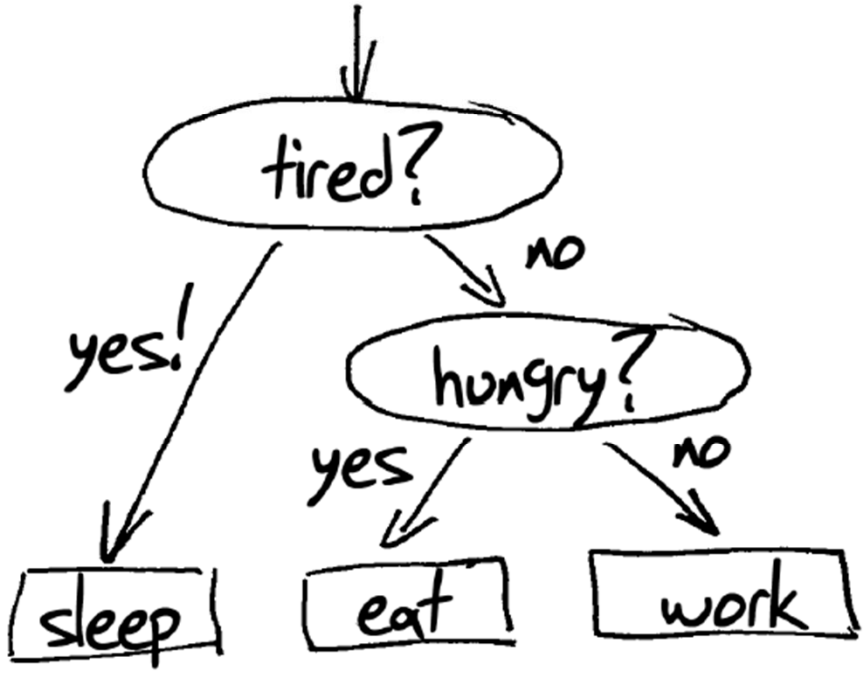
Binary Decision Diagrams (BDDs)

Randal E. Bryant

- *Binary Decision Diagrams* (BDDs) are a symbolic representation of Boolean functions
- Key idea: specific forms of BDDs are canonical
- [Bryant86] is one of the most-cited papers in computer science

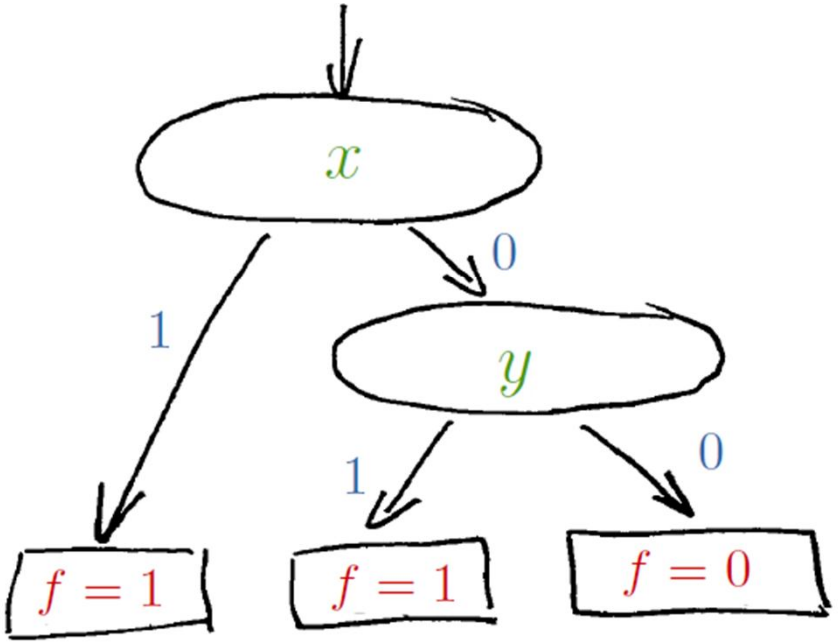
IEEE Transactions on CAD 1986

Decision Diagrams



- Distinguishes terminal from non-terminal nodes
- The edges are labeled with decisions
- The sink nodes (squares) are labeled with the outcome

Decision Diagrams for Functions



- This encodes $f(x, y) = x \vee y$
- Inner nodes: a variable v
- Out-edges: value for v
- Terminal nodes: function value

Decision Diagrams as "if-else" Normal Form

An "if-else" ternary operator: $x \rightarrow y1,y0$

$$x \rightarrow y1,y0 = (x \,.\, y1 + x' \,.\, y0)$$

If-else Normal Form (INF):

A boolean expression built entirely from
constants 1 and 0, positive literals, and if-else operator

Existence of INF for arbitrary boolean expression?

Guaranteed by Shannon Expansion:

$$f(x,\ldots) = (x' \,.\, f|_{x=0}) + (x \,.\, f|_{x=1})$$

$t[x,\ldots] = x \rightarrow t[1/x], t[0/x]$

How to compress Decision Diagrams?

x + x'y

(x → 1, (y → 1,0))

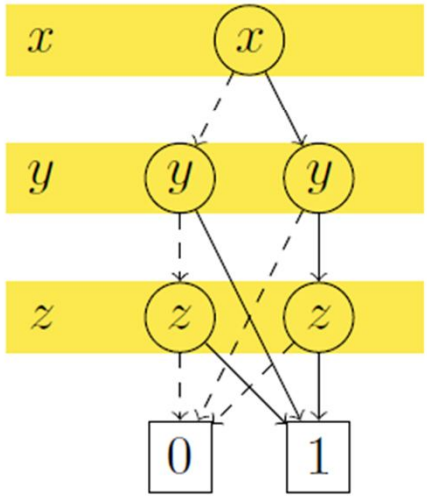
(y → (x → 0, 1), (x → 1,0))

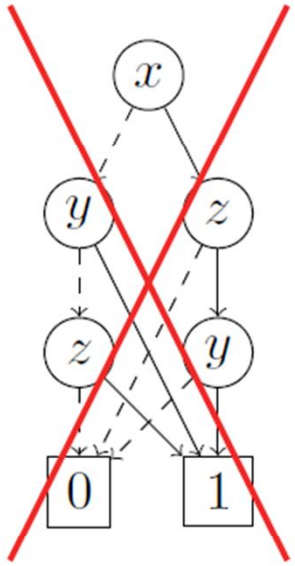
How to make them CANNONICAL:

Unique Normal Form?

Ordered BDDs

You get Cannonicity, not compactness





- Assign arbitrary total ordering to variables, e.g., $x < y < z$.
- Variables must appear in that order along all paths.

Reduced Ordered BDDs (RO-BDDs)

A reduced ordered BDD (RO-BDD) is obtained from an ordered BDD by applying three rules:

#1: Merge equivalent leaf nodes

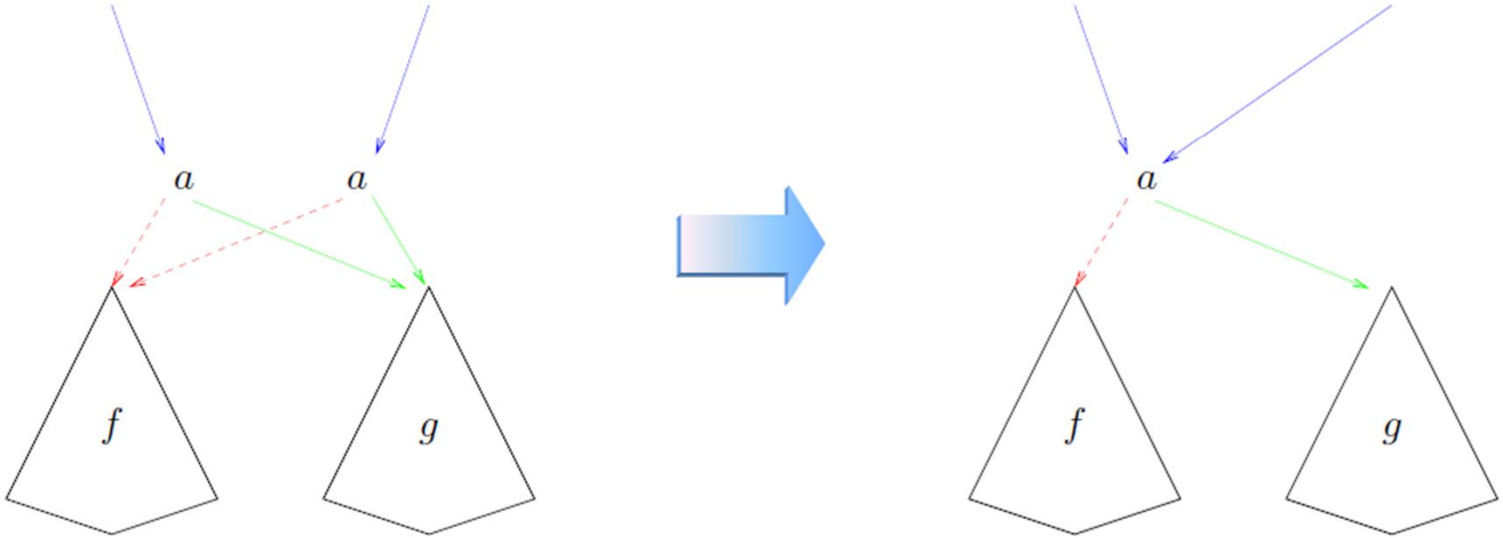
#2: Merge isomorphic nodes

#3: Eliminate redundant tests

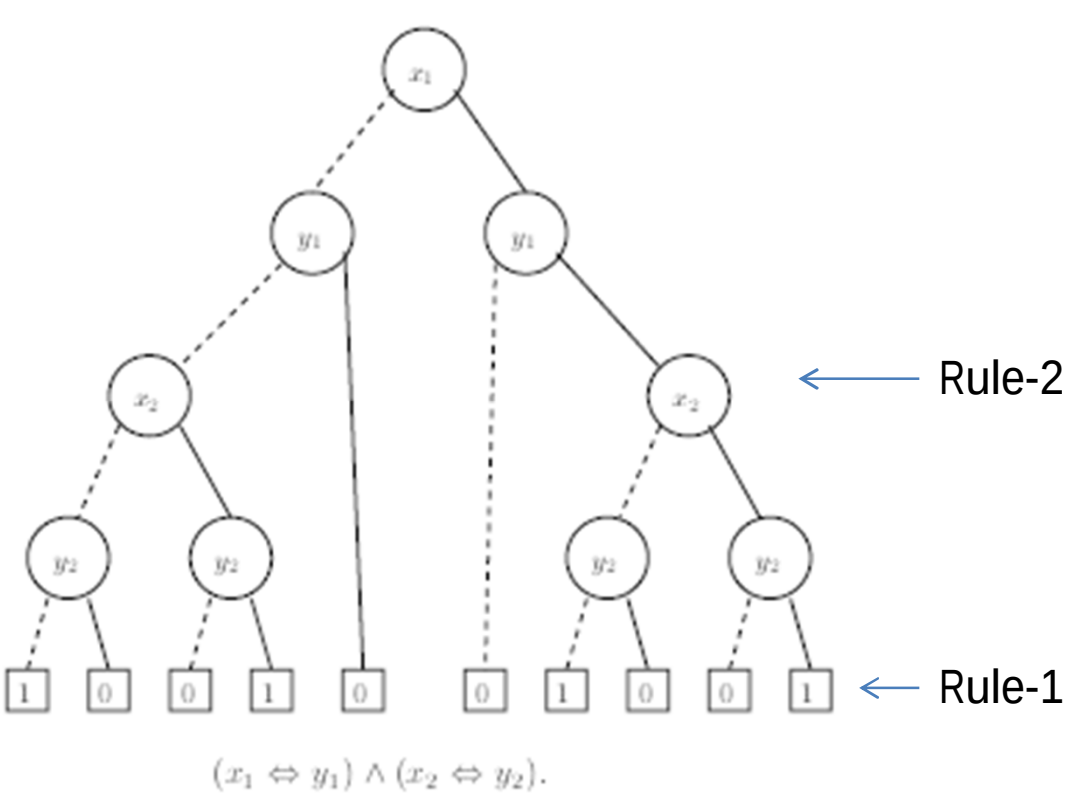
Reduction #1: Equivalent Leaf Nodes

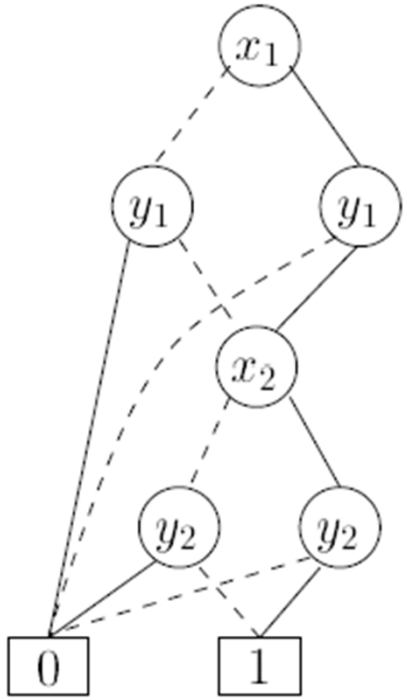
This one is trivial:

Reduction #2: Isomorphic Nodes



Merge nodes with the same variable and the same children


Rule-2
Rule-1
$(x_1 \Leftrightarrow y_1) \wedge (x_2 \Leftrightarrow y_2)$.


x_1
y_1
y_1
x_2
y_2
y_2
0
1

Reduction #3: Redundant Tests

Eliminate nodes where both out-edges go into the same node:

Henceforth, we assume we are dealing only with ROBDDs
and refer to them as just BDDs

Main BDD Theorem

Shannon Expansion: $f(x,\ldots) = (x' \cdot f|x=0) + (x \cdot f|x=1)$

Cannonicity Thm: Fix an arbitrary variable order, then BDD for every boolean function is ***unique***

Proof: By induction on the number (n) of vars in boolean function

Base Case: n =0, i.e., constant function

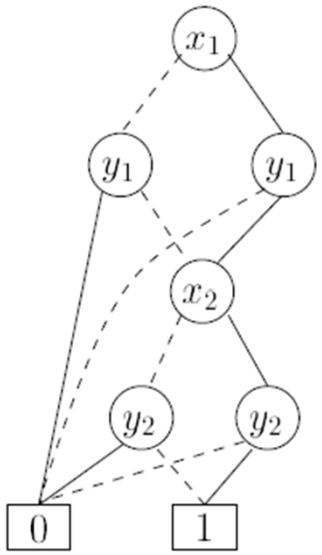
Induction Case:

Assume unique BDDs exist for functions on n variables

Can use rules of reduction to add n+1 var in shannon-order in a unique fashion: How?

Challenge: Finding a variable order that makes the BDD most compact

Some useful observations/facts about BDDs



P1: Variables always occur in the variable ordering on every path

P2: Every path from root to terminal 1 is a SAT assignment

P3: Every node represents a unique boolean function

- A restriction on original function
- A boolean function on a subset of variables

P4a: Every Valid formu1a is identical to 1
P4b: Every UNSAT formulas is identical to BDD 0,
P4c: A formula is SAT if it is not BDD 0

Observation: given a BDD it takes constant time to check whether a formula is

- a tautology,
- inconsistent,
- satisfiable

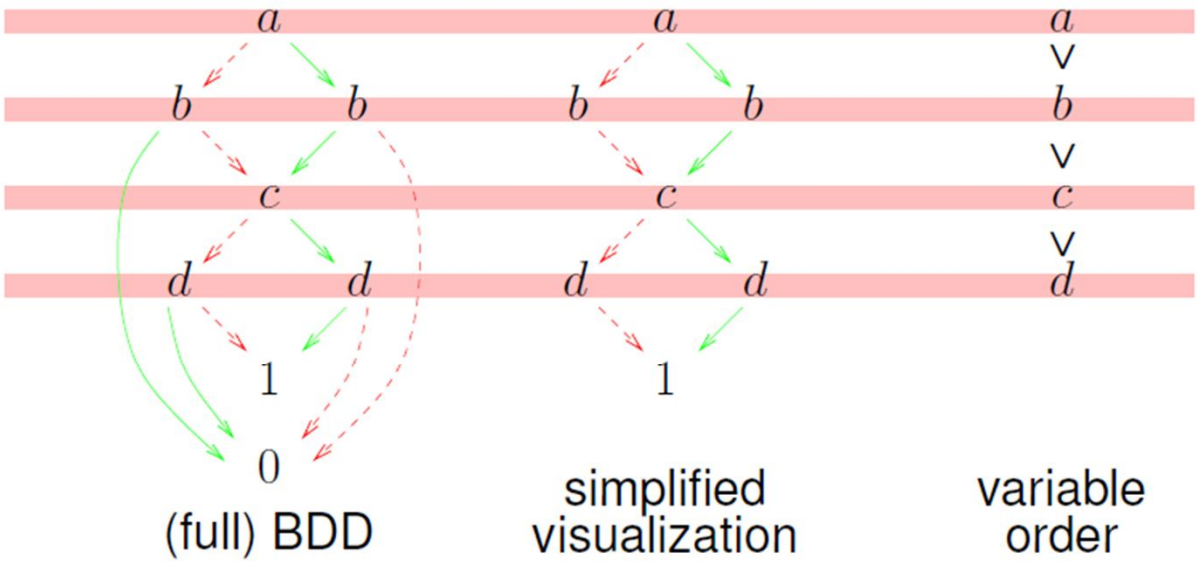
Aren't these hard?

What's the catch???

Variable Ordering and BDD Size

Equality of two 2-bit vectors

$(a, c) = (b, d)$ equivalent to $(a \leftrightarrow b) \wedge (c \leftrightarrow d)$



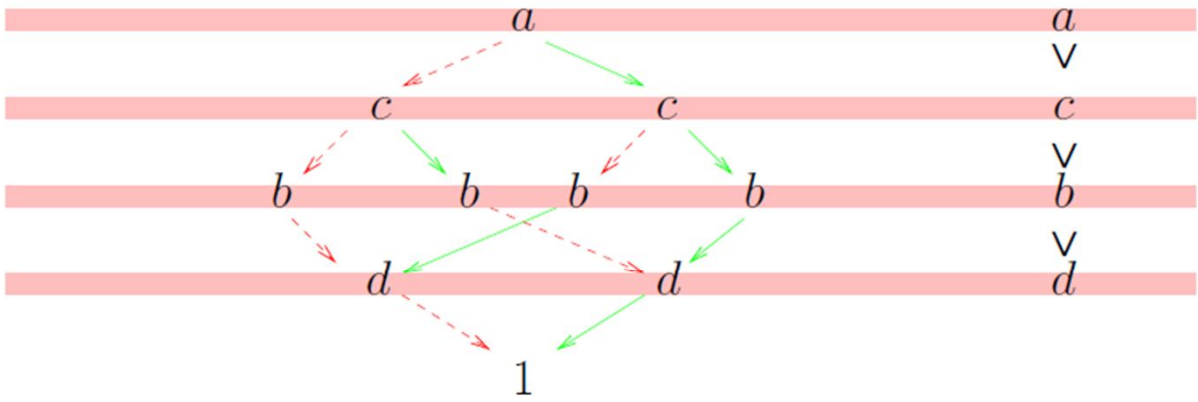
(full) BDD | simplified visualization | variable order

unique BDD for (one) optimal, **interleaved** variable ordering
(linear in the width of the bit vectors)

Variable Ordering and BDD Size

Equality of two 2-bit vectors

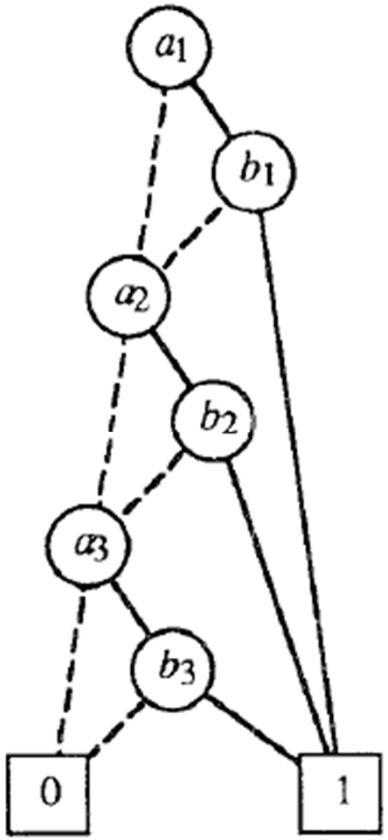
$(a, c) = (b, d)$ equivalent to $(a \leftrightarrow b) \wedge (c \leftrightarrow d)$

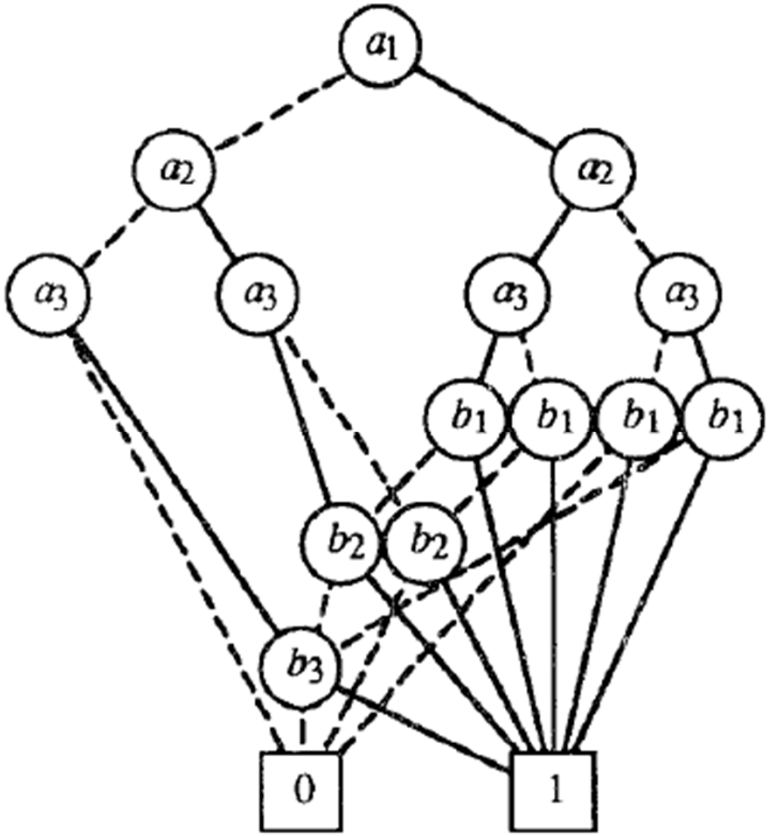


unique BDD for (one) worst case, **blocked** variable ordering
(exponential in the width of the bit vectors)

Impact of Variable Ordering

(a1. b1) + (a2 . b2) + (a3 . b3)





a1 <b1 <a2 < b2 < a3 <b3

a1 < a2 < a3 <b1 < b2 < b3

BDD Size Bounds for Some Classes of Functions

Table 1. OBDD Complexity for Common Function Classes

Function Class	Complexity	
	Best	Worst
Symmetric	linear	quadratic
Integer Addition (any bit)	linear	exponential
Integer Multiplication (middle bits)	exponential	exponential

Courtesy: [Bryant, R., ACM Computing Surveys, 1992]

Selecting a Good Ordering

✘ Intractable problem

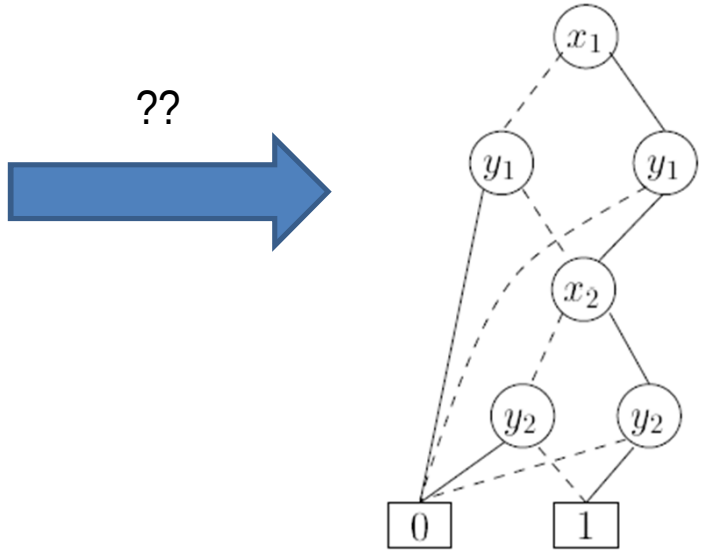
✘ Even when input is an OBDD
(i.e., to find optimum improvement to current ordering)

✔ Application-based heuristics
- Exploit characteristics of application
- E.g., ordering for functions of combinational circuit: traverse circuit graph depth-first from outputs to inputs

A Time Efficient Algorithm to Construct BDDs?

Given

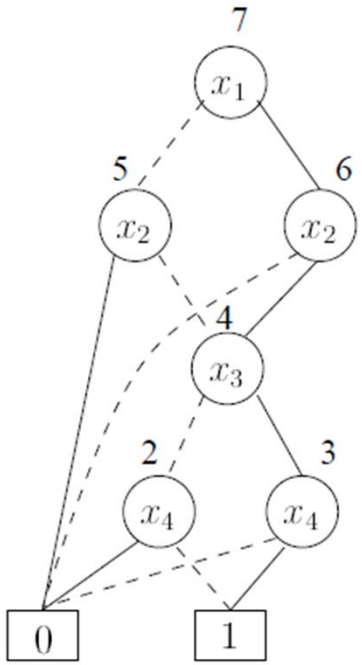
1. (x1 $\leftrightarrow$ y1) (x2 $\leftrightarrow$ y2)
2. x1 < y1 < x2 < y2



Requirements:

1. Suitable data structures for BDDs
2. Apply operations, e.g., negation, +, etc., on two existing BDDs

Data Structures for BDD



Node Table

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

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

Lookup Table

H: (i,l,h) → u

(i,l,h)	u
(1,5,6)	7
(2,4,0)	5
(4,0,1)	3
(5,*,*)	1
(2,0,4)	6
(3,2,3)	4
(4,1,0)	2
(5,$,$)	0

Nodes are uniquely numbered 0,1,2,3....
(with 0,1 denoting terminals)

Variables are numbered 1,2,3...n (as per chosen ordering),
(with terminals assigned n+1)

Data Structures for BDD

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

$init(T)$	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

Index creation for tables can be implemented as "hash" functions s.t. basic operations can be done in constant time

Lookup Table

H: (i,l,h) → u

(i,l,h)	u
(1,5,6)	7
(2,4,0)	5
(4,0,1)	3
(5,*,*)	1
(2,0,4)	6
(3,2,3)	4
(4,1,0)	2
(5,$,$)	0

Node Table

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

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

MK: Adding a node to BDD structure

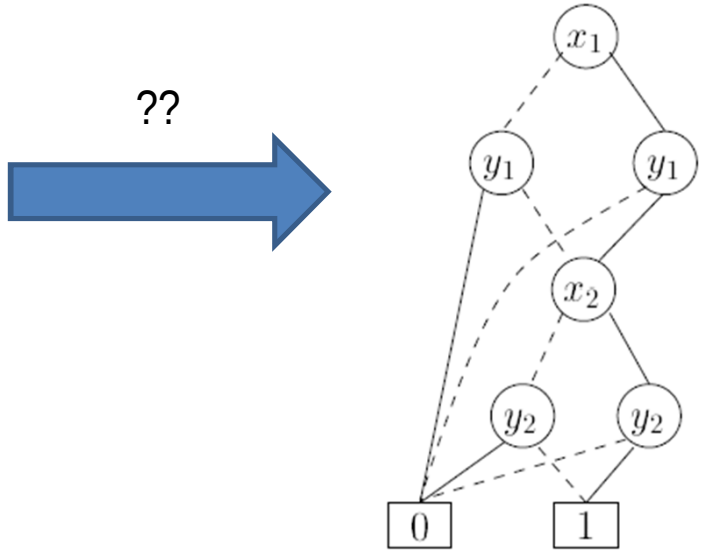
```
MK[T, H](i, l, h)
1:   if l = h then return l
2:   else if member(H, i, l, h) then
3:        return lookup(H, i, l, h)
4:   else u ← add(T, i, l, h)
5:        insert(H, i, l, h, u)
6:        return u
```

1. **Line 1: checks for redundancy**
2. **Line 2: A node is added to the table only if it doesn't exist**
   → unique BDD in graph for every boolean function
3. **Both T and H tables are updated**

A Time Efficient Algorithm to Construct BDDs?

Given

1. (x1 $\leftrightarrow$ y1) (x2 $\leftrightarrow$ y2)
2. x1 < y1 < x2 < y2



Requirements:

1. Suitable data structures for BDDs
2. Apply operations, e.g., negation, +, etc., on two existing BDDs

Recursion Recap

```
fact(n)
  if n = 0 then return 1
  else result ← (n * fact (n-1))
  return result
end fact
```

```
fib (n)
  if n <=1 then return 1
  else result ← (fib(n-1) * fib(n-2))
  return result
end fib
```

```
fibo(n)
  save_result[0] = 1; save_result[1] = 1
  function fib_alt(n)
     if save_result(n) = defined then result ← save_result(n)
     else result ← (fib(n-1) * fib(n-2))
     save_result(n) ← result
     return result
   end fib_alt

 return fib_alt(0)
end fibo
```

Build: Bottom-up BDD construction

```
BUILD[T, H](t)
1:   function BUILD'(t, i) =
2:       if i > n then
3:           if t is false then return 0 else return 1
4:       else v_0 ← BUILD'(t[0/x_i], i + 1)
5:           v_1 ← BUILD'(t[1/x_i], i + 1)
6:           return MK(i, v_0, v_1)
7:   end BUILD'
8:
9:   return BUILD'(t, 1)
```

Basis for the recursive algorithm:

Shannon Expansion:

f(x,...) = (x' . f|x=0) + (x . f|x=1)

t[x,...] = x → t[1/x], t[0/x]

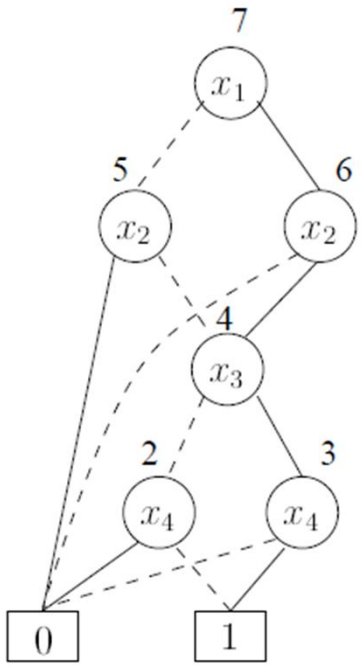
Not efficient:

O(2^n) recursive calls ,

n = #vars

Binary Operations on BDDs
(*For top-down efficient construction*)

(x1 ↔ y1).(x2 ↔ y2)



x1 . y2’

Some illustrative examples of BDD ops

1. x + x'

2. x + 0

3. x + y'

Laws of if-else operator that form basis for above manipulation:

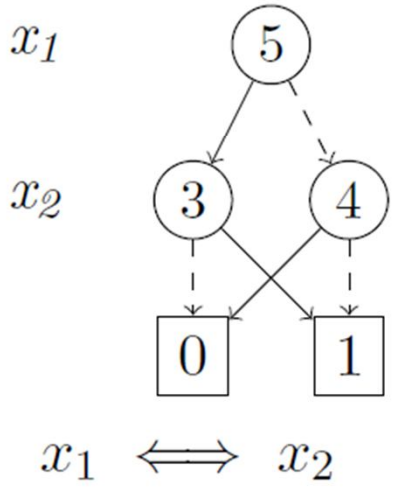
(x -> t1, t2) op (x → s1, s2)
≡ (x → t1 op s1, t2 op s2)

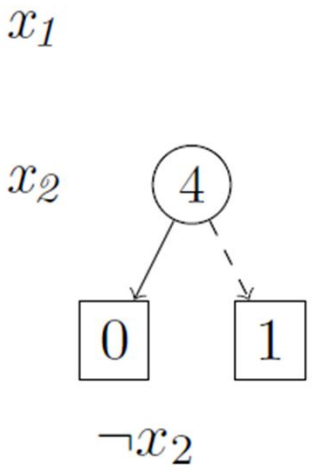
t op (x1 → s1, s2) ≡ (x1 → t op s1, t op s2)
(x1 → s1, s2) op t ≡ (x1 → s1 op t, s2 op t)

Apply: Binary operation on two BDDs

```
function APPLY(u1, u2) =
    if u1, u2 ∈ {0, 1} then
        u := u1 ⋆ u2;
    else if var(u1) = var(u2) then
        u := MK(var(u1), APPLY(low(u1),low(u2)),
                         APPLY(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;
```

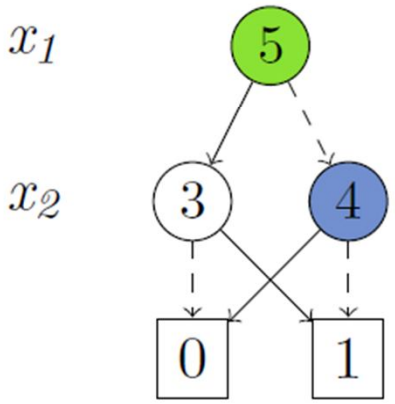
Example





Now compute BDD for (x1 ↔ x2) + x2' using APPLY!!

Example (cont.)



APPLY(5, 4):

var(5)=1, var(4)=2

MK(var(5), APP(low(5),4), APP(high(5),4));

MK(1, APP(4,4), APP(3,4)); MK(1, 4, 1);

APP(4, 4):

var(4)=var(4)=2

MK(2, APP(1,1), APP(0,0));

MK(2, 1, 0); = 4!

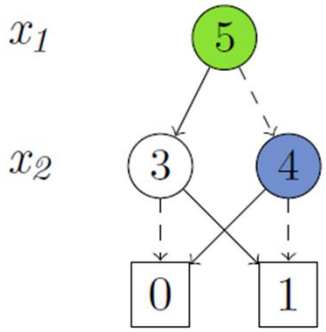
APP(3, 4):

var(3)=var(4)=2

MK(2, APP(0,1), APP(1,0));

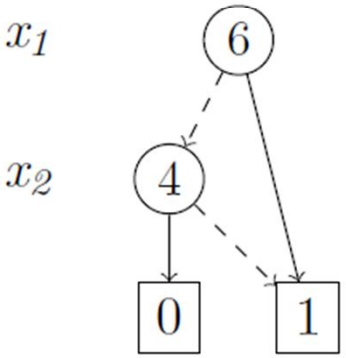
MK(2, 1, 1); = 1!

Example (cont.)



Original BDDs

Final result:



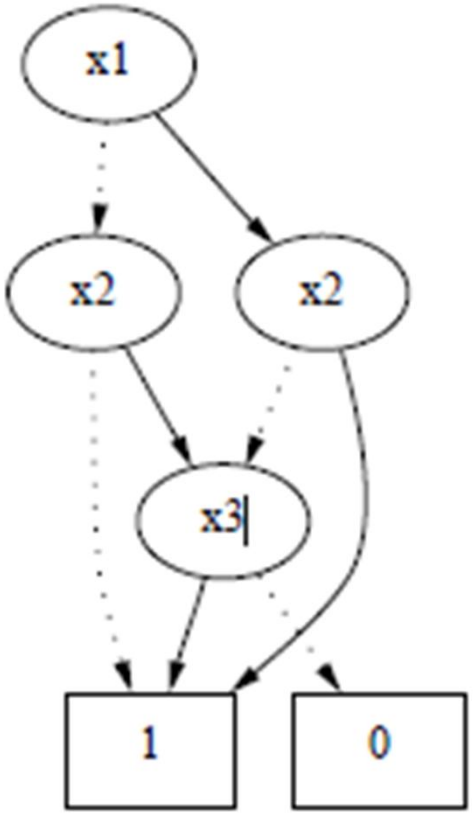
#	*var*	*low*	*high*	
0	3			
1	3			
2	–	–	–	
3	–	–	–	3 → (2,0,1)
4	$2\,x_2$	1	0	
5	–	–	–	5 → (1,3,4)
6	$1\,x_1$	4	1	

Optimized version of Apply

```
APPLY[T, H](op, u1, u2)
1:  init(G)
2:
3:  function APP(u1, u2) =
4:    if G(u1, u2) ≠ empty then return G(u1, u2)
5:    else if u1 ∈ {0, 1} and u2 ∈ {0, 1} then u ← op(u1, u2)
6:    else if var(u1) = var(u2) then
7:          u ← MK(var(u1), APP(low(u1), low(u2)), APP(high(u1), high(u2)))
8     else if var(u1) < var(u2) then
9           u ← MK(var(u1), APP(low(u1), u2), APP(high(u1), u2))
10:   else (* var(u1) > var(u2) *)
11:         u ← MK(var(u2), APP(u1, low(u2)), APP(u1, high(u2)))
12:   G(u1, u2) ← u
13:   return u
14: end APP
15:
16: return APP(u1, u2)
```

Time complexity: O(|u1| * |u2|)

Restrict operation Illustration



BDD for f(x1, 0, x3)?

BDD for f(1, x2, x3)?

$f(x1, x2, x3) = (x1 \leftrightarrow x2) + x3$

Steps for computing BDD for Restriction:

1. Look for the nodes associated with variable x
2. Eliminate the nodes directing all incoming edges towards b-edge of the eliminated node

Existential Quantification

For BDD f and a variable x compute BDD for

$$\exists\, x.\ f$$

This is equivalent to Apply(+, Restrict(1/x, f), Restrict(0/x, f))

- Two RESTRICT operations
- One call to APPLY

BDD: Summary

- DAG representation of boolean functions
  - Every boolean function has a unique RO-BDD
  - sat,unsat, and even equivalence checking are constant time
- Efficient manipulation algorithms for operations
- Need to find good orderings for compactness
- Widely used in commercial EDA tools for verification and synthesis, especially in hardware design