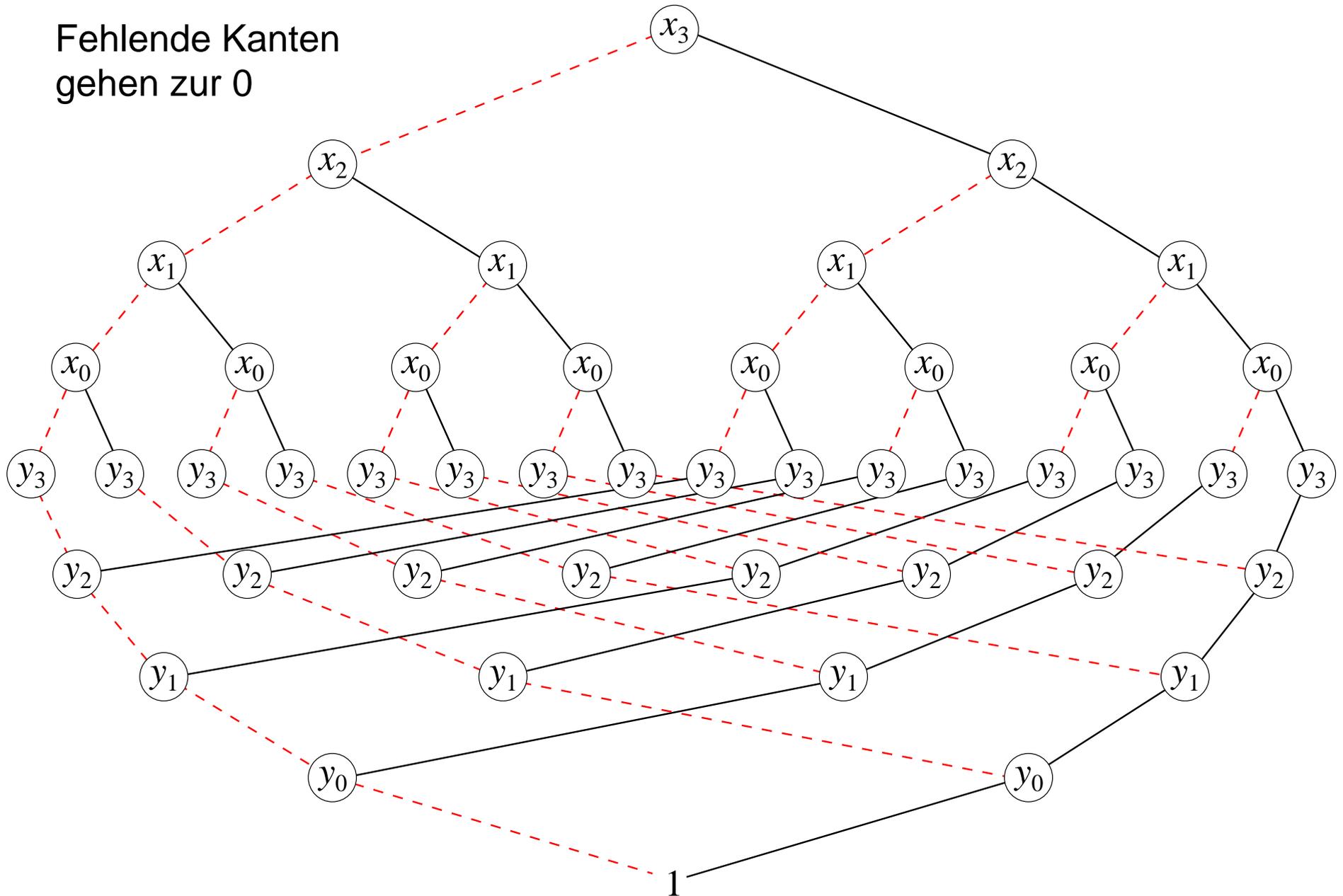
Verschränkte Variablen-Ordnung  
(interleaved)

$$x_3 > y_3 > x_2 > y_2 > x_1 > y_1 > x_0 > y_0$$

Allgemein:

Vergleich zweier  $n$ -Bit Vektoren braucht bei verschränkter Ordnung  $3 \cdot n$  innere Knoten.

Fehlende Kanten  
gehen zur 0



- es gibt exponentielle Unterschiede zwischen Variablen-Ordnungen
- es gibt aber leider auch *exponentielle* Funktionen:  
BDD ist immer exponentiell, z.B. mittlere Ausgabe-Bits von Multiplizierer
- es gibt Heuristiken zur statischen Variablen-Ordnung:  
meist einfache Auflistung der Variablen in DFS des DAG/AIG  
(etwa wie in `input_cone_aig`)
- zusätzlich gibt es noch *dynamische Umordnung von Variablen*  
basiert auf *in-place* Austausch von benachbarten Variablen

Kanonizität der BDDs für boolesche Funktionen ergibt:

- ein BDD ist eine Tautologie, gdw. er nur aus dem 1-Blatt besteht
- ein BDD ist erfüllbar, gdw. er nicht aus dem 0-Blatt besteht
- zwei BDDs sind äquivalent, gdw. die BDDs sind isomorph  
(werden BDDs wie AIGs in der gleichen Unique-Tabelle gespeichert, dann ist der Test auf Äquivalenz ein einfacher Pointer-Vergleich)

**Frage:** Wo ist die NP-Vollständig der Erfüllbarkeit geblieben?

**Antwort:** Versteckt sich im Aufwand der Erzeugung eines BDDs.

lautet wie folgt:

$$f(x) \equiv x \cdot f(1) \vee \bar{x} \cdot f(0)$$

Sei nun  $x$  die oberste Variable zweier BDDs  $f$  und  $g$ :

$$f \equiv \text{ite}(x, f_1, f_0) \qquad g \equiv \text{ite}(x, g_1, g_0)$$

mit  $f_i$  bzw.  $g_i$  die Kinder von  $f$  und  $g$  für  $i = 0, 1$ . Dann folgt

$$f(0) = f_0 \qquad g(0) = g_0 \qquad f(1) = f_1 \qquad g(1) = g_1$$

da nach dem **R** in **ROBDD**  $x$  nur ganz oben in  $f$  und  $g$  vorkommt.

$$\begin{aligned} (f@g)(x) &\equiv x \cdot (f@g)(1) \vee \bar{x} \cdot (f@g)(0) \\ &\equiv x \cdot (f(1)@g(1)) \vee \bar{x} \cdot (f(0)@g(0)) \\ &\equiv x \cdot (f_1@g_1) \vee \bar{x} \cdot (f_0@g_0) \end{aligned}$$

wobei  $@$  eine beliebige zweistellige boolesche Operation ist (z.B.  $\wedge$ ,  $\vee$ ,  $\oplus$ , ...)

**Rekursives Schema zur Berechnung von Operationen auf BDDs**

```
typedef struct BDD BDD;
```

```
struct BDD  
{  
    int idx, mark;  
    BDD *child[2], *next;  
};
```

```
#define sign_bdd(ptr) (1 & (unsigned) ptr)
```

```
#define strip_bdd(ptr) ((BDD*) (~1 & (unsigned) ptr))
```

```
#define not_bdd(ptr) ((BDD*) (1 ^ (unsigned) ptr))
```

```
#define true_bdd ((BDD*) 1)
```

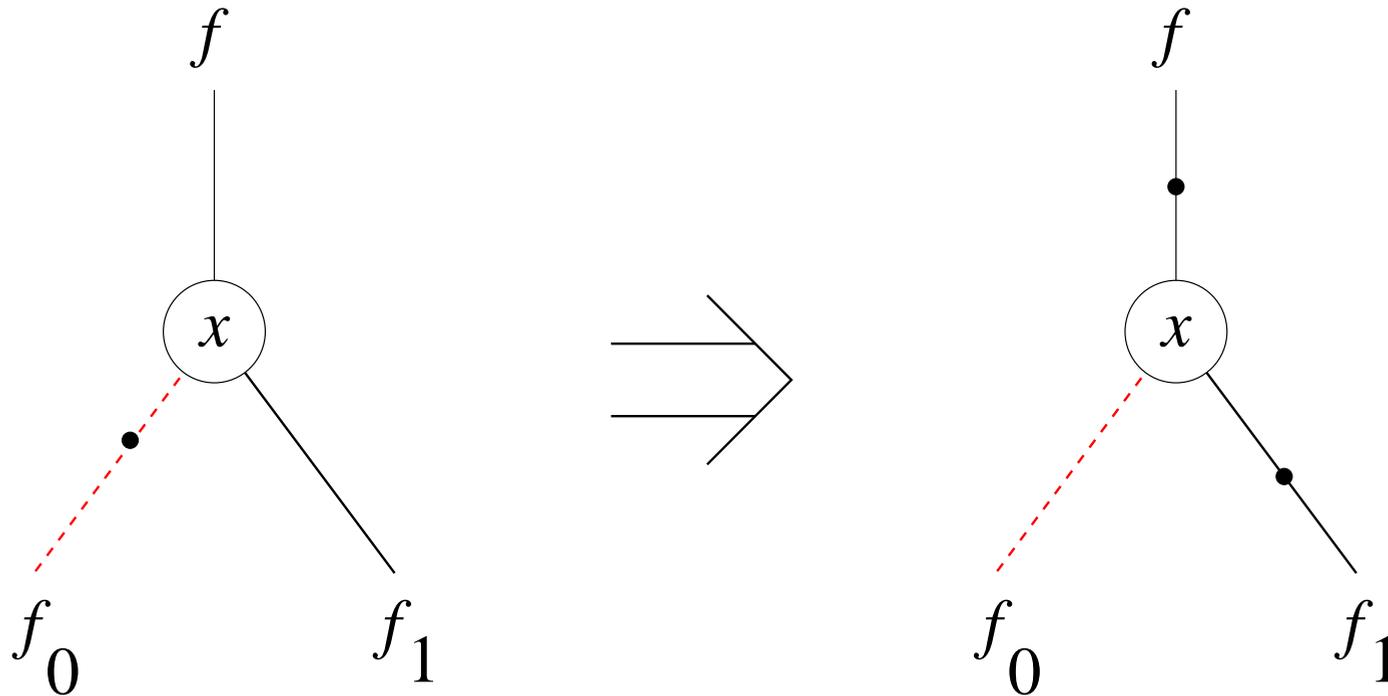
```
#define false_bdd ((BDD*) 0)
```

```
#define is_constant_bdd(ptr) \  
    ((ptr) == true_bdd || (ptr) == false_bdd)
```

```
#define UNIQUE_SIZE (1 << 20)
BDD *unique[UNIQUE_SIZE];

BDD **
find_bdd (int idx, BDD * c0, BDD * c1)
{
    BDD *r, **p;
    unsigned h = (idx + ((int) c0) * 65537 + 13 * (int) c1);
    h = h & (UNIQUE_SIZE - 1);
    for (p = unique + h; (r = *p); p = &r->next)
        if (r->idx == idx && r->child[0] == c0 && r->child[1] == c1)
            break;

    return p;
}
```



$$\begin{aligned}
 \text{ite}(x, f_1, \overline{f_0}) &\equiv x \cdot f_1 \vee \overline{x} \cdot \overline{f_0} \equiv \overline{(\overline{x} \vee \overline{f_1}) \cdot (x \vee f_0)} \\
 &\equiv \overline{x \cdot \overline{f_1} \vee \overline{x} \cdot f_0 \vee \overline{f_1} \cdot f_0} \\
 &\equiv \overline{x \cdot \overline{f_1} \vee \overline{x} \cdot f_0} \equiv \text{ite}(x, \overline{f_1}, f_0)
 \end{aligned}$$

```
BDD *
new_bdd_aux (int idx, BDD * c0, BDD * c1)
{
    BDD *res;

    assert (!sign_bdd (c0));

    if ((res = *find_bdd (idx, c0, c1)))
        return res;

    res = (BDD *) malloc (sizeof (BDD));
    memset (res, 0, sizeof (BDD));
    res->idx = idx;
    res->child[0] = c0;
    res->child[1] = c1;
    *find_bdd (idx, c0, c1) = res;

    return res;
}
```

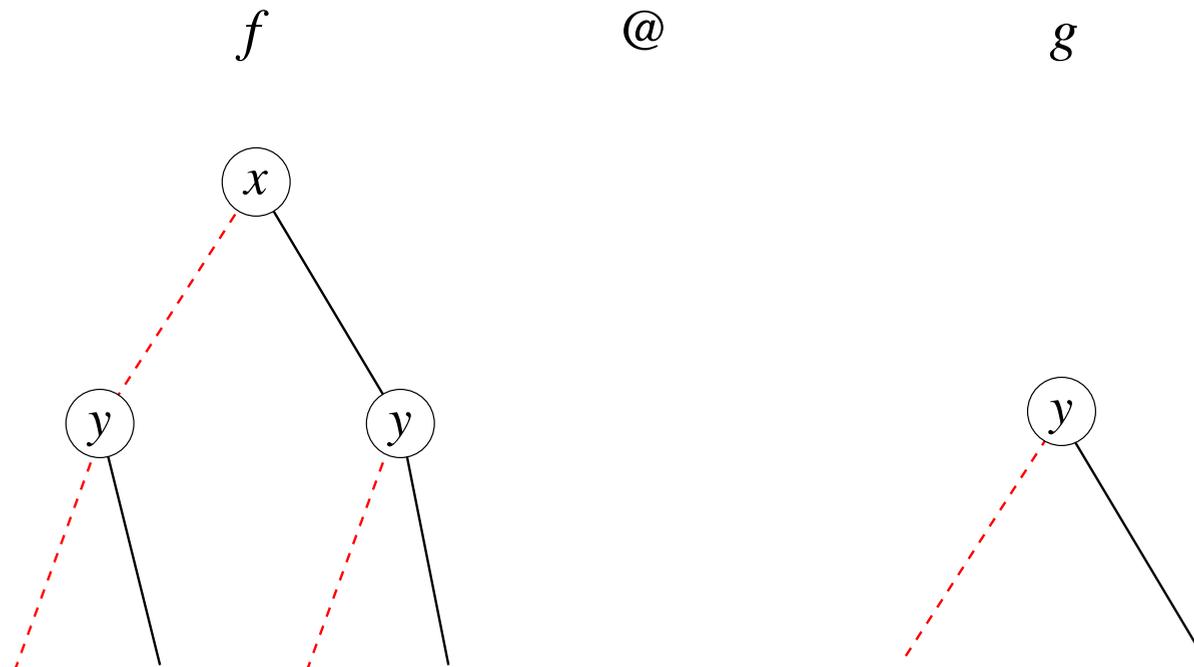
```
BDD *
new_bdd (int idx, BDD * c0, BDD * c1)
{
    BDD *res;
    int sign;

    if (c0 == c1)
        return c0;

    if ((sign = sign_bdd (c0))
        {
            c0 = not_bdd (c0);
            c1 = not_bdd (c1);
        }

    res = new_bdd_aux (idx, c0, c1);

    return sign ? not_bdd (res) : res;
}
```



$$(f \wedge g)(x) \equiv x \cdot (f_1 \wedge g) \vee \bar{x} \cdot (f_0 \wedge g)$$

bei nicht passenden Indizes wird nur ein Knoten zerlegt

```
int
top_idx_bdd (BDD * a, BDD * b)
{
    int res[2];
    res[0] = (is_constant_bdd (a)) ? -1 : strip_bdd (a)->idx;
    res[1] = (is_constant_bdd (b)) ? -1 : strip_bdd (b)->idx;
    return res[res[0] < res[1]];
}
```

gibt einfach den Index der Variablen im obersten Knoten zurück

(hier kann einer der beiden Argumente eine Konstante sein)

```
BDD *  
cofactor (BDD * bdd, int pos, int idx)  
{  
    BDD *res;  
    int sign;  
    if (is_constant_bdd (bdd))  
        return bdd;  
    if ((sign = sign_bdd (bdd))  
        bdd = not_bdd (bdd);  
    res = (bdd->idx == idx) ? bdd->child[pos] : bdd;  
    return sign ? not_bdd (res) : res;  
}
```

$pos$ -ter Kofaktor von  $bdd$  ist  $pos$ -tes Kind wenn Variablen-Index passt

ansonsten wird der gleiche BDD zurückgegeben

(man beachte Propagierung des Vorzeichens)

```
void
```

```
cofactor2 (BDD * a, BDD * b, BDD * c[2][2], int *idx_ptr)
{
    int idx = *idx_ptr = top_idx_bdd (a, b);
    c[0][0] = cofactor (a, 0, idx);
    c[0][1] = cofactor (a, 1, idx);
    c[1][0] = cofactor (b, 0, idx);
    c[1][1] = cofactor (b, 1, idx);
}
```

bestimme Top-Index der beiden BDDs a und b und

berechne Kofaktoren bezüglich des Top-Index

```
BDD *
basic_and (BDD * a, BDD * b)
{
    assert (is_constant_bdd (a) && is_constant_bdd (b));
    return (BDD *) (((unsigned) a) & (unsigned) b);
}
```

```
BDD *
basic_or (BDD * a, BDD * b)
{
    assert (is_constant_bdd (a) && is_constant_bdd (b));
    return (BDD *) (((unsigned) a) | (unsigned) b);
}
```

```
BDD *
basic_xor (BDD * a, BDD * b)
{
    assert (is_constant_bdd (a) && is_constant_bdd (b));
    return (BDD *) (((unsigned) a) ^ (unsigned) b);
}
```

```
typedef BDD *(*BasicFunctor) (BDD *, BDD *);

BDD *
apply (BasicFunctor op, BDD * a, BDD * b)
{
    BDD *tmp[2], *c[2][2];
    int idx;
    if (is_constant_bdd (a) && is_constant_bdd (b))
        return op (a, b);
    cofactor2 (a, b, c, &idx);
    tmp[0] = apply (op, c[0][0], c[1][0]);
    tmp[1] = apply (op, c[0][1], c[1][1]);
    return new_bdd (idx, tmp[0], tmp[1]);
}
```

Definition eines Funktions-Pointer-Typ, zur Aufnahme von Operationen

```
BDD *  
and_bdd (BDD * a, BDD * b)  
{  
    return apply (basic_and, a, b);  
}
```

```
BDD *  
or_bdd (BDD * a, BDD * b)  
{  
    return apply (basic_or, a, b);  
}
```

```
BDD *  
xor_bdd (BDD * a, BDD * b)  
{  
    return apply (basic_xor, a, b);  
}
```

```
BDD *  
var_bdd (int idx)  
{  
    return new_bdd (idx, false_bdd, true_bdd);  
}
```

- es fehlt noch ein **Cache**: dann linear in beiden Argumenten  
im Cache merkt man sich schon mal durchgeführte Berechnungen  
dann können alle Knoten von beiden Argumenten nur einmal auftreten
- **Optimierung**: Spezialalgorithmen für AND und XOR einführen  
auch wenn nur ein Argument konstant ist, kann man früher abbrechen  
ebenso, wenn beide Argumente bis auf das Vorzeichen identisch sind
- weitere BDD-Algorithmen basieren auf gleichem Schema:  
Kofaktoren bestimmen, Rekursion, mit Shannon zusammensetzen

- Kanonische Datenstruktur für boolesche Funktionen
- kein Allheilmittel aber wesentlich besser als DNF
- häufig kompakt für in der Praxis auftretenden Funktionen
- Trade-Off: *Platz statt Zeit*
- Anwendungen:  
Synthese, Symbolische Simulation, Equivalence Checking, Model-Checking