Binary Decision Diagrams

(Based on [Clarke et al. 1999] and [Bryant 1986])

Yih-Kuen Tsay
(original created by Ming-Hsien Tsai)

Dept. of Information Management
National Taiwan University

Boolean Functions

- Boolean functions are widely used in
  - digital logic design and testing,
  - artificial intelligence,
  - combinatorics, and
  - model checking.
- Boolean operators
  - Conjunction (and): $x \cdot y \quad (x \wedge y)$
  - Disjunction (or): $x + y \quad (x \vee y)$
  - Negation (not): $\bar{x} \quad (\neg x)$
  - Equivalence (if and only if): $\leftrightarrow$
- Example: $f(x_1, x_2, x_3, x_4) = (x_1 \leftrightarrow x_2) \cdot (x_3 \leftrightarrow x_4)$

Representations of Boolean Functions

- A variety of methods had earlier been developed for representing and manipulating Boolean functions:
  - Truth table
  - Karnaugh map
  - Sum-of-products form
  - Binary decision tree
- These representations are quite impractical, because every function of n arguments has a representation of size 2^n or more.

Truth Table

A truth table for $f(x_1, x_2, x_3, x_4) = (x_1 \leftrightarrow x_2) \cdot (x_3 \leftrightarrow x_4)$.

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

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

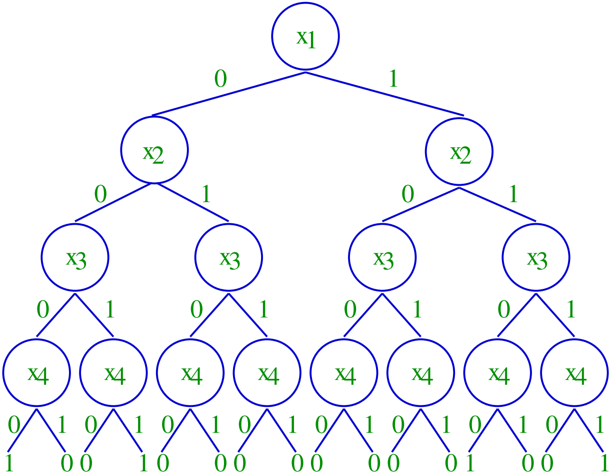
Karnaugh Map

A Karnaugh table for $f(x_1, x_2, x_3, x_4) = (x_1 \leftrightarrow x_2) \cdot (x_3 \leftrightarrow x_4)$.

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

Binary Decision Tree

A binary decision tree for $f(x_1, x_2, x_3, x_4) = (x_1 \leftrightarrow x_2) \cdot (x_3 \leftrightarrow x_4)$.



Representations of Boolean Functions (cont.)

- More practical approaches utilize representations that, at least for many functions, are not of exponential size.
  - reduced sum of products
  - factored into unate (cf. monotone) functions
- These representations still suffer from several drawbacks:
  - Certain common functions require representations of exponential size.
  - Performing a simple operation could yield a function with an exponential representation.
  - None of these representations are *canonical forms* (which are convenient for equivalence testing).

Binary Decision Diagrams

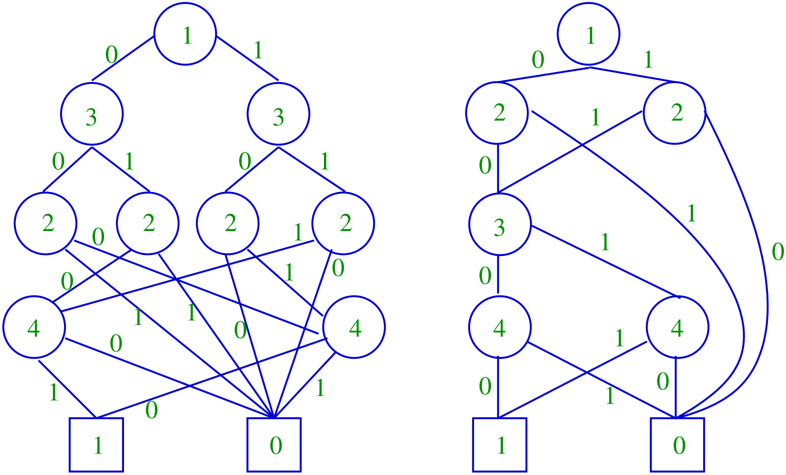
- A binary decision diagram (BDD) represents a Boolean function as a rooted, directed acyclic graph (function graph).
- We use $r(G)$ to denote the root of a function graph G.
- The vertex set V of a function graph G contains two types of vertices.
  - A nonterminal vertex v has
    - an argument index $index(v) \in \{1, \ldots, n\}$ and
    - two children $low(v), high(v) \in V$.
  - A terminal vertex v has a value $value(v) \in \{0, 1\}$.

Ordered Binary Decision Diagrams

- An ordered binary decision diagram (OBDD) is defined by imposing a total ordering over the nonterminal vertices.
  - For any nonterminal vertex v,
    - if $low(v)$ is nonterminal, then we must have $index(v) < index(low(v))$;
    - if $high(v)$ is nonterminal, then we must have $index(v) < index(high(v))$.
- Further minimality conditions will be introduced later.
- OBDDs are representations of Boolean functions with *canonical forms* and *reasonable size*.
- The size of the graph is highly sensitive to arguments ordering.

Ordering

Two OBDDs for $f(x_1, x_2, x_3, x_4) = (x_1 \leftrightarrow x_2) \cdot (x_3 \leftrightarrow x_4)$ with different orderings.



Notations

- All functions have the same n arguments: $x_1, \cdots, x_n$.
- A restriction of f is denoted $f|_{x_i=b}$ where b is a constant.

$$f|_{x_i=b}(x_1, \ldots, x_n) = f(x_1, \ldots, x_{i-1}, b, x_{i+1}, \ldots, x_n)$$

- A composition of f and g is denoted $f|_{x_i=g}$ where g is a Boolean function.

$$f|_{x_i=g}(x_1, \ldots, x_n) = f(x_1, \ldots, x_{i-1}, g(x_1, \ldots, x_n), x_{i+1}, \ldots, x_n)$$

Notations (cont.)

- The Shannon expansion of a function around variable x_i is given by:

$$f = x_i \cdot f\,|_{x_i=1} + \bar{x}_i \cdot f\,|_{x_i=0}$$

- The dependency set of a function f is denoted I_f.

$$I_f = \{i \mid f\,|_{x_i=0} \neq f\,|_{x_i=1}\}$$

- The satisfying set of a function f is denoted S_f.

$$S_f = \{(x_1, \ldots, x_n) \mid f(x_1, \ldots, x_n) = 1\}$$

Correspondence

- A function graph (OBDD) G having root vertex v denotes a function f_v defined recursively as follows:
  - If v is a terminal vertex:
    - If $value(v) = 1$, then $f_v = 1$.
    - If $value(v) = 0$, then $f_v = 0$.
  - If v is a nonterminal vertex with $index(v) = i$, then f_v is the function

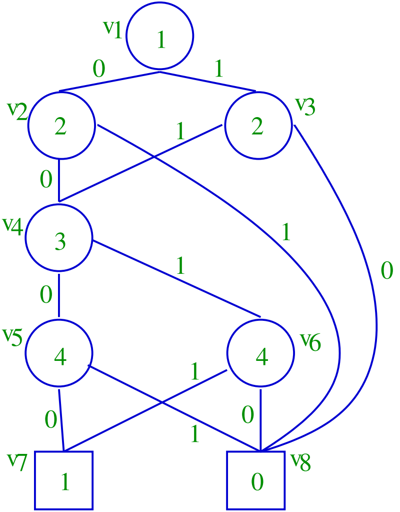
$$f_v(x_1, \ldots, x_n) = \bar{x}_i \cdot f_{low(v)}(x_1, \ldots, x_n) + x_i \cdot f_{high(v)}(x_1, \ldots, x_n).$$

Correspondence (cont.)

- A path in the graph starting from the root is defined by a set of argument values.
- The value of the function for these arguments equals the value of the terminal vertex at the end of the path.
- Every vertex in the graph is contained in at least one path.

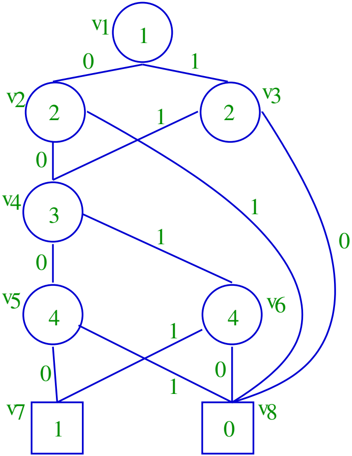
Correspondence (cont.)

$$
\begin{aligned}
f_{v_8} &= 0 \\
f_{v_7} &= 1 \\
f_{v_6} &= \bar{x}_4 \cdot f_{v_8} + x_4 \cdot f_{v_7} \\
&= x_4 \\
f_{v_5} &= \bar{x}_4 \cdot f_{v_7} + x_4 \cdot f_{v_8} \\
&= \bar{x}_4 \\
f_{v_4} &= \bar{x}_3 \cdot f_{v_5} + x_3 \cdot f_{v_6} \\
&= \bar{x}_3 \cdot \bar{x}_4 + x_3 \cdot x_4 \\
&\cdots \\
&\cdots \\
&\cdots \\
f_{v_1} &= (\bar{x}_1 \cdot \bar{x}_2 + x_1 \cdot x_2) \cdot (\bar{x}_3 \cdot \bar{x}_4 + x_3 \cdot x_4)
\end{aligned}
$$



Subgraph

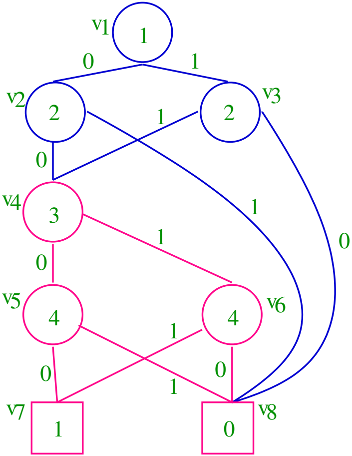
- For any vertex v in a function graph G, the subgraph rooted at v, denoted by $sub(G, v)$ is defined as the graph consisting of v and all its descendants.



Subgraph

IM NTU

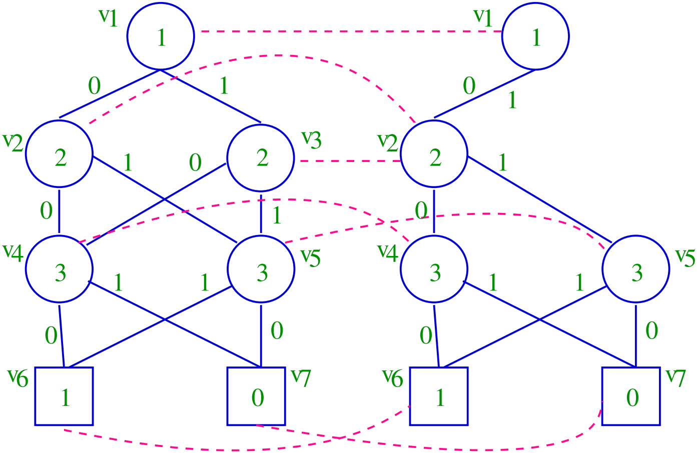
- For any vertex v in a function graph G, the subgraph rooted at v, denoted by $sub(G, v)$ is defined as the graph consisting of v and all its descendants.



Isomorphism

- Function graphs G and G' are isomorphic, denoted by $G \sim G'$, if there exists a one-to-one function σ from vertices of G onto the vertices of G' such that for any vertex v if $\sigma(v) = v'$, then either
  - both v and v' are terminal vertices with $value(v) = value(v')$, or
  - both v and v' are nonterminal vertices with $index(v) = index(v')$, $\sigma(low(v)) = low(v')$, and $\sigma(high(v)) = high(v')$

Isomorphism (cont.)



Is this an isomorphic mapping? (part of it is)

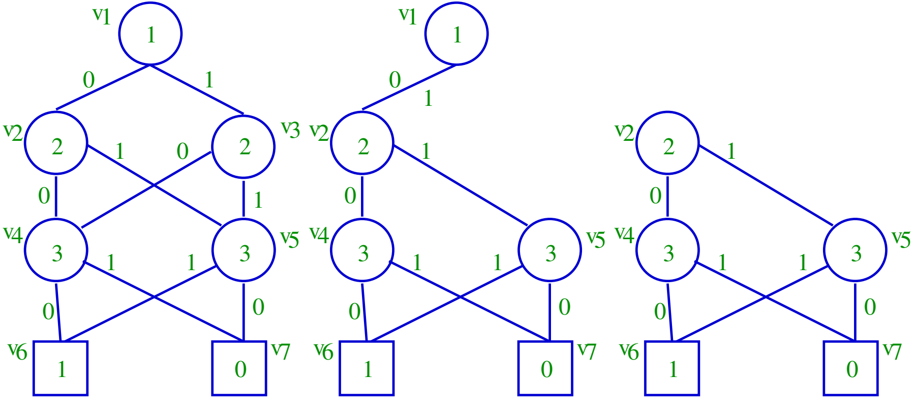
Isomorphism (cont.)

- The isomorphic mapping σ is quite constrained:
  - $r(G)$ must map to the $r(G')$,
  - $low(r(G))$ must map to $low(r(G'))$,
  - and so on all the way down to the terminal vertices.
- Lemma 1: If G is isomorphic to G' by mapping σ, denoted by $G \sim_\sigma G'$, then for any vertex v in G, $sub(G, v) \sim sub(G', \sigma(v))$.

Reduced Function Graph

- A function graph G is reduced if
  - it contains no vertex v with $low(v) = high(v)$,
  - nor does it contain distinct vertices v and v' such that the subgraphs rooted by v and v' are isomorphic.
- A reduced function graph is now commonly called (Reduced) OBDD.
- Lemma 2: For every vertex v in a reduced function graph G, $sub(G, v)$ is itself a reduced function graph.

Reduced Function Graph (cont.)



Canonical Form

- Theorem: For any Boolean function f, there is a unique (up to isomorphism) reduced function graph denoting f and any other function graph denoting f contains more vertices.

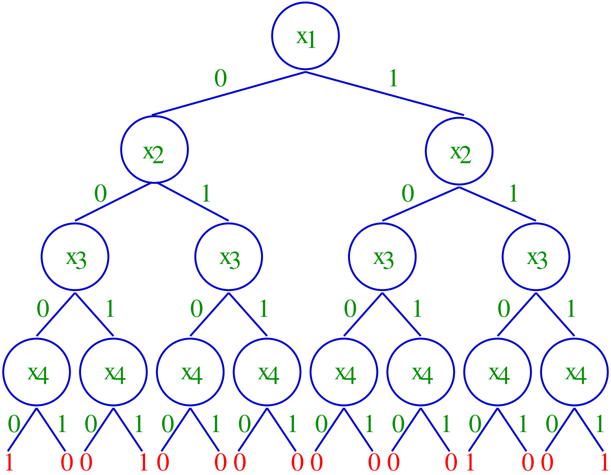
Basic Operations

Procedure	Result	Time Complexity
Reduce	G reduced to canonical form	$O(\|G\| \cdot \log \|G\|)$
Restrict	$f\|_{x_i=b}$	$O(\|G\| \cdot \log \|G\|)$
Apply	$f_1 \langle op \rangle f_2$	$O(\|G_1\| \cdot \|G_2\|)$
Compose	$f_1\|_{x_i=f_2}$	$O(\|G_1\|^2 \cdot \|G_2\|)$
Satisfy-one	some element of S_f	$O(n)$
Satisfy-all	S_f	$O(n \cdot \|S_f\|)$
Satisfy-count	$\|S_f\|$	$O(\|G\|)$

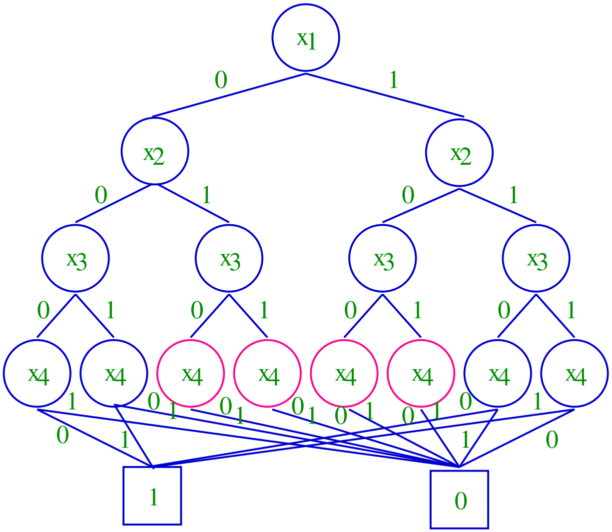
Reduction

- The *reduction* algorithm transforms an arbitrary function graph into a reduced graph denoting the same function.
- The algorithm works from the terminal vertices up to the root:
  - Remove duplicate terminals (terminal vertices v and u such that $value(v) = value(u)$).
  - Remove duplicate nonterminals (nonterminal vertices v and u such that $index(v) = index(u)$, $id(low(v)) = id(low(u))$, and $id(high(v)) = id(high(u))$).
  - Remove duplicate tests (a nonterminal vertex v such that $low(v) = high(v)$).

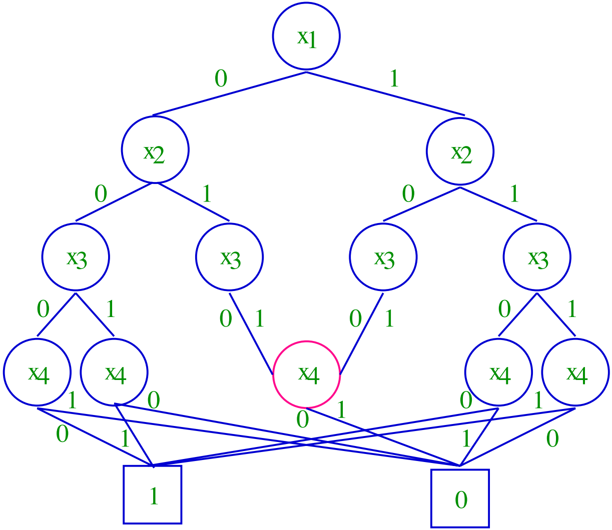
A Reduction Example



A Reduction Example

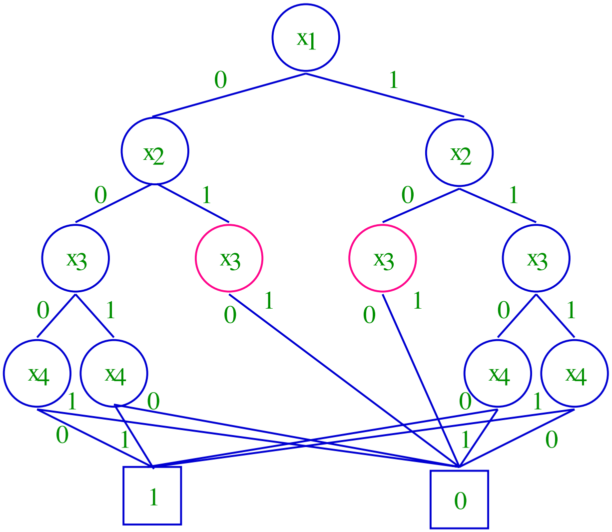


A Reduction Example



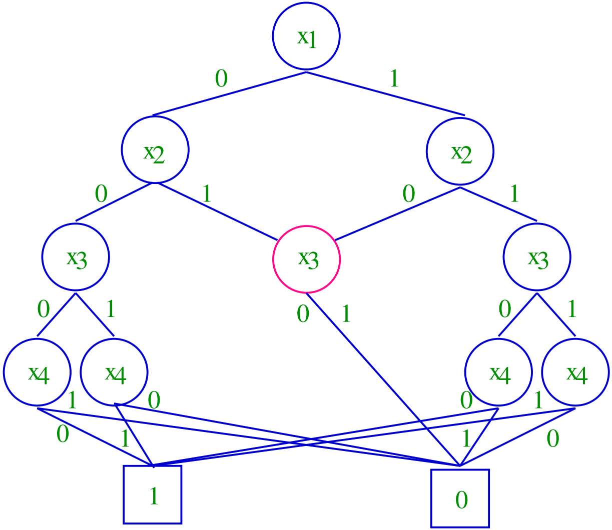
Note: not strictly bottom to top (for better layouts).

A Reduction Example



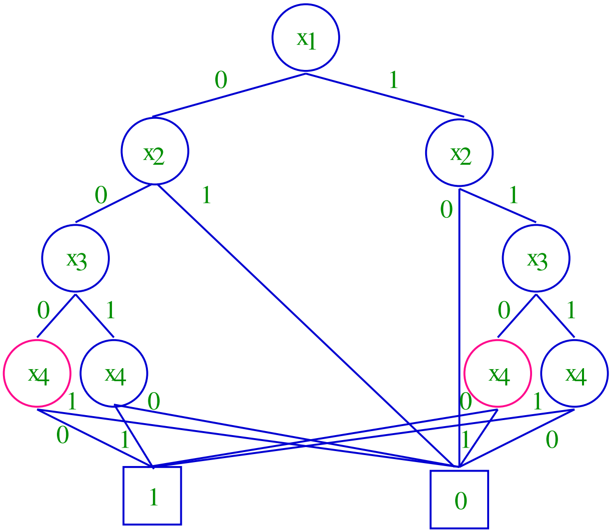
Note: not strictly bottom to top (for better layouts).

A Reduction Example

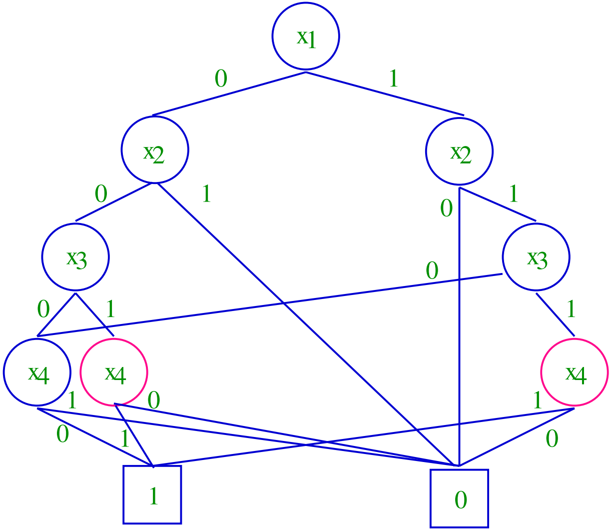


Note: not strictly bottom to top (for better layouts).

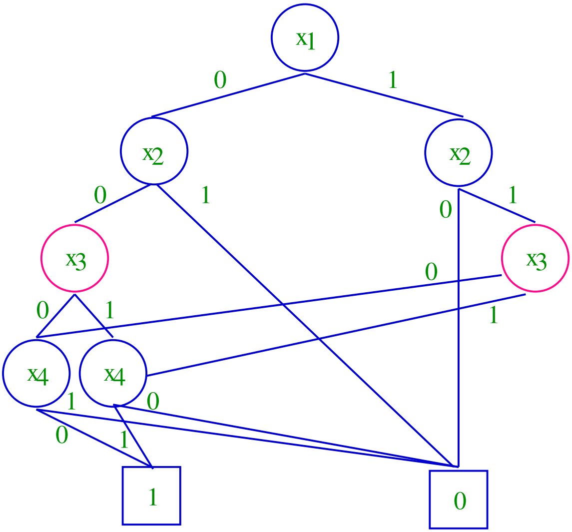
A Reduction Example



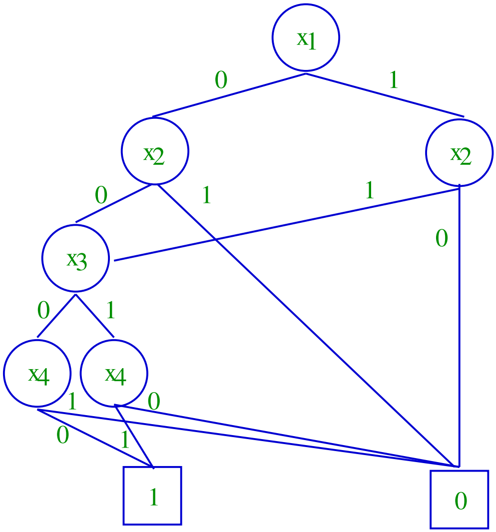
A Reduction Example



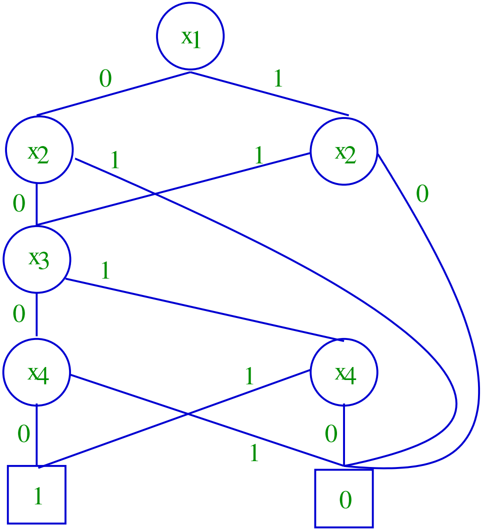
A Reduction Example



A Reduction Example



A Reduction Example

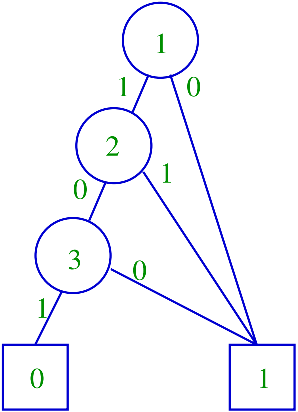


Restriction

- The procedure *Restrict* transforms the graph representing a function f into one representing the function $f|_{x_i=b}$.
- Steps of *Restrict*:
  - Look for a vertex v with $index(v) = i$.
  - Change it to point either to $low(v)$ (for $b = 0$) or to $high(v)$ (for $b = 1$).
  - After changing every vertex v with $index(v) = i$, run the reduction procedure.

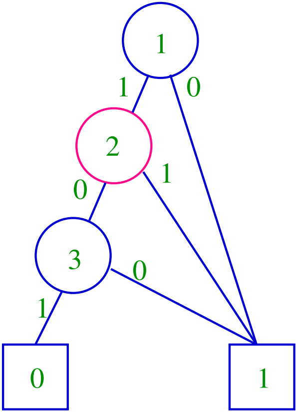
A Restriction Example

$$\overline{x_1 \cdot \overline{x_2} \cdot x_3}\Big|_{x_2 = 0} \quad = \quad \overline{x_1 \cdot x_3}$$



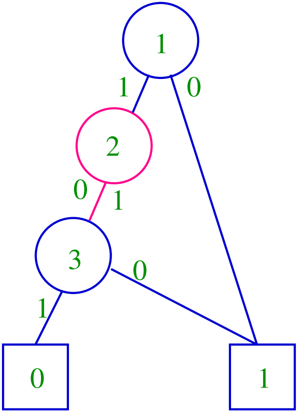
A Restriction Example

$$\overline{x_1 \cdot \overline{x_2} \cdot x_3}\Big|_{x_2 = 0} = \overline{x_1 \cdot x_3}$$



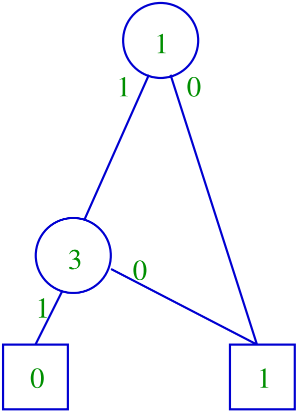
A Restriction Example

$$\overline{x_1 \cdot \overline{x_2} \cdot x_3}\Big|_{x_2 = 0} = \overline{x_1 \cdot x_3}$$



A Restriction Example

$$\overline{x_1 \cdot \overline{x_2} \cdot x_3}\Big|_{x_2 = 0} = \overline{x_1 \cdot x_3}$$



Apply

- The procedure *Apply* takes graphs representing functions f_1 and f_2, a binary operator $\langle op \rangle$, and produces a reduced graph representing the function $f_1 \langle op \rangle f_2$ defined as:

$$[f_1 \langle op \rangle f_2](x_1, \ldots, x_n) = f_1(x_1, \ldots, x_n) \langle op \rangle f_2(x_1, \ldots, x_n).$$

- It is based on the following recursion derived from the Shannon expansion:

$$f_1 \langle op \rangle f_2 = \bar{x}_i \cdot (f_1 \,|_{x_i=0} \, \langle op \rangle f_2 \,|_{x_i=0}) + x_i \cdot (f_1 \,|_{x_i=1} \, \langle op \rangle f_2 \,|_{x_i=1})$$

Apply (cont.)

- Given function f_1 rooted at v_1 and function f_2 rooted at v_2, there are four cases to consider:
  - v_1 and v_2 are terminals: $f_1 \langle op \rangle f_2 = value(v_1) \langle op \rangle value(v_2)$
  - $index(v_1) = index(v_2)$: use the derived recursion
  - $index(v_1) (= i) < index(v_2)$: $f_2 |_{x_i = 0} = f_2 |_{x_i = 1} = f_2$, so

$$f_1 \langle op \rangle f_2 = \bar{x}_i \cdot (f_1 |_{x_i=0} \langle op \rangle f_2) + x_i \cdot (f_1 |_{x_i=1} \langle op \rangle f_2)$$

  - $index(v_1) > index(v_2)$: analogous to the above
- To avoid repeating the operation on two same nodes, we use dynamic programming.

Apply (cont.)

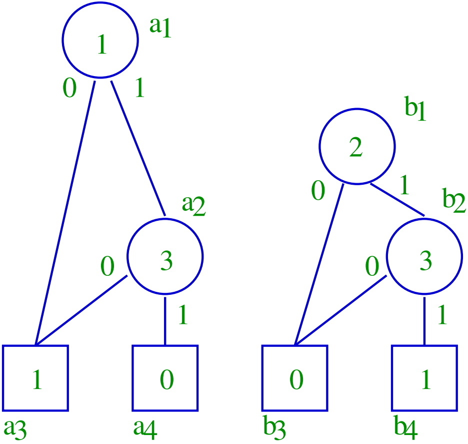
function *Apply*($v1$, $v2$: vertex $\langle op \rangle$: operator): vertex
{var T: array[1..$|G_1|$, 1..$|G_2|$] of vertex;}
begin
    Initialize all elements of T to null;
    $u :=$ *Apply-step*($v1$, $v2$);
    return(*Reduce*(u));
end;

Apply (cont.)

```
function Apply-step(v1, v2: vertex): vertex;
begin
    u:= T[v1.id, v2.id];
    if u ≠ null then return(u);   // have already evaluated
    u:= new vertex record; u.mark := false;
    T[v1.id, v2.id] := u;   // add vertex to table
    u.value := v1.value ⟨op⟩ v2.value;
    if u.value ≠ X   // X means "don't care"; for a nonterminal v, v.value = X
    then begin u.index := n + 1; u.low := null; u.high := null; end
    else begin   // create nonterminal and evaluate further down
        u.index := Min(v1.index, v2.index);
        if v1.index = u.index
            then begin vlow1 := v1.low; vhigh1 := v1.high end
            else begin vlow1 := v1; vhigh1 := v1 end;
        if v2.index = u.index
            then begin vlow2 := v2.low; vhigh2 := v2.high end
            else begin vlow2 := v2; vhigh2 := v2 end;
        u.low := Apply-step(vlow1, vlow2);
        u.high := Apply-step(vhigh1, vhigh2);
    end;
    return(u);
end;
```

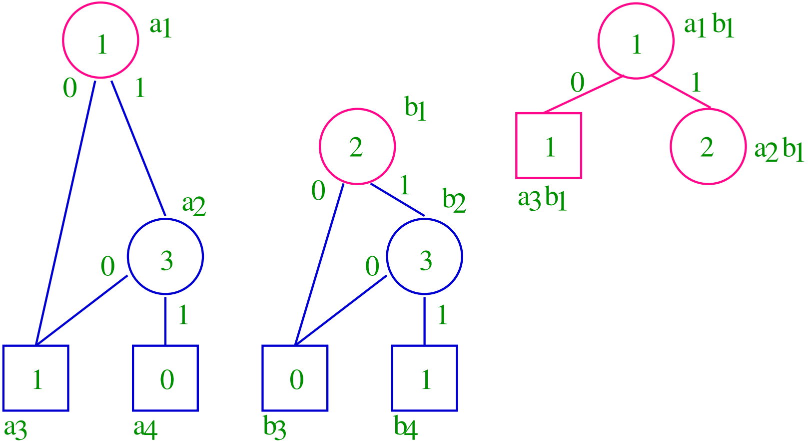
An Apply Example

$$\overline{x_1 \cdot x_3} \quad + \quad x_2 \cdot x_3 \quad = \quad \overline{x_1 \cdot \overline{x_2} \cdot x_3}$$

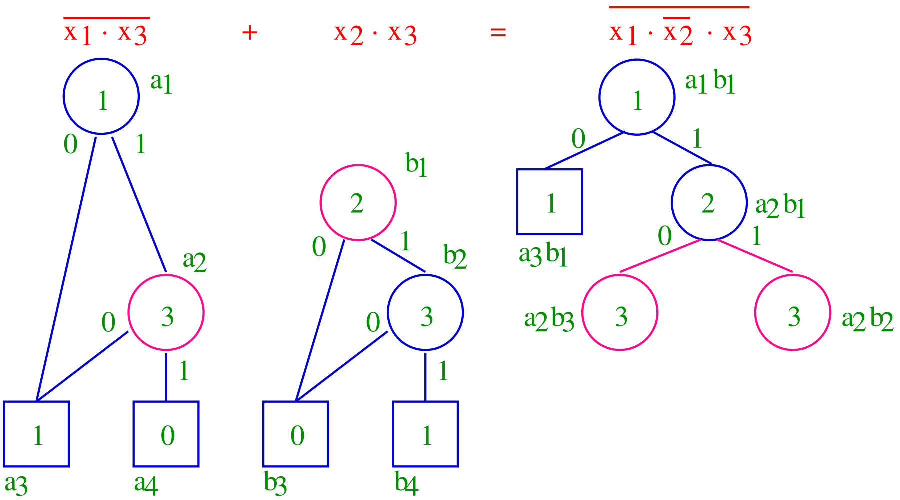


An Apply Example

$$\overline{x_1 \cdot x_3} \quad + \quad x_2 \cdot x_3 \quad = \quad \overline{x_1 \cdot \overline{x_2} \cdot x_3}$$

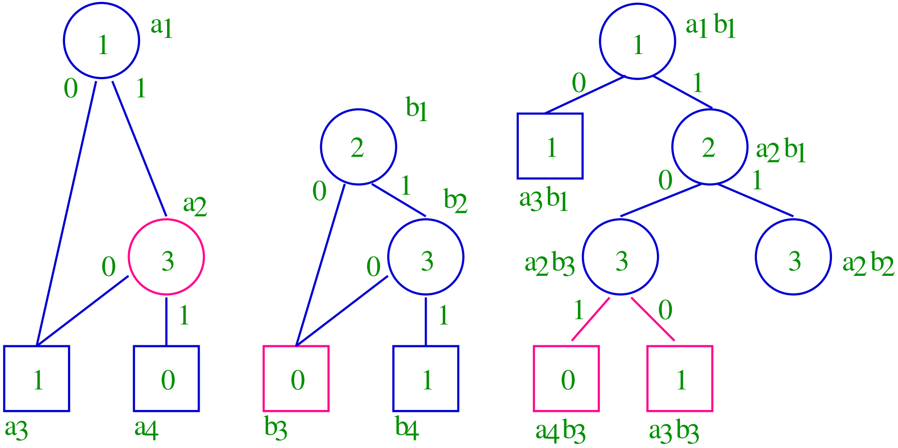


An Apply Example



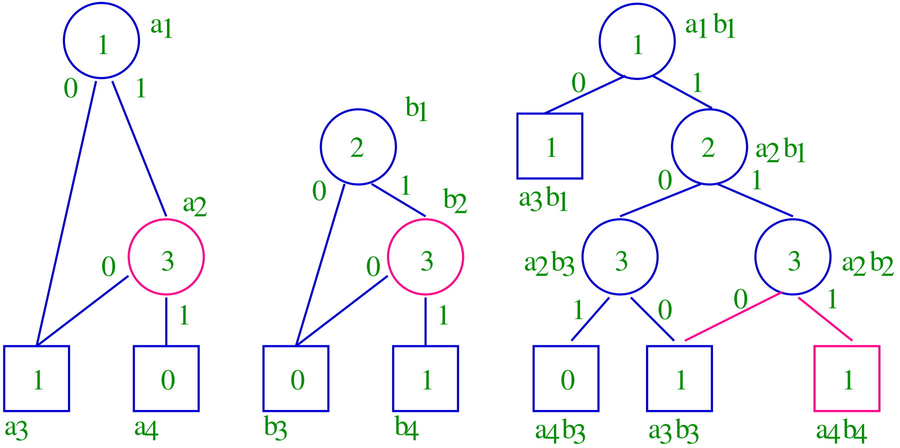
An Apply Example

$$\overline{x_1 \cdot x_3} \quad + \quad x_2 \cdot x_3 \quad = \quad \overline{x_1 \cdot \overline{x_2} \cdot x_3}$$



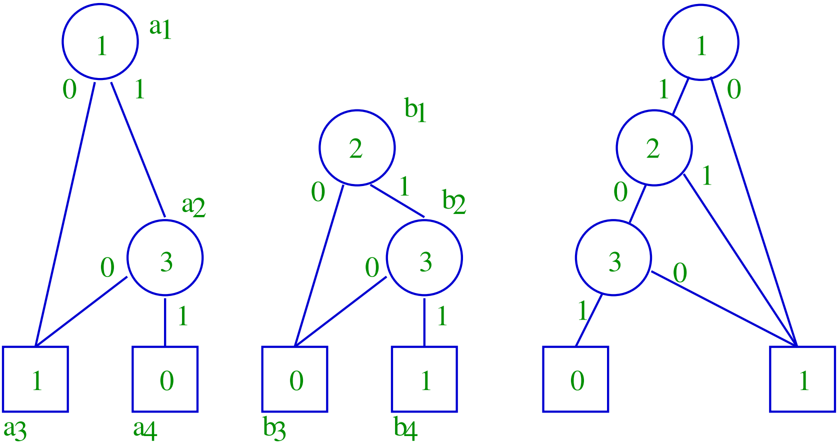
An Apply Example

$$\overline{x_1 \cdot x_3} \quad + \quad x_2 \cdot x_3 \quad = \quad \overline{x_1 \cdot \overline{x_2} \cdot x_3}$$



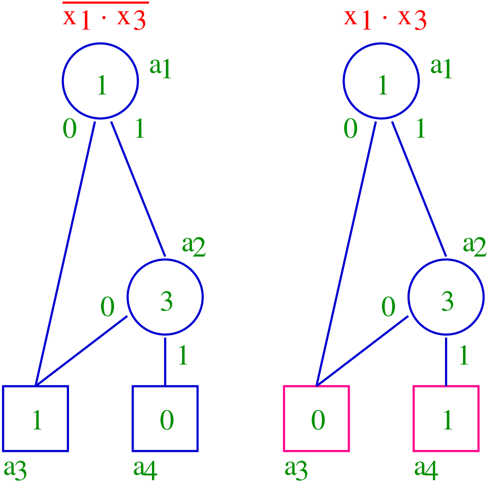
An Apply Example

$$\overline{x_1 \cdot x_3} \quad + \quad x_2 \cdot x_3 \quad = \quad \overline{x_1 \cdot \overline{x_2} \cdot x_3}$$



Complementation

- To complement an OBDD, simply complement its terminal vertices.



Composition

- The procedure *Compose* constructs the graph for the function obtained by composing two functions.
- Composition can be expressed in terms of restriction and Boolean operations according to the following expansion:

$$f_1|_{x_i = f_2} = f_2 \cdot f_1|_{x_i=1} + (\neg f_2) \cdot f_1|_{x_i=0}$$

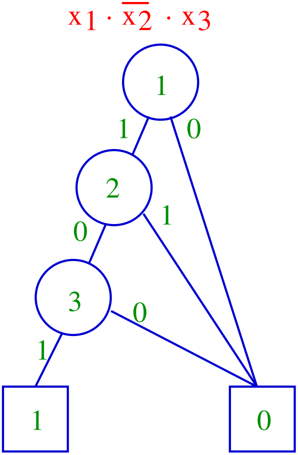
- It is sufficient to use *Restrict* and *Apply* to implement *Compose*.

Satisfy-one

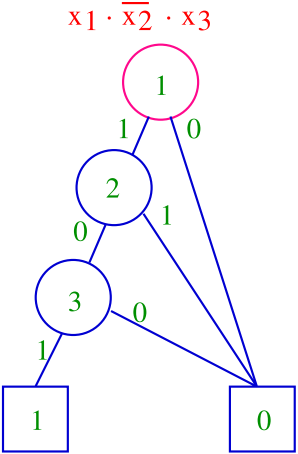
- The *Satisfy-one* procedure utilizes a classic depth-first search with backtracking.

```
function Satisfy-one(v: vertex; x: array[1..n] of int): boolean
begin
    if value(v) = 0 then return false;
    if value(v) = 1 then return true;
    x[i] := 0;
    if Satisfy-one(low(v), x) then return true;
    x[i] := 1;
    return Satisfy-one(high(v), x);
end;
```

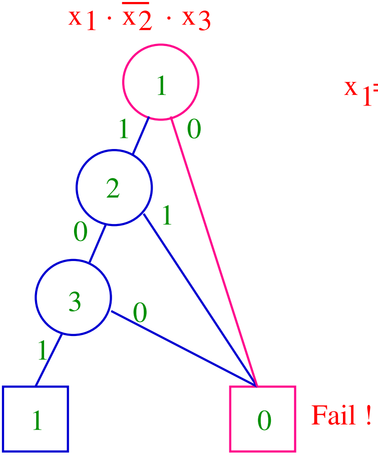
A Satisfy-one Example



A Satisfy-one Example

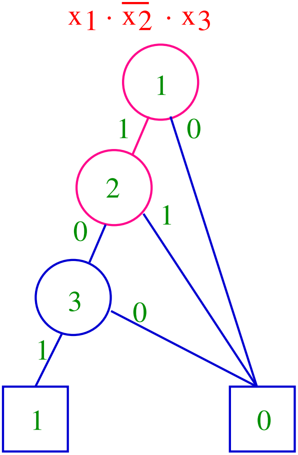


A Satisfy-one Example



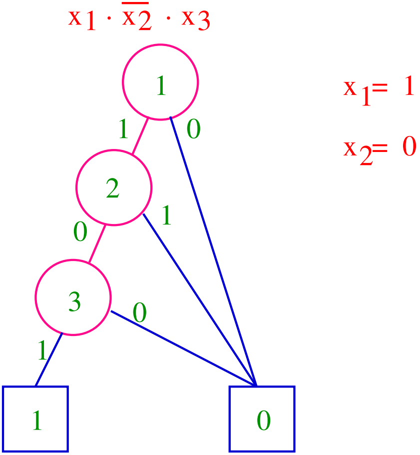
$x_1 = 0$

A Satisfy-one Example

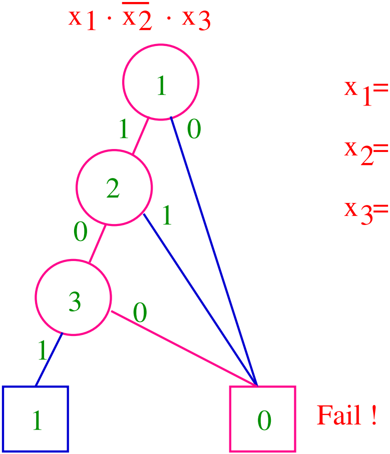


$x_1 = 1$

A Satisfy-one Example



A Satisfy-one Example

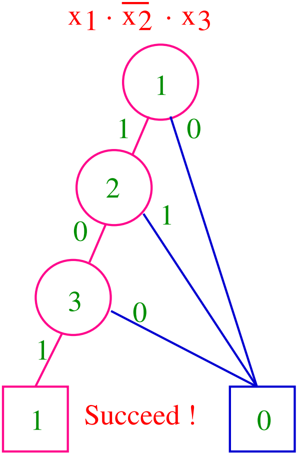


$x_1 = 1$

$x_2 = 0$

$x_3 = 0$

A Satisfy-one Example



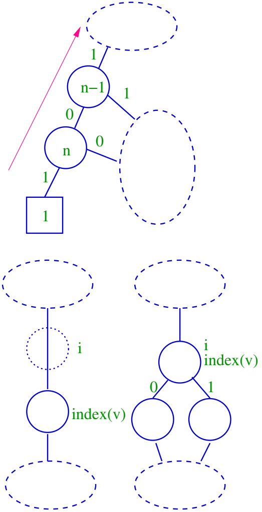
$x_1 = 1$

$x_2 = 0$

$x_3 = 1$

Satisfy-all

```
procedure Satisfy-all(i: int; v: vertex; x: array[1..n] of int):
begin
        if value(v) = 0 then return;
        if i = n + 1 and value(v) = 1
        then begin
                Print element x[1],...,x[n];
                return;
        end;
        if index(v) > i
        then begin
                x[i] := 0; Satisfy-all(i + 1, v, x);
                x[i] := 1; Satisfy-all(i + 1, v, x);
        end
        else begin
                x[i] := 0; Satisfy-all(i + 1, low(v), x);
                x[i] := 1; Satisfy-all(i + 1, high(v), x);
        end
end;
```



Satisfy-count

- The procedure *Satisfy-count* computes a value α_v to each vertex v in the graph according to the following recursive formula:
  - If v is a terminal vertex: $\alpha_v = value(v)$.
  - If v is a nonterminal vertex:

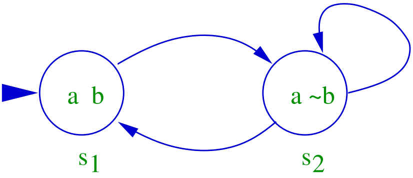
$$\alpha_v = \alpha_{low(v)} \cdot 2^{index(low(v)) - index(v)} + \alpha_{high(v)} \cdot 2^{index(high(v)) - index(v)}$$

- Once we have computed these values for a graph with root v, we compute the size of the satisfying set as

$$|S_f| = \alpha_v \cdot 2^{index(v)-1}$$

Kripke Structures

- Given a set of atomic propositions AP, a Kripke structure M is a four tuple (S, S_0, R, L):
  - S is a finite set of states.
  - $S_0 \subseteq S$ is the set of initial states.
  - $R \subseteq S \times S$ is a transition relation that must be total.
  - $L : S \rightarrow 2^{AP}$ is a function that labels each state with the set of atomic propositions true in that state.



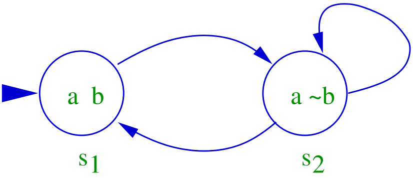
First Order Representations

- The initial states can be represented by the formula:

$$(a \wedge b)$$

- The transitions can be represented by the formula:

$$\begin{array}{ll} (a \wedge b \wedge a' \wedge \neg b') & \vee \\ (a \wedge \neg b \wedge a' \wedge \neg b') & \vee \\ (a \wedge \neg b \wedge a' \wedge b') & \end{array}$$



OBDD Representations

- Use x_1, x_2, x_3, x_4 to represent a, b, a', b' respectively.
- The characteristic function of initial states:

$$(a \wedge b)$$

becomes

$$(x_1 \cdot x_2)$$

OBDD Representations (cont.)

- The characteristic function of transitions:

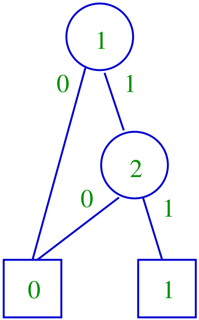
$$
\begin{array}{ll}
(a \wedge b \wedge a' \wedge \neg b') & \vee \\
(a \wedge \neg b \wedge a' \wedge \neg b') & \vee \\
(a \wedge \neg b \wedge a' \wedge b') &
\end{array}
$$

becomes

$$
\begin{array}{ll}
(x_1 \cdot x_2 \cdot x_3 \cdot \bar{x}_4) & + \\
(x_1 \cdot \bar{x}_2 \cdot x_3 \cdot \bar{x}_4) & + \\
(x_1 \cdot \bar{x}_2 \cdot x_3 \cdot x_4) &
\end{array}
$$

OBDD Representations (cont.)

Initial states: $x_1 \cdot x_2$



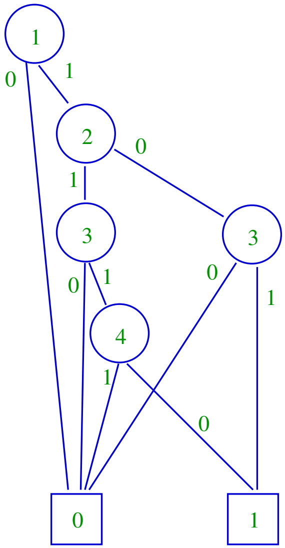
OBDD Representations (cont.)

Transitions:

$(x_1 \cdot x_2 \cdot x_3 \cdot \bar{x}_4) \quad +$

$(x_1 \cdot \bar{x}_2 \cdot x_3 \cdot \bar{x}_4) \quad +$

$(x_1 \cdot \bar{x}_2 \cdot x_3 \cdot x_4)$



Summary

- OBDDs are representations of Boolean functions with
  - canonical forms and
  - reasonable size.
- Transition systems can be encoded in Boolean functions and thus representable in OBDDs.
- Symbolic model checking becomes possible with OBDDs.