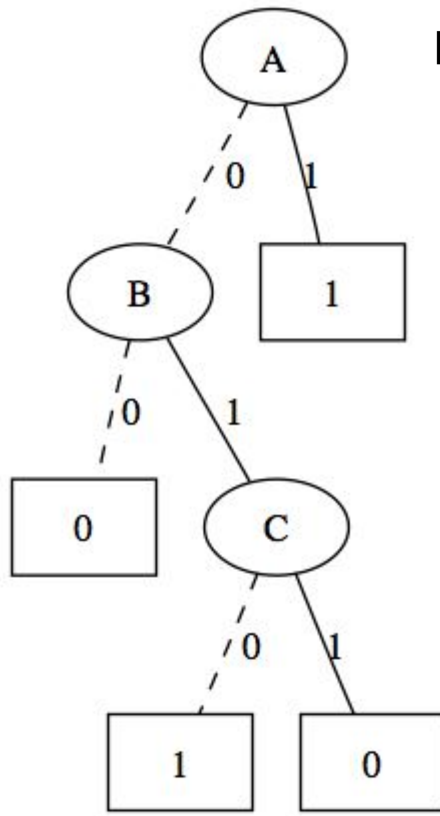
Binary Decision Diagrams

Example

Directed acyclic graph

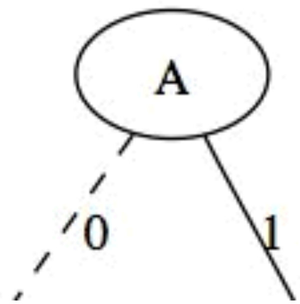
non-terminal node

What function
is represented
by the graph?



terminal node

Shannon Decomposition



$ABC + \bar{A}\bar{C}$

low edge

high edge

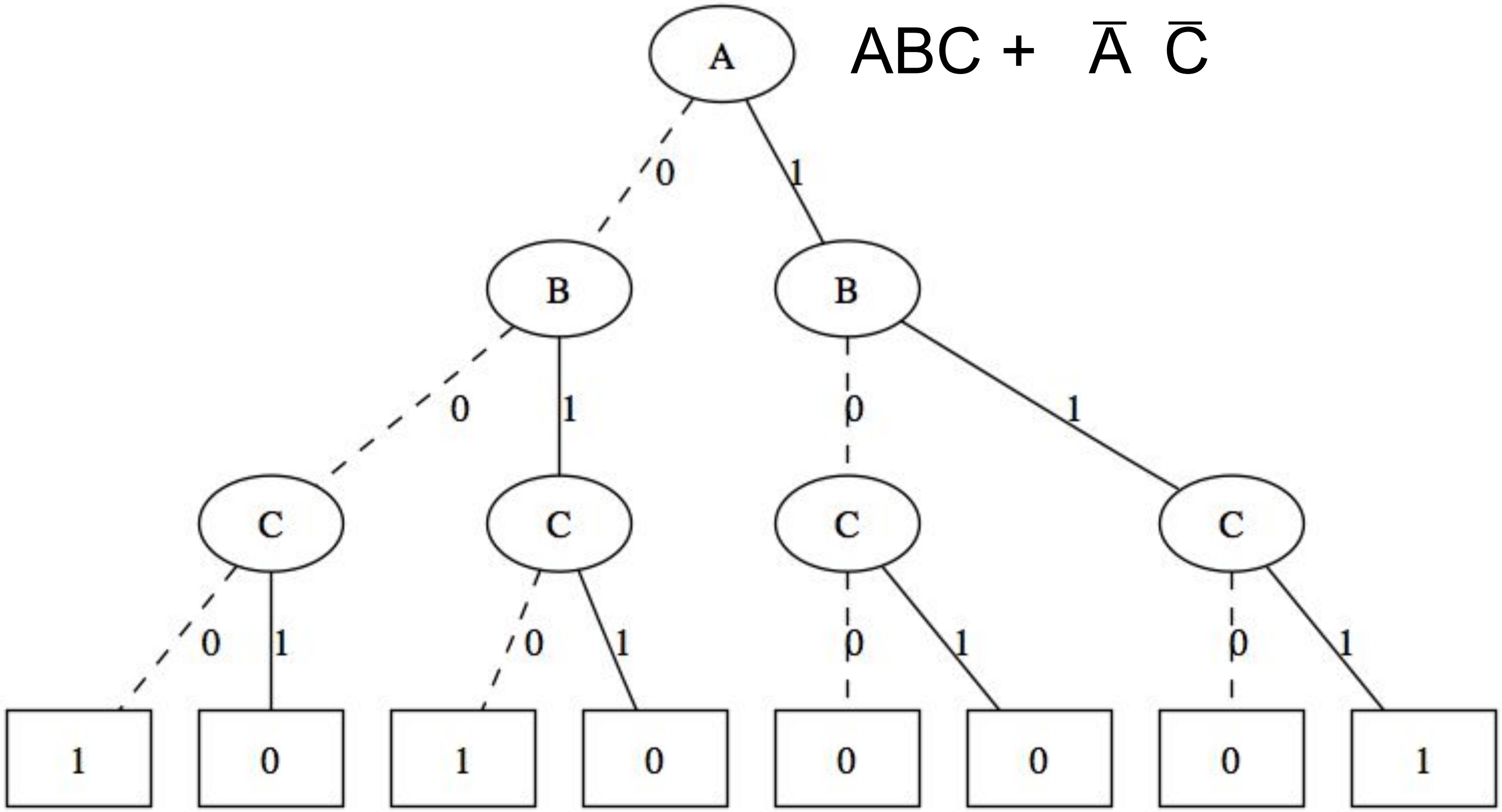
$\bar{C}$

BC

Apply the decomposition recursively

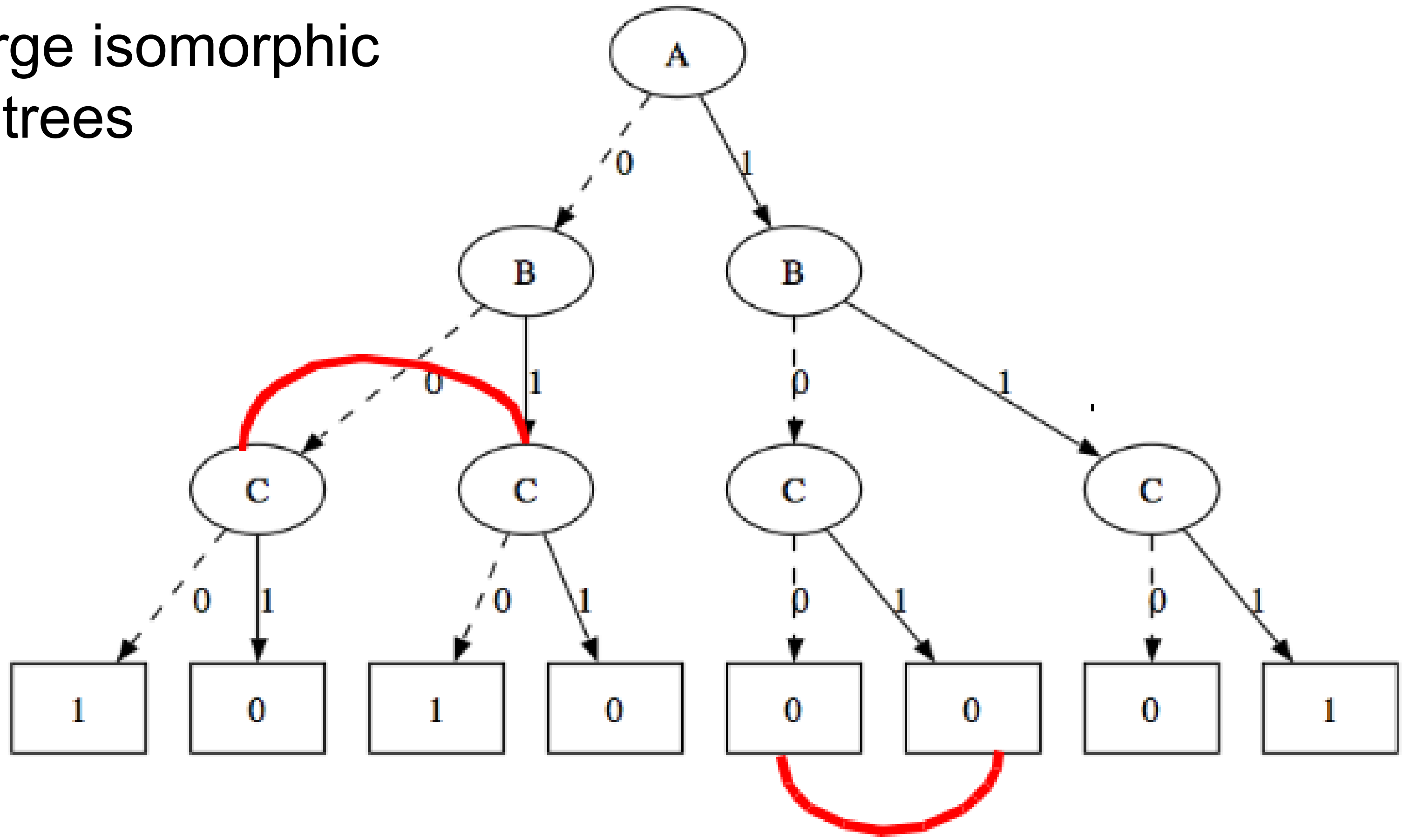
Ordered BDD

$ABC + \bar{A}\,\bar{C}$



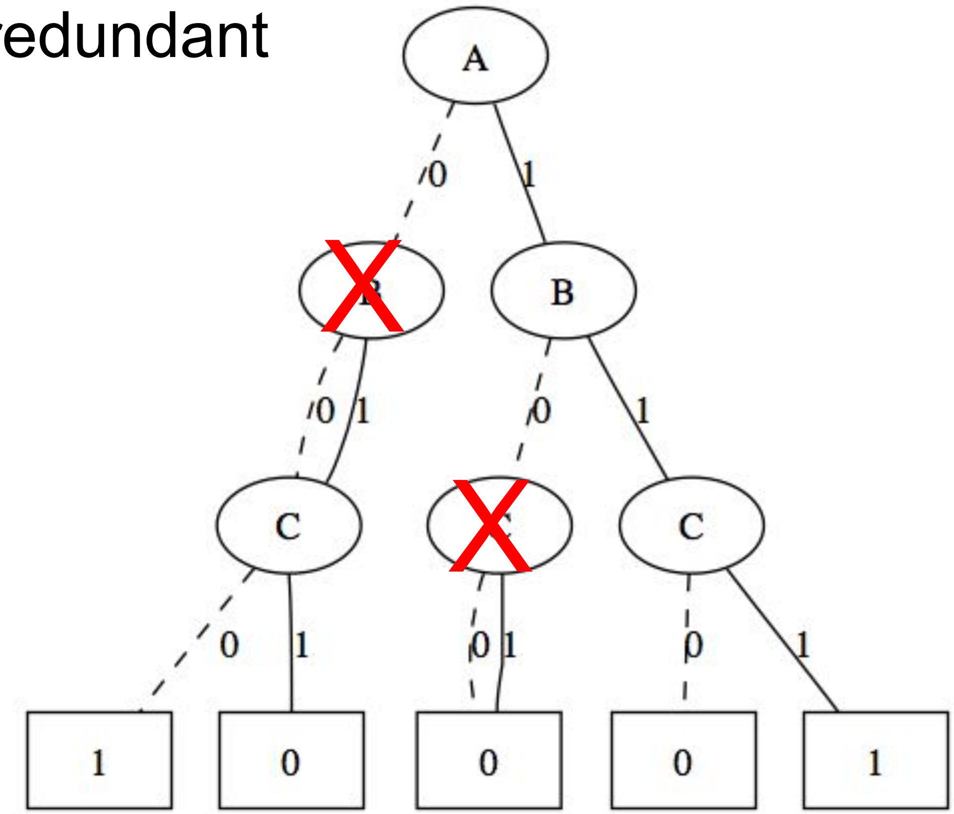
Reduced Ordered BDD

Merge isomorphic subtrees

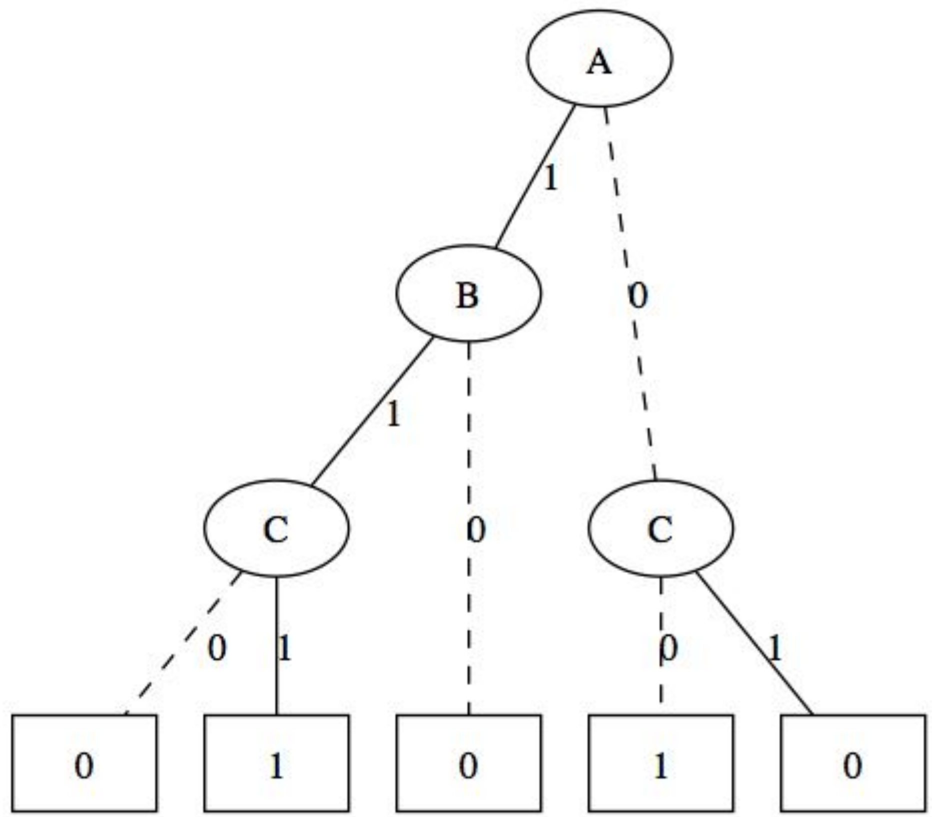


Reduced Ordered BDD

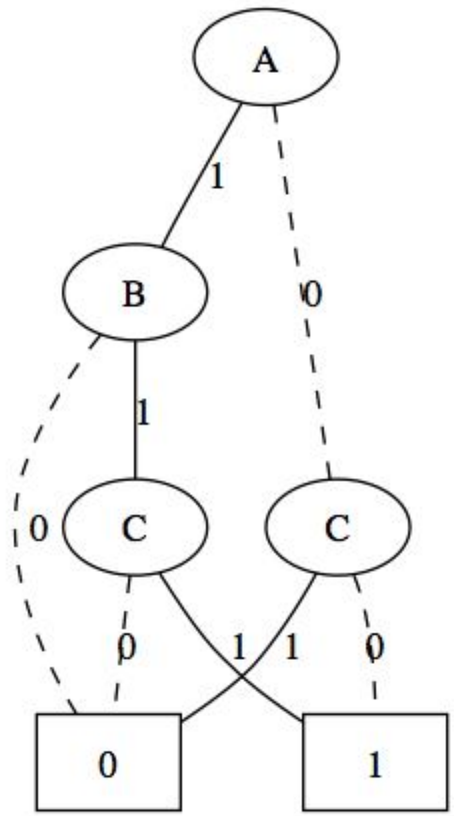
Remove redundant nodes



ROBDD



ROBDD



BDD

A Binary Decision Diagram (BDD) is a rooted, directed acyclic graph

- with one or two terminal nodes of out-degree zero labeled 0 or 1, and a set of variable nodes u of out-degree two.
- The two outgoing edges are given by two functions low(u) and high(u). (In pictures, these are shown as dotted and solid lines, respectively.) A variable var(u) is associated with each variable node.

ROBDD

- A BDD is Ordered (OBDD) if on all paths through the graph the variables respect a given linear order $x_1 < x_2 < \dots < x_n$. An OBDD is Reduced (ROBDD) if

1. (**uniqueness**) no two distinct nodes u and v have the same variable name and low- and high-successor, i.e.,

   var(u) = var(v); low(u) = low(v); high(u) = high(v) implies u = v;

and

2. (**non-redundant tests**) no variable node u has identical low- and high-successor, i.e.,

   low(u) = high(u)

Canonicity

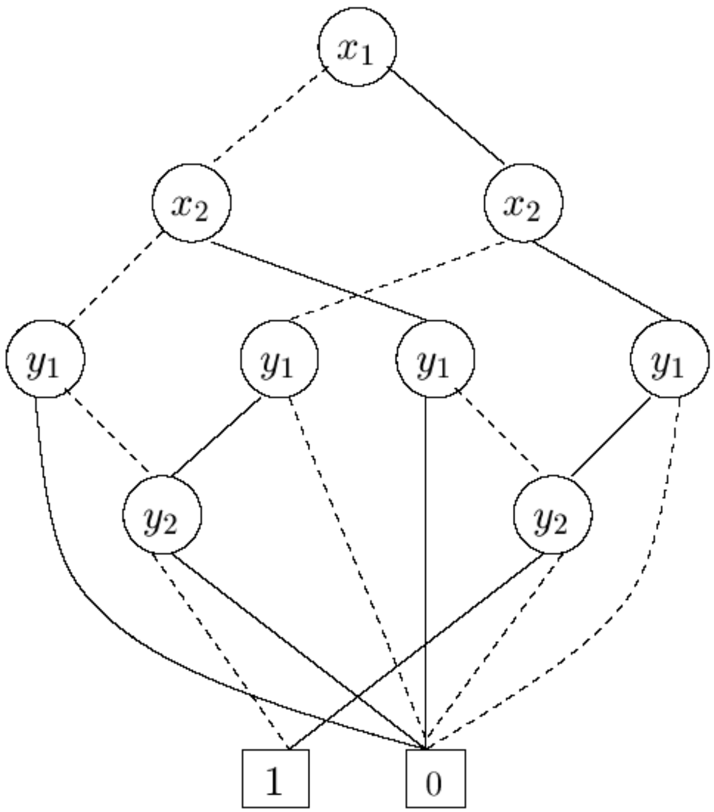
- For any function $f : B^n \rightarrow B$ there is exactly one ROBDD u with variable ordering $x_1 < x_2 < \ldots < x_n$ such that $f^u = f(x_1, \ldots, x_n)$.

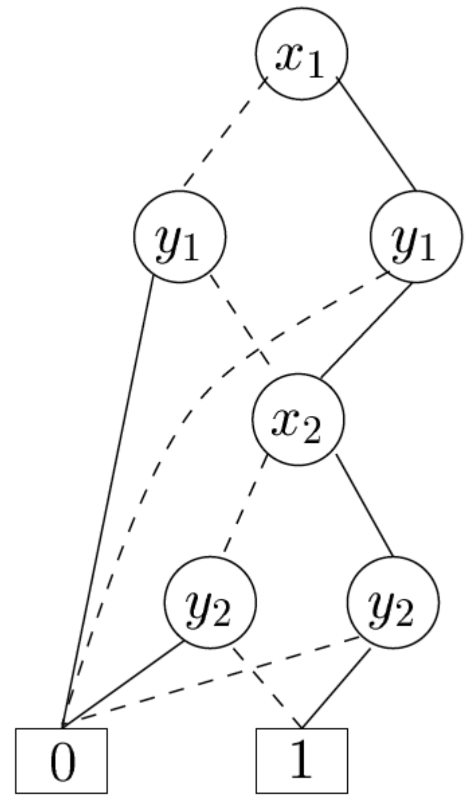
Proof:

Consequences of Canonicity

- How do you represent a tautology?
  (i.e. all variable assignments yield 1)
- How do you know that the function is satisfiable?
  (i.e. there is at least one assignment for the variables such that the function evaluates to 1)

Variable Order



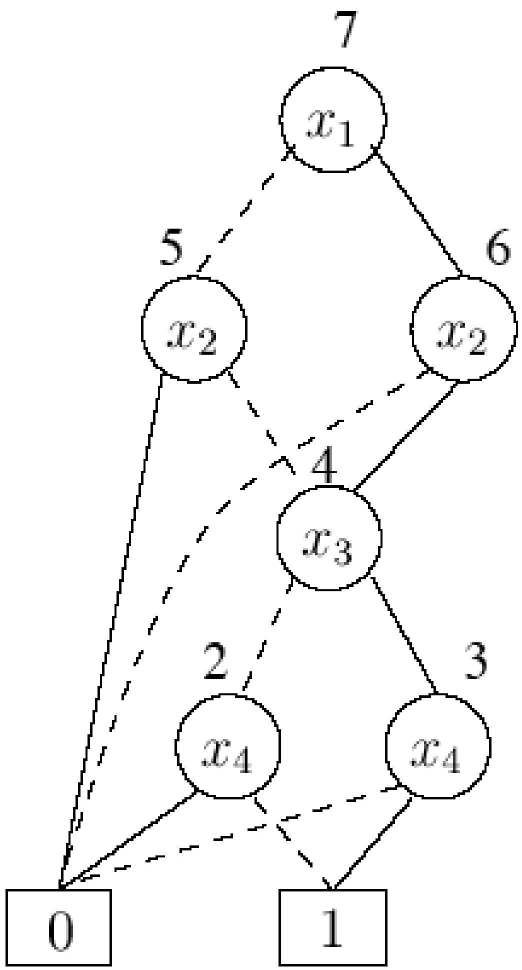


$x_1x_2y_1y_2 \vee \bar{x}_1x_2\ \bar{y}_1y_2 \vee x_1\ \bar{x}_2y_1\ \bar{y}_2 \vee \bar{x}_1\ \bar{x}_2$
$\bar{y}_1\ \bar{y}_2$

Constructing and manipulating BDDs

- Nodes will be represented as numbers 0, 1, 2, … with 0 and 1 reserved for the terminal nodes.
- The variables in the ordering $x_1 < x_2 < \ldots < x_n$ are represented by their indices 1, 2, …, n.
- The ROBDD is stored in a table $T: u \rightarrow (i, l, h)$ which maps a node u to its three attributes var(u) = i, low(u) = l, and high(u) = h.

Example



$T : u \mapsto (i, l, h)$

u	var	low	$high$
0	5		
1	5		
2	4	1	0
3	4	0	1
4	3	2	3
5	2	4	0
6	2	0	4
7	1	5	6

The H Table

- In order to ensure that the OBDD being constructed is reduced, it is necessary to determine from a triple (i, l, h) whether there exists a node u with var(u) = i, low(u) = l, and high(u) = h. For this purpose we assume the presence of a table H : (i, l, h) → u mapping triples (i, l, h) of variable indices i, and nodes l, h to nodes u. The table H is the “inverse" of the table T, i.e., for variable nodes u, T(u) = (i, l, h), if and only if, H(i, l, h) = u.

Operations on T and H

$T : u \mapsto (i, l, h)$

$init(T)$	initialize T to contain only 0 and 1
$u \leftarrow add(T, i, l, h)$	allocate a new node u with attributes (i, l, h)
$var(u), low(u), high(u)$	lookup the attributes of u in T

$H : (i, l, h) \mapsto u$

$init(H)$	initialize H to be empty
$b \leftarrow member(H, i, l, h)$	check if (i, l, h) is in H
$u \leftarrow lookup(H, i, l, h)$	find $H(i, l, h)$
$insert(H, i, l, h, u)$	make (i, l, h) map to u in H

The Function Mk

$\textsc{Mk}[T, H](i, l, h)$

$\quad$ **if** $l = h$ **then return** l

$\quad$ **else if** $member(H, i, l, h)$ **then**

$\qquad$ **return** $lookup(H, i, l, h)$

$\quad$ **else** $u \leftarrow add(T, i, l, h)$

$\qquad$ $insert(H, i, l, h, u)$

$\qquad$ **return** u

What is the complexity of Mk?

The Build Function

$\text{BUILD}[T, H](t)$

function $\text{BUILD'}(t, i) =$

    if $i > n$ **then**

        if t is false **then return** 0 **else return** 1

    else $v_0 \leftarrow \text{BUILD'}(t[0/x_i], i+1)$

        $v_1 \leftarrow \text{BUILD'}(t[1/x_i], i+1)$

        return $\text{MK}(i, v_0, v_1)$

end BUILD'

return $\text{BUILD'}(t, 1)$

Example

- Show build Build(A $\oplus$ B $\oplus$ C)

- What is the running time of Build?
- Can it be improved?
- How?

Apply

All the binary Boolean operators on ROBDDs are implemented by the same general algorithm $\textsc{apply}(op, u_1, u_2)$ that for two ROBDDs computes the ROBDD for the Boolean expression t^{u_1} *op* t^{u_2}. The construction of $\textsc{apply}$ is based on the Shannon expansion (2):

$$t \;=\; x \to t[1/x], t[0/x]\,.$$

Observe that for all Boolean operators *op* the following holds:

$$(x \to t_1, t_2)\ op\ (x \to t_1', t_2') \;=\; x \to t_1\ op\ t_1',\ t_2\ op\ t_2' \tag{4}$$

The Function Apply

$\text{APPLY}[T, H](op, u_1, u_2)$

```
function APP(u1, u2) =
    if u1 ∈ {0,1} and u2 ∈ {0,1} then u ← op(u1, u2)
  else if var(u1) = var(u2) then
    u ← MK(var(u1), APP(low(u1), low(u2)), APP(high(u1), high(u2)))
  else if var(u1) < var(u2) then
    u ← MK(var(u1), APP(low(u1), u2), APP(high(u1), u2))
  else (* var(u1) > var(u2) *)
    u ← MK(var(u2), APP(u1, low(u2)), APP(u1, high(u2)))
  return u
end APP
return APP(u1, u2)
```

Complexity?

The Function Apply

$\textsc{Apply}[T, H](op, u_1, u_2)$

$\quad init(G)$ ← Dynamic programming

$\quad$ **function** $\textsc{app}(u_1, u_2) =$

$\quad\quad$ **if** $G(u_1, u_2) \neq empty$ **then return** $G(u_1, u_2)$

$\quad\quad$ **else if** $u_1 \in \{0, 1\}$ **and** $u_2 \in \{0, 1\}$ **then** $u \leftarrow op(u_1, u_2)$

$\quad\quad$ **else if** $var(u_1) = var(u_2)$ **then**

$\quad\quad\quad u \leftarrow \textsc{mk}(var(u_1), \textsc{app}(low(u_1), low(u_2)), \textsc{app}(high(u_1), high(u_2)))$

$\quad\quad$ **else if** $var(u_1) < var(u_2)$ **then**

$\quad\quad\quad u \leftarrow \textsc{mk}(var(u_1), \textsc{app}(low(u_1), u_2), \textsc{app}(high(u_1), u_2))$

$\quad\quad$ **else** $(*\ var(u_1) > var(u_2)\ *)$

$\quad\quad\quad u \leftarrow \textsc{mk}(var(u_2), \textsc{app}(u_1, low(u_2)), \textsc{app}(u_1, high(u_2)))$

$\quad\quad G(u_1, u_2) \leftarrow u$

$\quad\quad$ **return** u

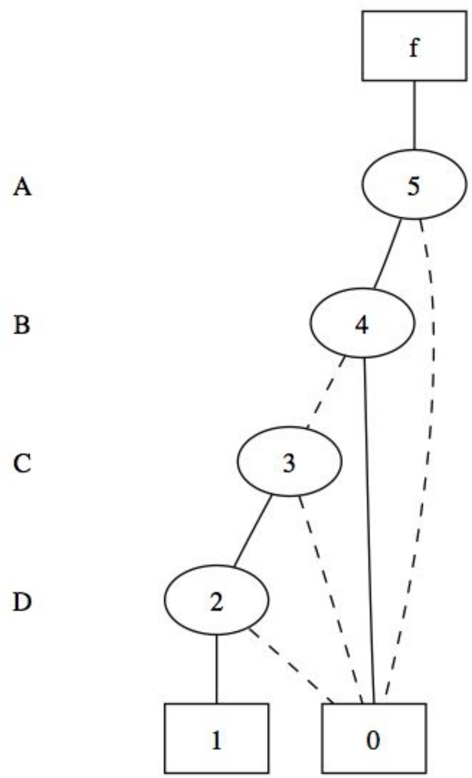
$\quad$ **end** $\textsc{app}$

return $\textsc{app}(u_1, u_2)$

Complexity?

Example

$f = A\ \overline{B}CD$

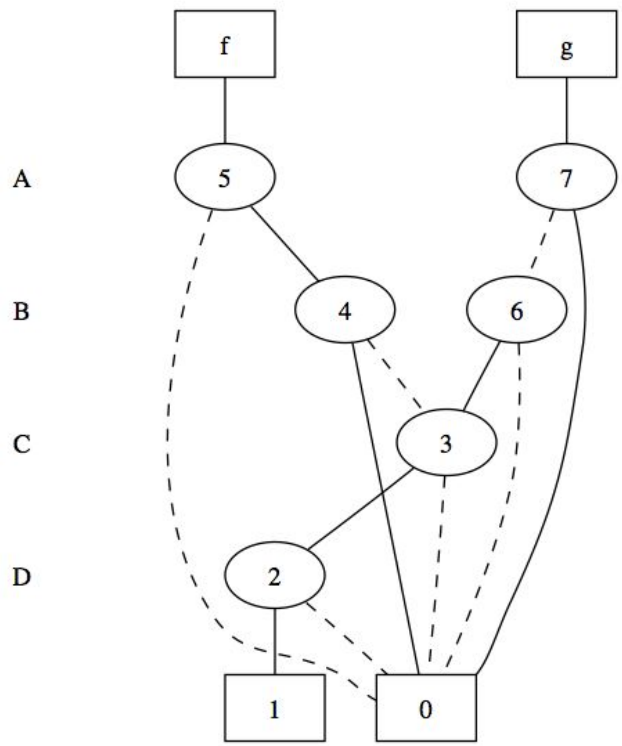


i	v	l	h
0	5		
1	5		
2	4	0	1
3	3	0	2
4	2	3	0
5	1	0	4

add function
$g = \ \overline{A}BCD$

Example

$f = A\,\bar{B}CD$ $g = \bar{A}BCD$

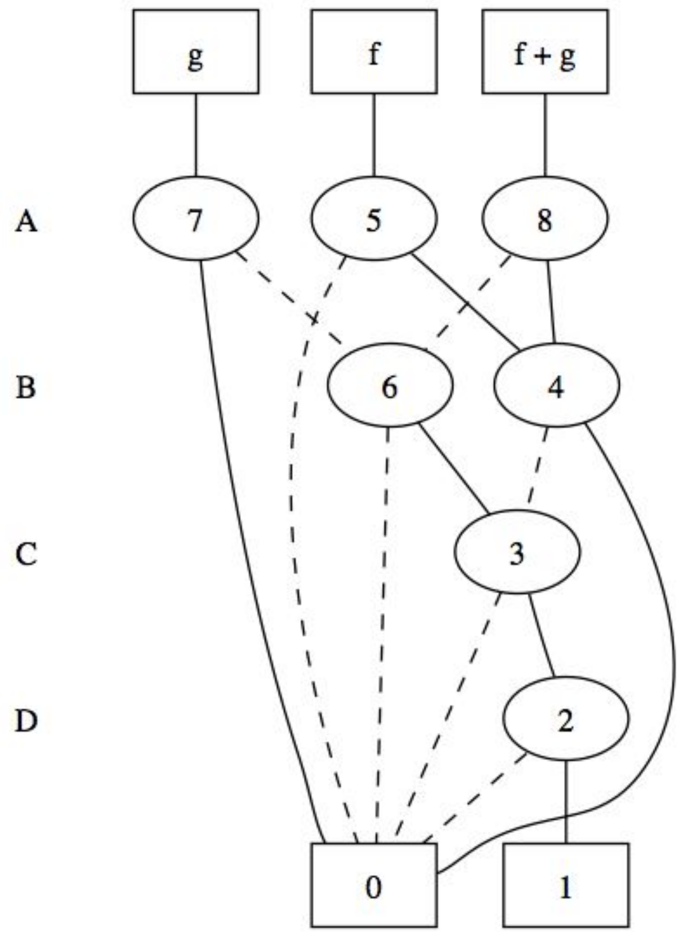


i	v	l	h
0	5		
1	5		
2	4	0	1
3	3	0	2
4	2	3	0
5	1	0	4
6	2	0	3
7	1	6	0

Example

- APPLY(f,g,+)

i	v	l	h
0	5		
1	5		
2	4	0	1
3	3	0	2
4	2	3	0
5	1	0	4
6	2	0	3
7	1	6	0
8	1	6	4

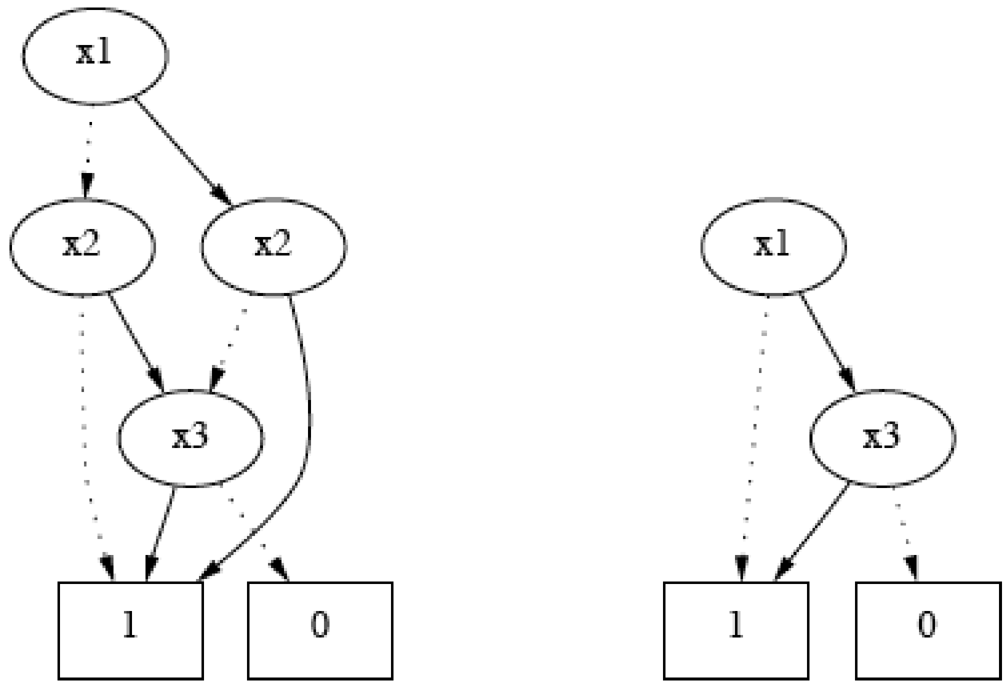


This is a forest (many trees)

How many nodes can a forest have?

Restrict

x2 = 0



Restrict

$\textsc{Restrict}[T, H](u, j, b) =$

function $res(u) =$

  if $var(u) > j$ **then return** u

  else if $var(u) < j$ **then return** $\textsc{mk}(var(u), res(low(u)), res(high(u)))$

  else (* $var(u) = j$ *) **if** $b = 0$ **then return** $res(low(u))$

  else (* $var(u) = j, b = 1$ *) **return** $res(high(u))$

end res

return $res(u)$

SatCount

- Count the number of assignment for which the function is true

$\textsc{SatCount}[T](u)$

function $count(u)$
    if $u = 0$ **then** $res \leftarrow 0$
    else if $u = 1$ **then** $res \leftarrow 1$
    else $res \leftarrow 2^{var(low(u))-var(u)-1} * count(low(u)) + 2^{var(high(u))-var(u)-1} * count(high(u))$
    return res
end $count$

return $2^{var(u)-1} * count(u)$

AnySat

- Find an assignment for which the function is true

$\textsc{AnySat}(u)$

$\quad$ **if** $u = 0$ **then** Error
$\quad$ **else if** $u = 1$ **then return** $[]$
$\quad$ **else if** $low(u) = 0$ **then return** $[x_{var(u)} \mapsto 1, \textsc{AnySat}(high(u))]$
$\quad$ **else return** $[x_{var(u)} \mapsto 0, \textsc{AnySat}(low(u))]$

AllSat

- Find all assignments for which the function is true

$\textsc{AllSat}(u)$

if $u = 0$ **then return** $\langle\ \rangle$
else if $u = 1$ **then return** $\langle\ [\ ]\ \rangle$
else return
$\langle$add $[x_{var(u)} \mapsto 0]$ in front of all
truth-assignments in $\textsc{AllSat}(low(u))$,
add $[x_{var(u)} \mapsto 1]$ in front of all
truth-assignments in $\textsc{AllSat}(high(u))\rangle$

Simplify

```
SIMPLIFY(d, u)
function sim(d, u)
    if d = 0 then return 0
    else if u ≤ 1 then return u
    else if d = 1 then
        return MK(var(u), sim(d, low(u)), sim(d, high(u)))
    else if var(d) = var(u) then
        if low(d) = 0 then return sim(high(d), high(u))
        else if high(d) = 0 then return sim(low(d), low(u))
        else return MK(var(u),
                       sim(low(d), low(u)),
                       sim(high(d), high(u)))
    else if var(d) < var(u) then
        return MK(var(d), sim(low(d), u), sim(high(d), u))
    else
        return MK(var(u), sim(d, low(u)), sim(d, high(u)))
end sim

return sim(d, u)
```

Optimizations

- deleted nodes
  - memory management
- negated edges
- variable reordering
  - sifting