VE527

Computer-Aided Design of Integrated Circuits

Binary Decision Diagram

Outline

- Binary Decision Diagram (BDD): Introduction
- BDD Reduction
- BDDs for Common Functions
- Sharing in BDD
- Real Way to Build BDD
- Applications and Issues of BDD

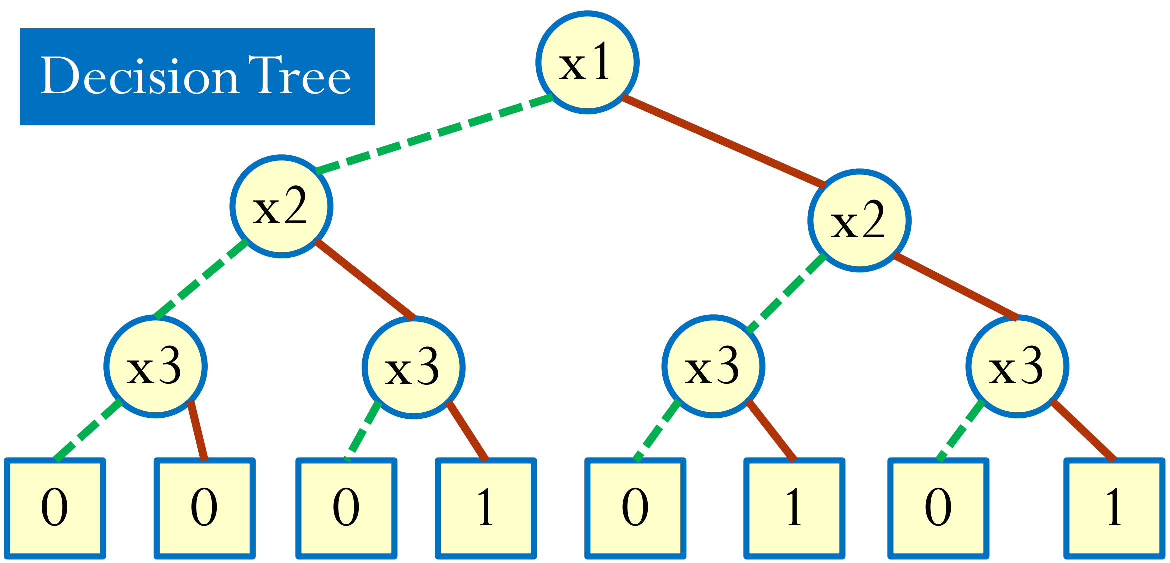
Binary Decision Diagrams (BDD)

- Originally studied by several people
- … got **practically useful** in 1986
  - Randal Bryant of CMU made breakthrough on **Reduced Ordered BDD (ROBDD)**.
- Reference
  - R. Bryant, "Symbolic Boolean manipulation with ordered binary decision diagrams," ACM Computing Surveys, vol. 24, no. 3, pp. 293–318, 1992.

Binary Decision Diagrams for Truth Tables

- Big Idea #1: **Binary Decision Diagram**
  - Turn a truth table for the Boolean function into a **Decision Diagram**.
  - In simplest case, graph is just a **tree**.
  - **Vertex** represents a **variable**.

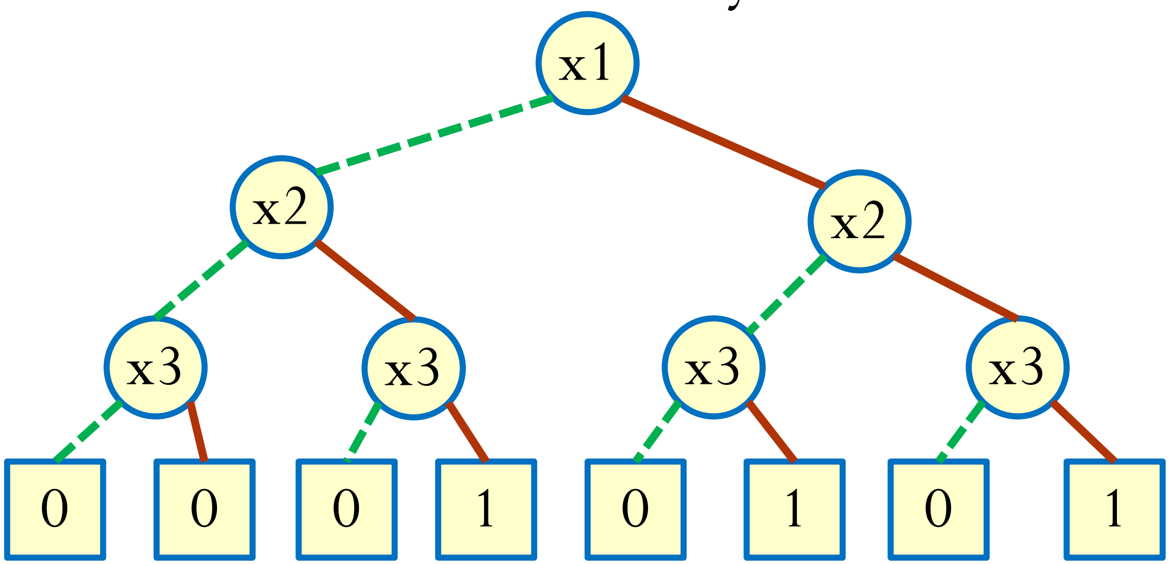
x1	x2	x3	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1



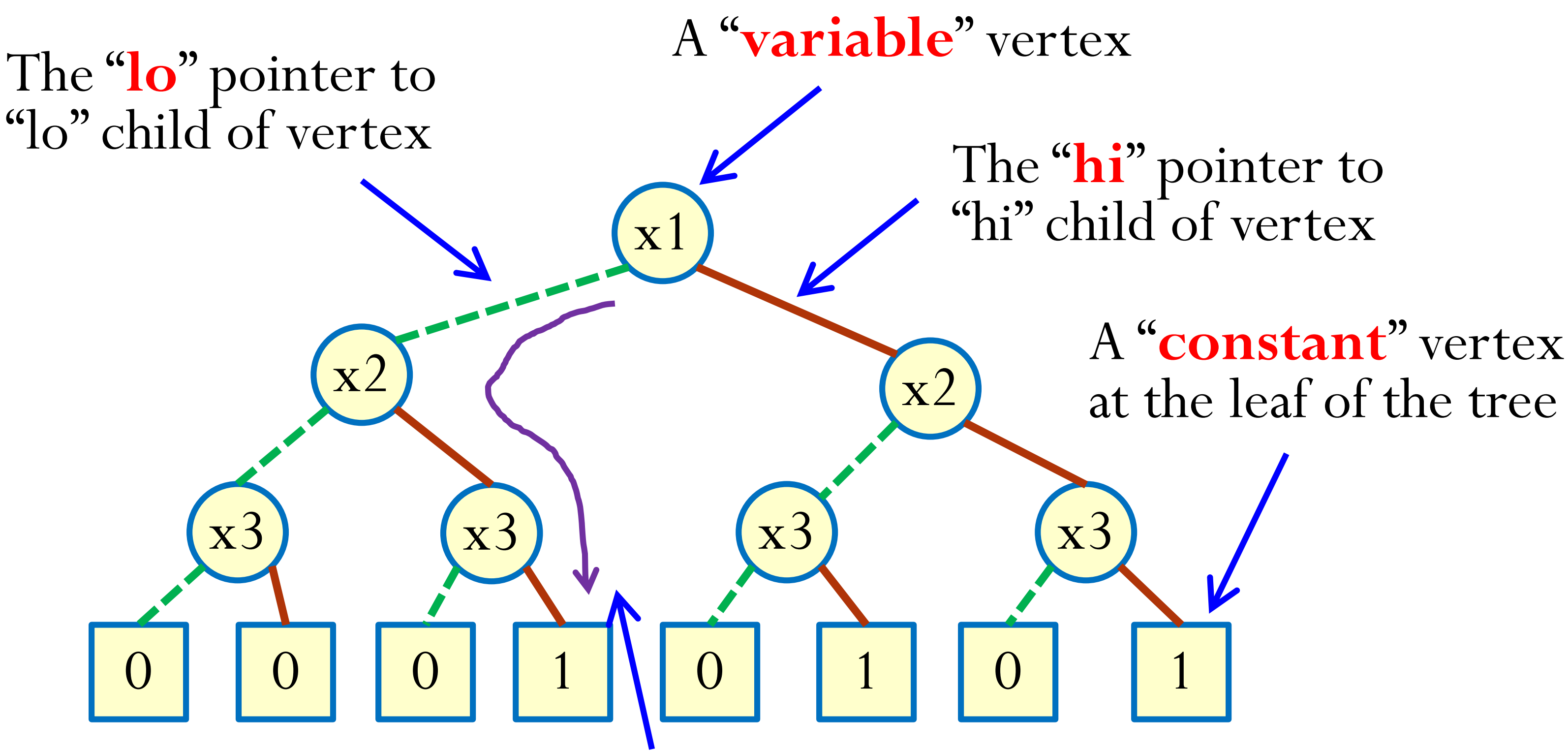
Binary Decision Diagrams

- **Edge** out of a vertex is a **decision** (0 or 1) on that variable.
  - Follow **green dashed** line for 0.
  - Follow **red solid** line for 1.
  - By convention, don't draw arrows on the edges, we know where they go.
- Function value determined by **leaf value**.

x1	x2	x3	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1



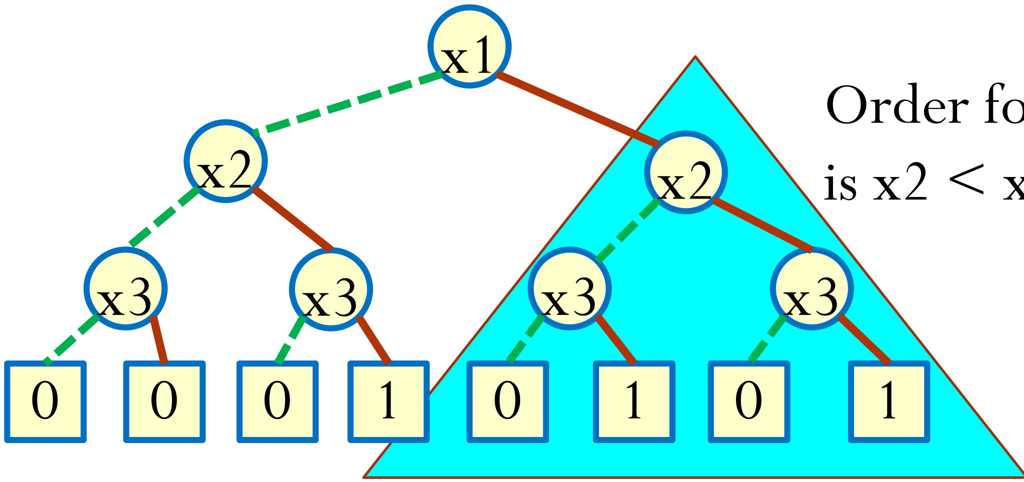
Binary Decision Diagrams
Some Terminology



The '**variable ordering**', which is the order in which decisions about variables are made. Here, it is x1 < x2 < x3.

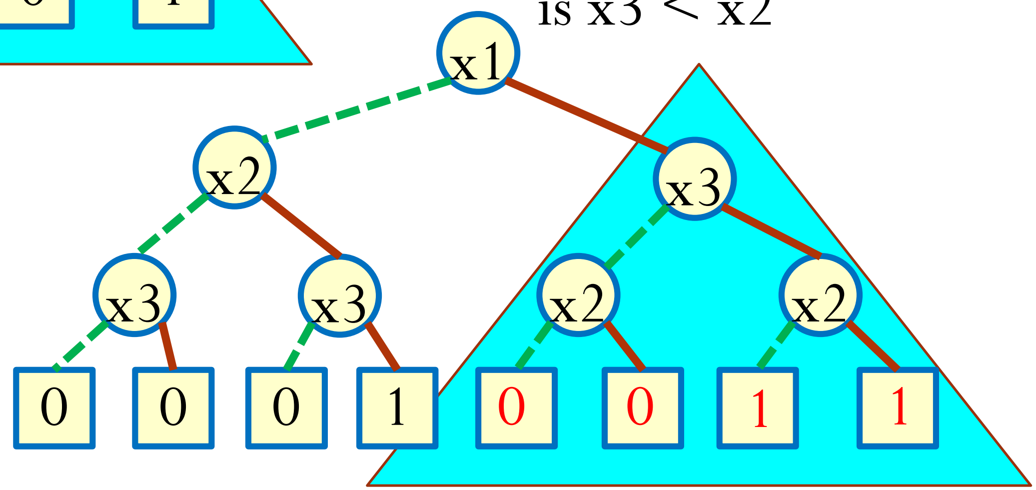
Ordering

- Different variable orders are possible.



Order for this subtree is x2 < x3

Order for this subtree is x3 < x2



Binary Decision Diagrams
Observations

- Each path from root to leaf traverses variables in **some** order.
- Each such path constitutes **a row** of the truth table, i.e., a decision about what output is when variables take these particular values.
- However, we have not yet specified anything about the **order** of decisions.
- The decision diagram is **NOT unique** for this function.

Terminology: Canonical form

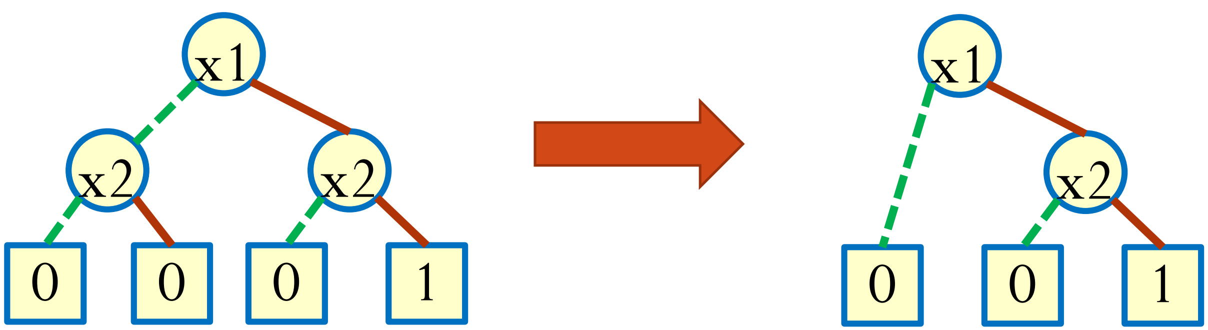
- A representation of Boolean function that is unique for each Boolean function
  - Same function of same variables always produces this exact **same** representation.
  - Does not depend on gate-level implementation

- Example: a truth table is **canonical** (up to variable order).

- We want a canonical-form data structure(!)
  - Very useful! E.g., verification

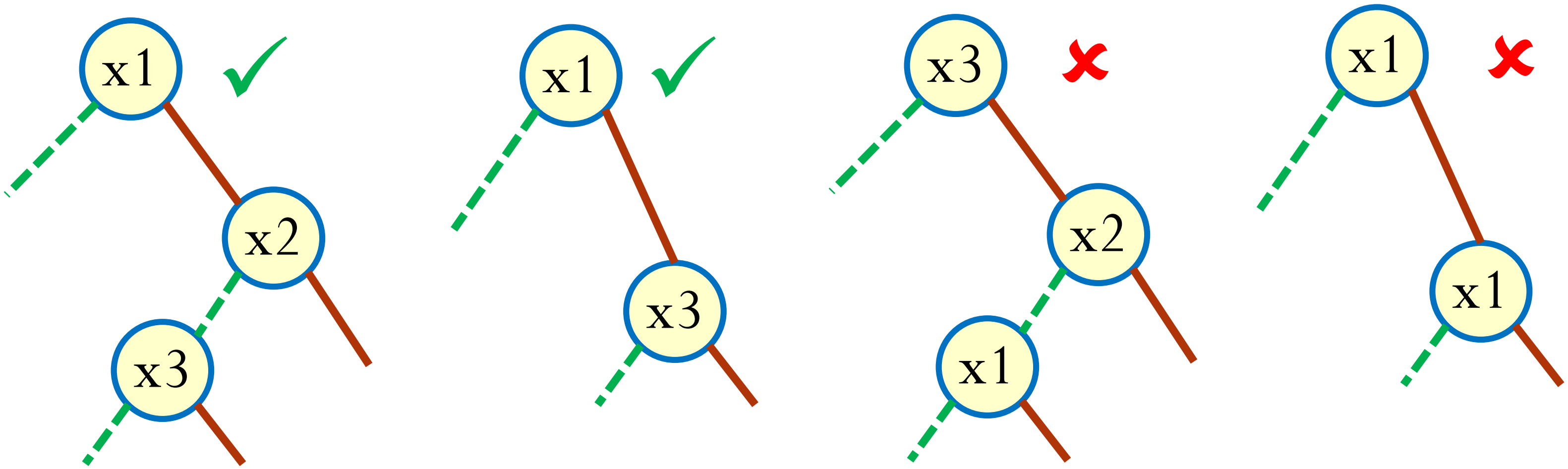
Binary Decision Diagrams

- What's wrong with this diagram representation?
  - It is **not canonical**, and it is way **too big** to be useful (it is as big as truth table!)
- Big idea #2: **ordering**
  - Restrict **global ordering** of variables.
  - It means: **every path** from root to a leaf visits variables in the **SAME** order.
  - Note: it is OK to **omit** a variable if you don't need to check it to decide which leaf node to reach for final value of function.



Ordering BDD Variables: Example

- Assign (an arbitrary) **global ordering** to vars: x1 < x2 < x3
  - Variables must appear in this specific order along all paths; ok to skip vars



- Property: each variable assignment corresponds to only one path from the root to the leaf. Thus, no **conflicting** result.

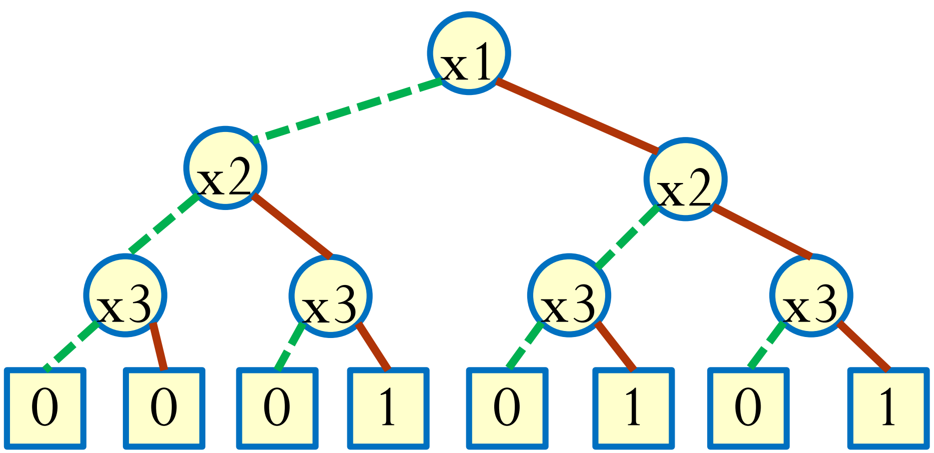
Outline

- Binary Decision Diagram (BDD): Introduction
- BDD Reduction
- BDDs for Common Functions
- Sharing in BDD
- Real Way to Build BDD
- Applications and Issues of BDD

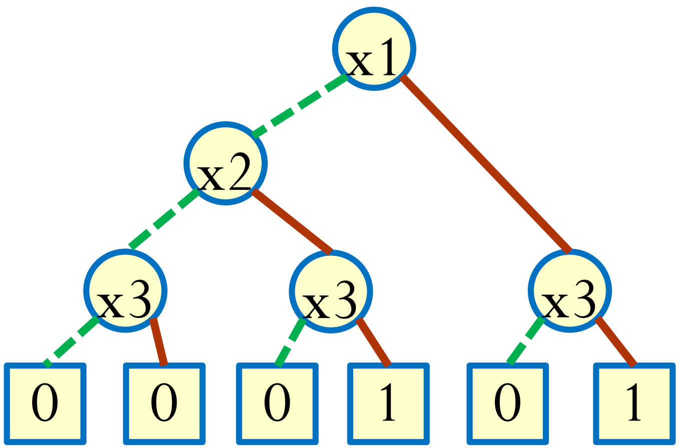
Binary Decision Diagrams

- OK, now what's wrong with it?
  - Variable ordering simplifies things, but still **too big**, and **not canonical**.

Original Decision Diagram

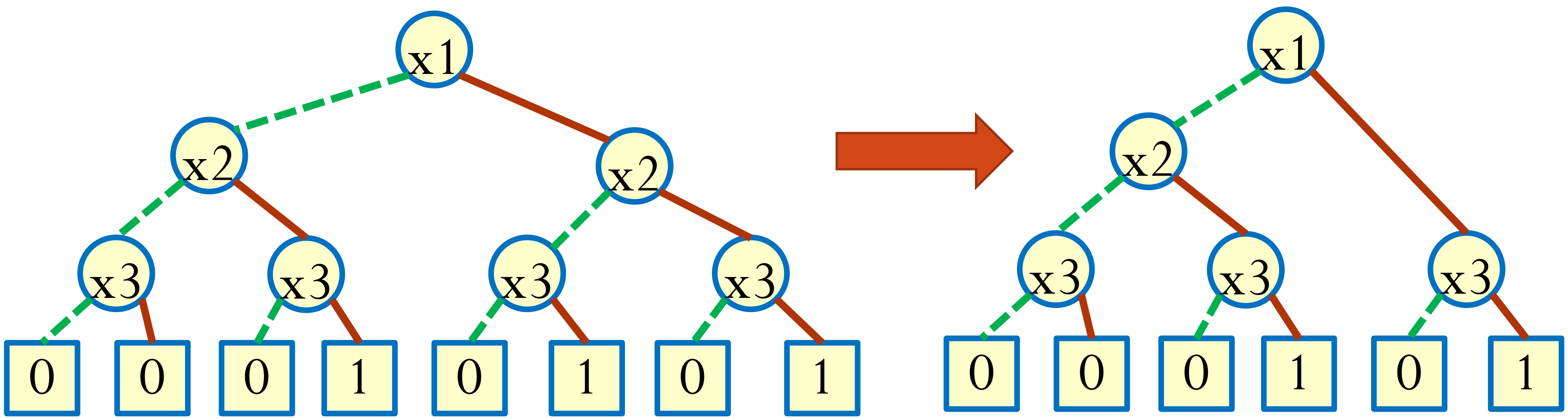


Equivalent, but Different Decision Diagram



Binary Decision Diagrams

- Big Idea #3: **Reduction**
  - Identify **redundancies** in the graph that can remove unnecessary nodes and edges.
  - Removal of x2 node and its children, replace with x3 node is an example of this.



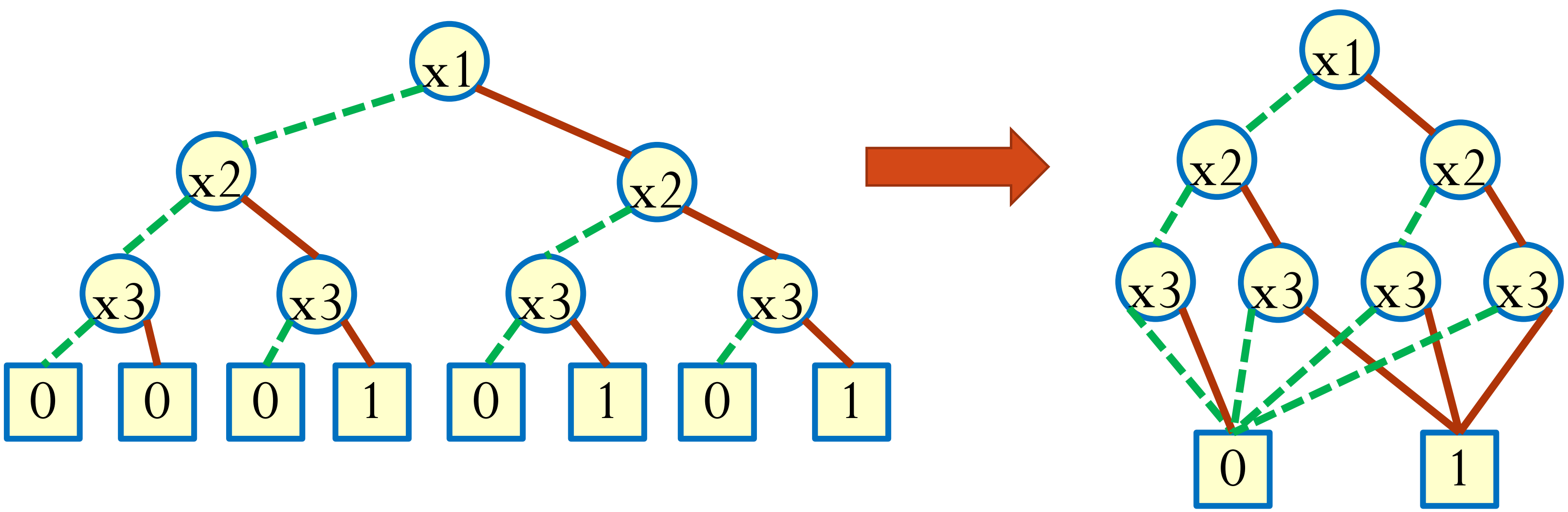
Binary Decision Diagrams

Reduction

- Why are we doing this?
  - **Graph size**: Want result as **small** as possible.
  - **Canonical form**: For same function, given same variable order, want there to be **exactly one** graph that represents this function.

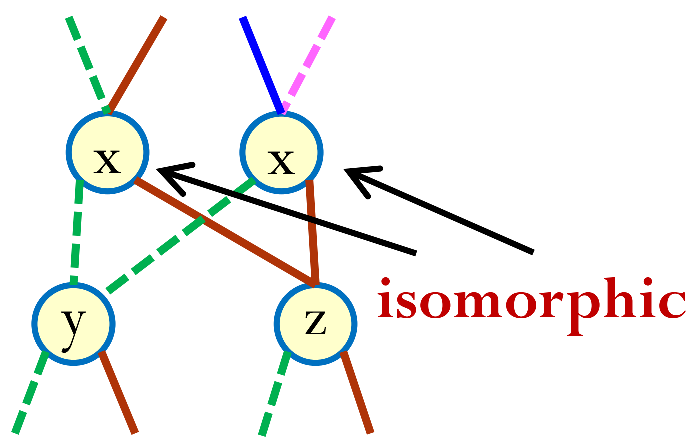
Reduction Rules

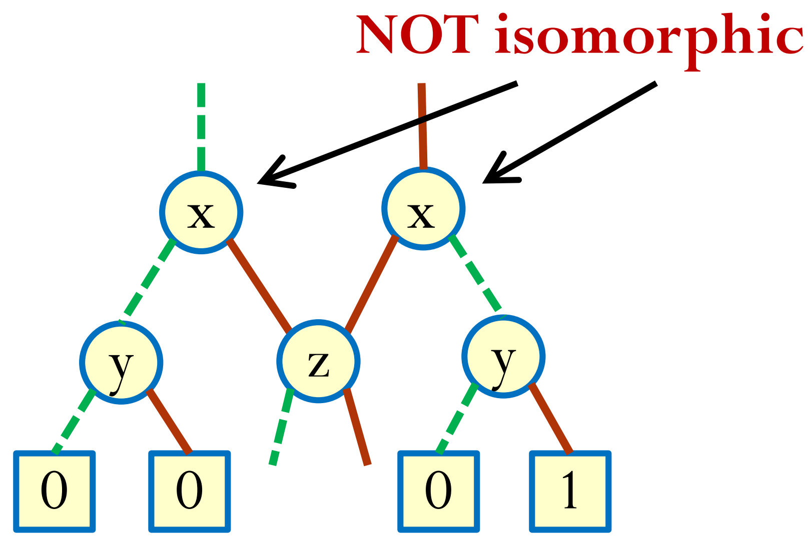
- **Reduction Rule 1**: **Merge equivalent <u>leaves</u>**
  - Just keep one copy of each **constant leaf** – anything else is totally wasteful.
  - Redirect all edges that went into the redundant leaves into this one kept node.
- Apply Rule 1 to our example…



Reduction Rules

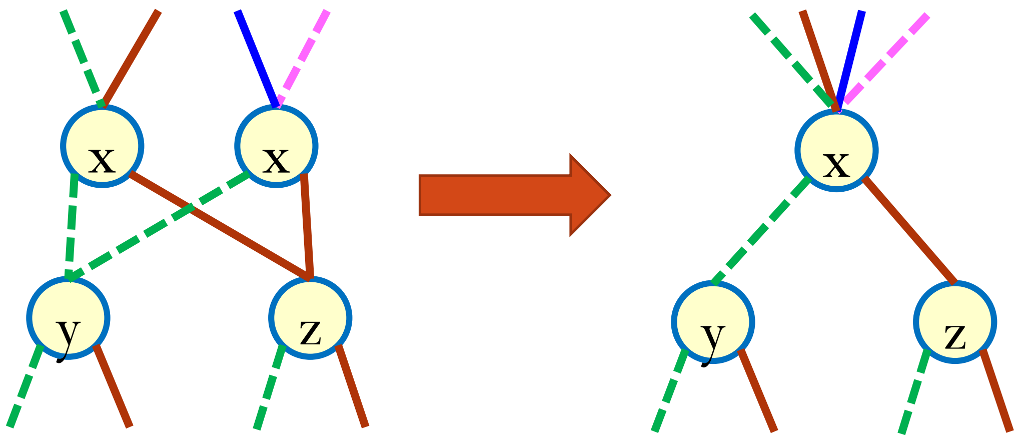
- **Reduction Rule 2**: **Merge isomorphic nodes**
- Isomorphic = 2 nodes with **same** variable and **identical** children
  - Cannot distinguish these nodes on how they contribute to decisions in graph.
  - Note: means exact same physical child nodes, not just children with same label.





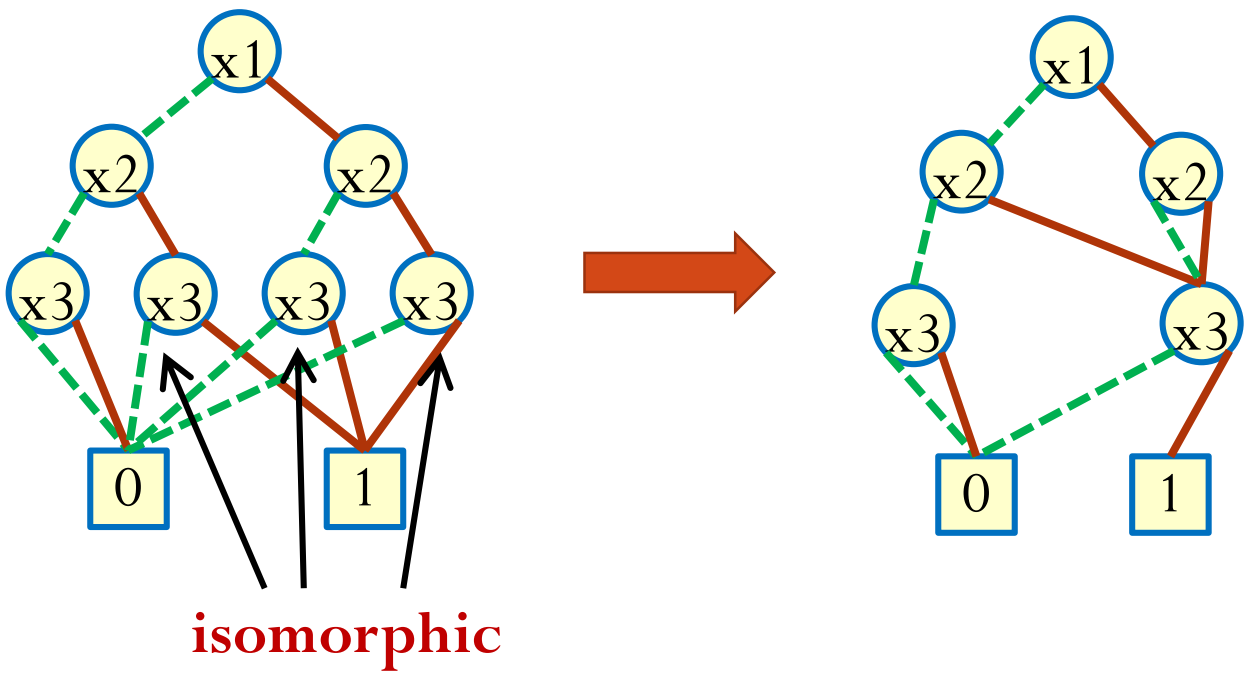
Steps of Merging Isomorphic Nodes

1. Remove **redundant** node.
2. Redirect all edges that **went into** the **redundant** node into the one copy that you kept.
   - For the example below, edges into right "x" node now are moved into left.



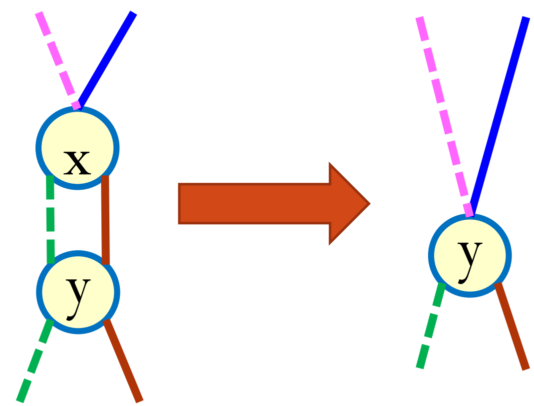
Reduction Rules

- Apply Rule 2, merging redundant nodes, to our example.



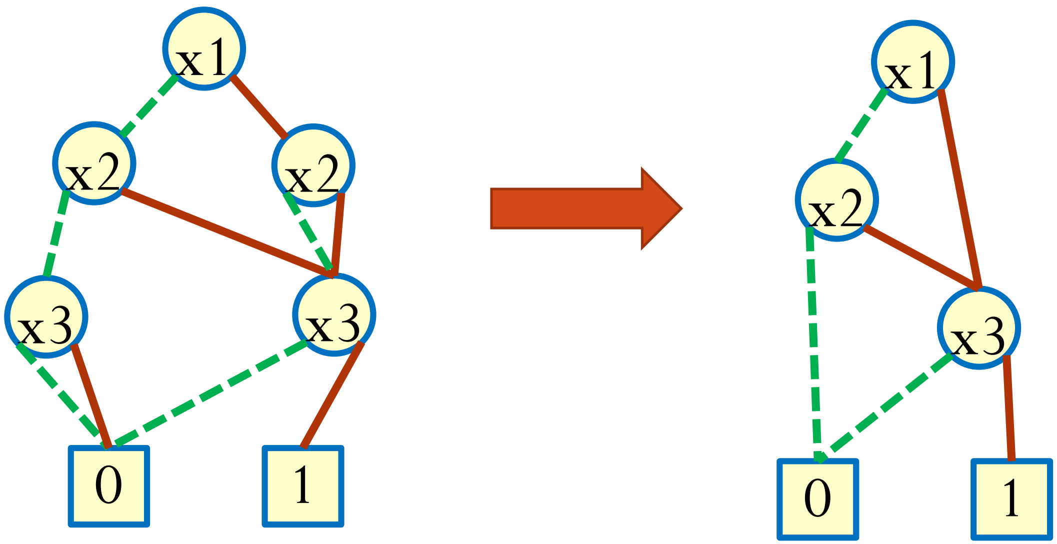
Reduction Rules

- **Reduction Rule 3**: **Eliminate Redundant Tests**
- **Redundant test**: both children of a node (x) go to the same node (y)
  - … so we don't care what value x node takes.
- Steps
  1. Remove redundant node.
  2. Redirect all edges into redundant node (x) into child (y) of the removed node.



Reduction Rules

- Apply Rule 3, eliminating redundant tests, to our example.



We are done!

Reduction Rules

- The above is a simple example.
  - The reduction process terminates by applying each rule ***once***.

- … But in real case, you may need to **iteratively** apply Rule 2 and 3.
  - It is only done when you cannot find any match of rule 2 or 3.

- Is this how programs **really** do it?
  - **No**!! We will talk about that later…

Binary Decision Diagrams (BDDs)
Big Results

- Recap: What did we do?
  - Start with a decision diagram in the form of a tree, order the variables, and reduce the diagram
  - Name: **Reduced Ordered BDD (ROBDD)**

- Big result: ROBDD is a canonical form data structure for any Boolean function.
  - **Same function** always generates exactly **same graph**… for **same variable ordering**.
  - Two functions **identical** if and only if ROBDD graphs are **isomorphic** (i.e., same).
- Nice property: **Simplest** form of graph is **canonical**.

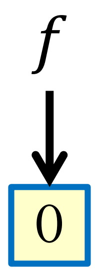
Outline

- Binary Decision Diagram (BDD): Introduction
- BDD Reduction
- BDDs for Common Functions
- Sharing in BDD
- Real Way to Build BDD
- Applications and Issues of BDD

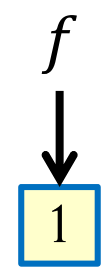
BDDs: Representing Simple Things

- NOTE: In an ROBDD, a Boolean function is really just **a pointer to the root node** of the graph.

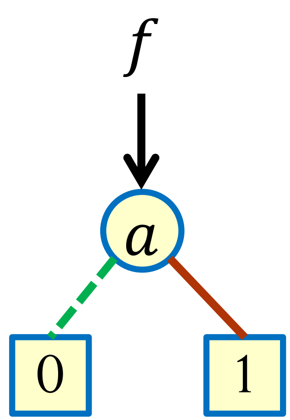
ROBDD for $f(a, b, \ldots, z) = 0$



ROBDD for $f(a, b, \ldots, z) = 1$

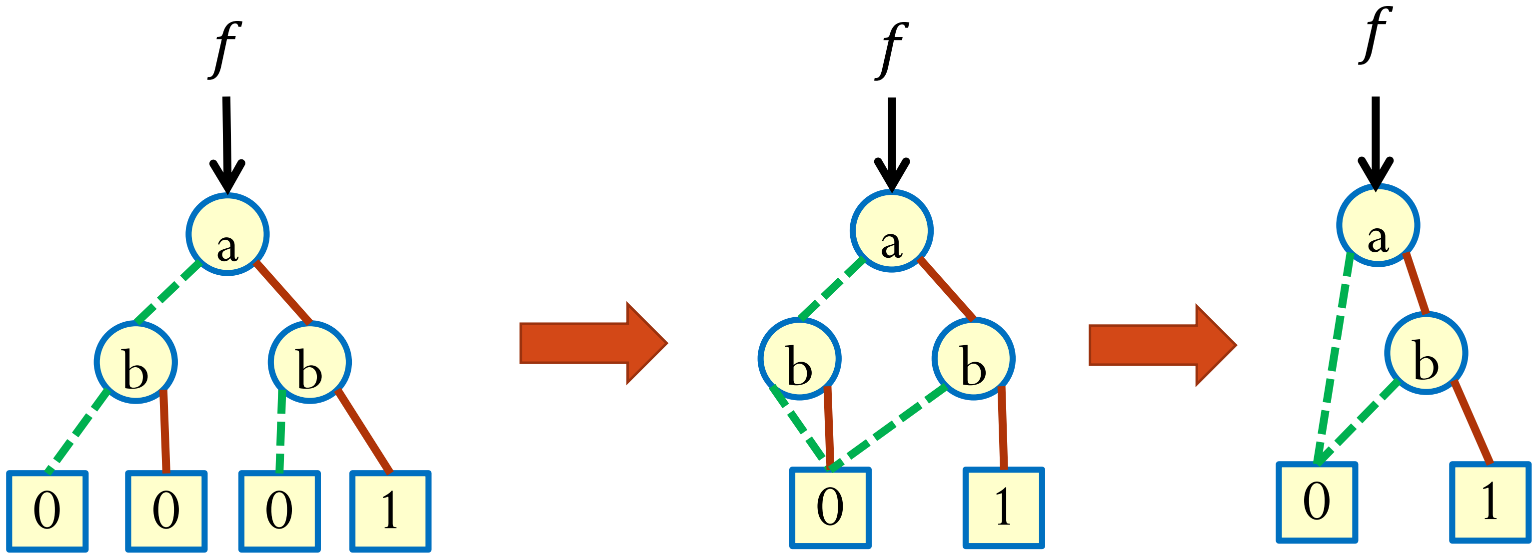


ROBDD for $f(a, b, \ldots, z) = a$



ROBDD for AND

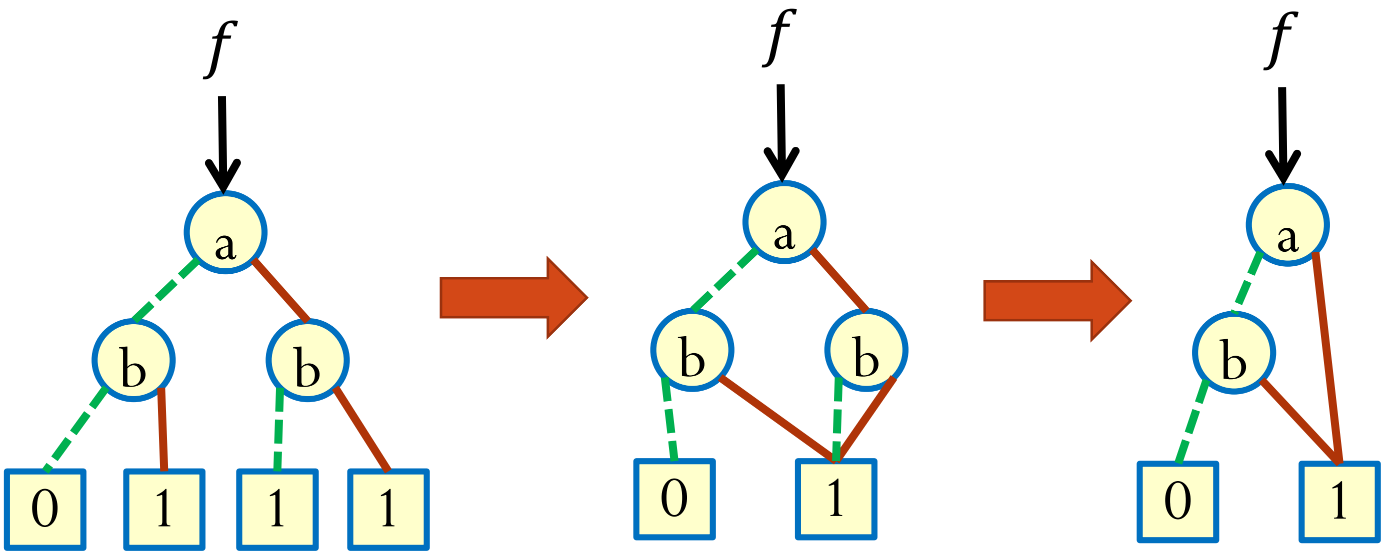
$f(a, b) = ab$



Same graph for $f(a, b, \dots, z) = ab$

ROBDD for OR

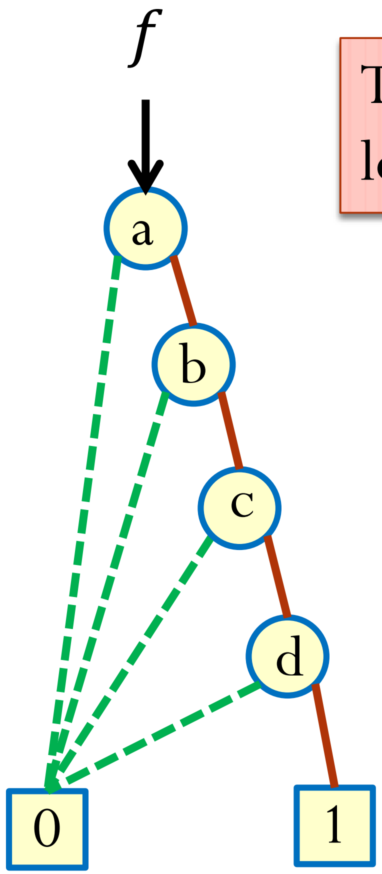
$f(a, b) = a + b$



Same graph for $f(a, b, \ldots, z) = a + b$

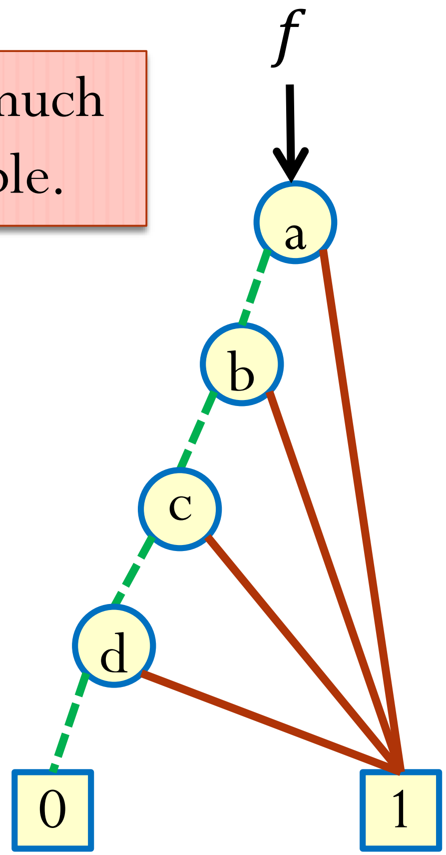
ROBDD for AND/OR on Multiple Inputs

$f(a,b,c,d) = abcd$



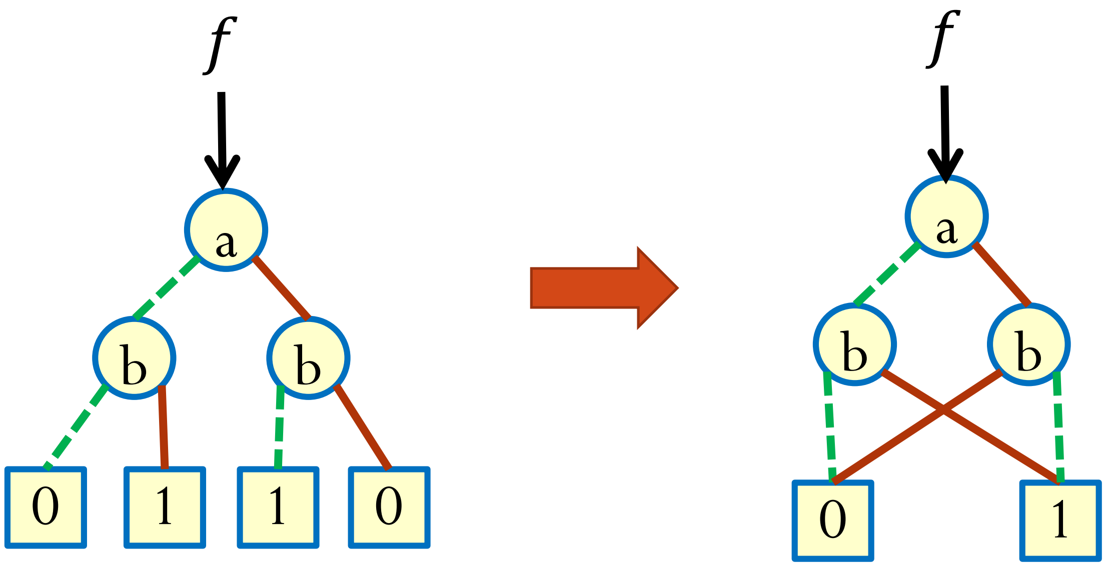
The complexity is much lower than truth table.

$f(a,b,c,d) = a + b + c + d$



ROBDD for XOR

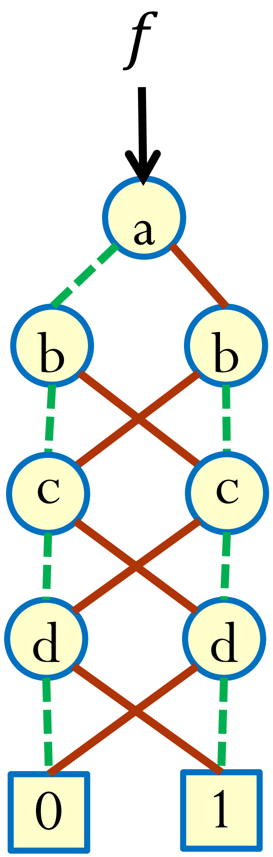
$f(a,b) = a \oplus b$



Same graph for $f(a, b, \dots, z) = a \oplus b$

ROBDD for XOR on Multiple Inputs

$$f(a, b, c, d) = a \oplus b \oplus c \oplus d$$



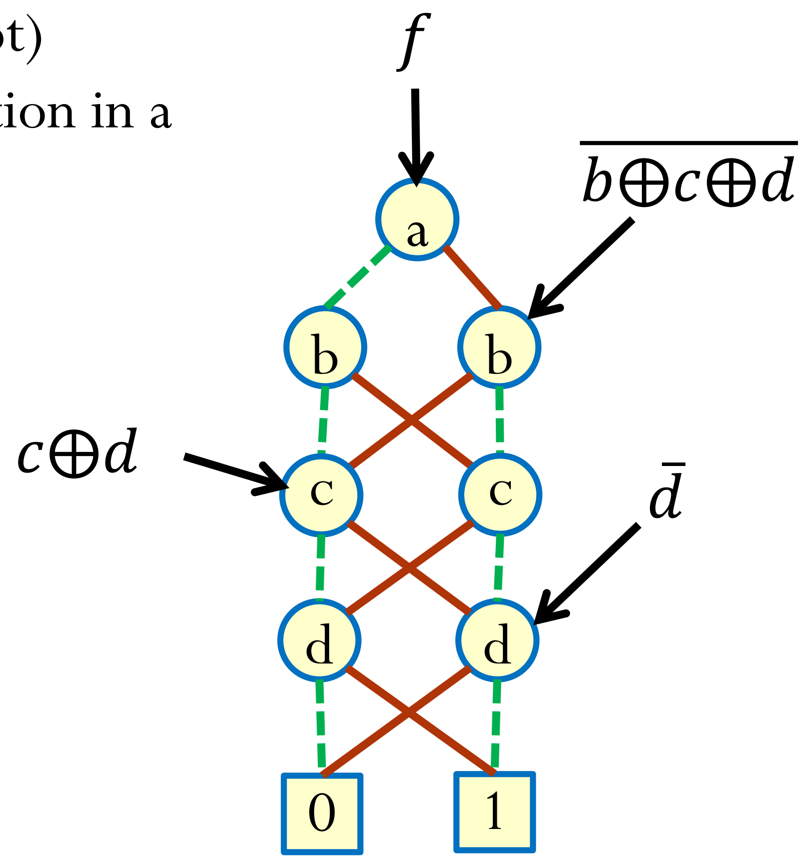
Outline

- Binary Decision Diagram (BDD): Introduction
- BDD Reduction
- BDDs for Common Functions
- Sharing in BDD
- Real Way to Build BDD
- Applications and Issues of BDD

Sharing in BDDs

- Very important technical point:
  - **Every** BDD node (not ***just*** root) represents **some** Boolean function in a **canonical** way.
  - BDD is good at extracting & representing **sharing of subfunctions** in subgraphs.

$f(a,b,c,d)$
$= a \oplus b \oplus c \oplus d$



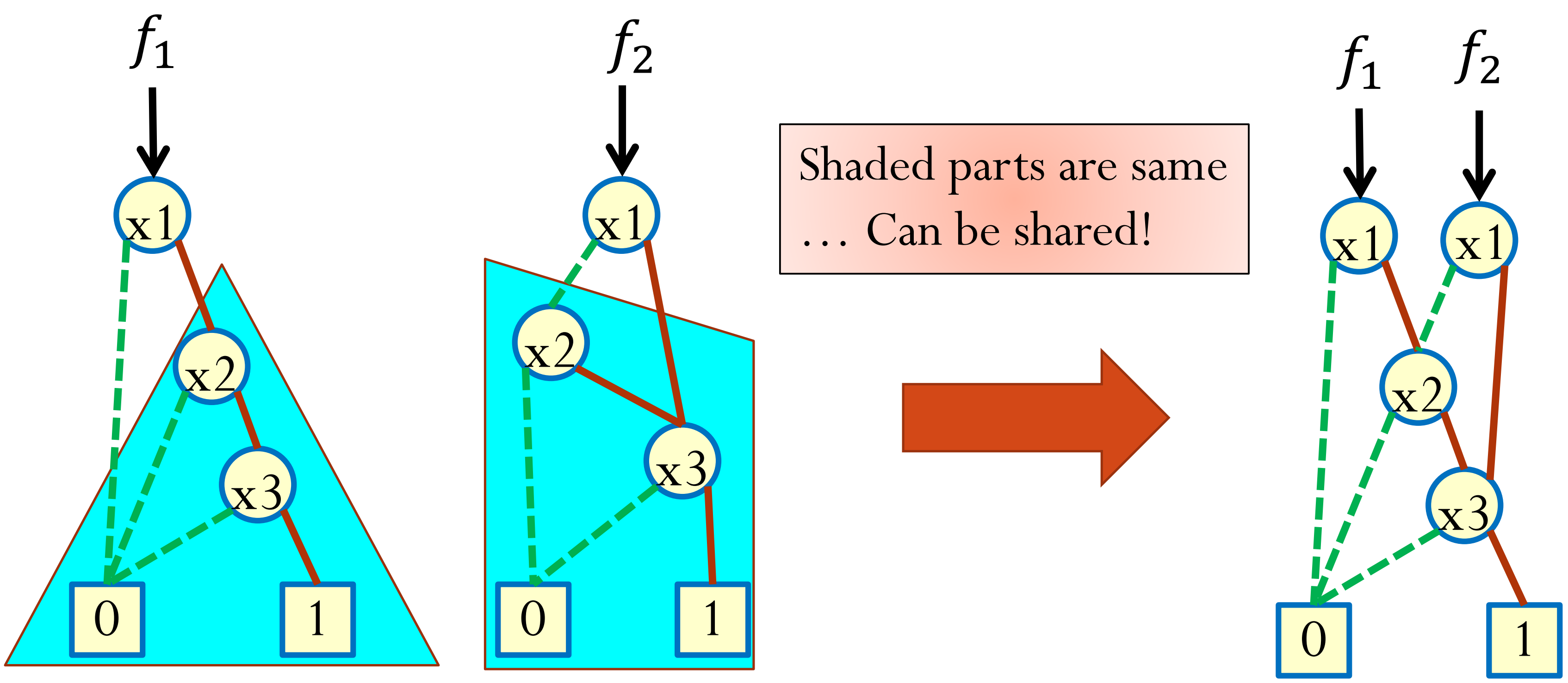
BDD Sharing: Multi-Rooted BDD

- If we are building BDDs for multiple functions,
  - …then there may be **same subgraphs** among different BDDs.
  - Don't represent same things multiple times; share them!
- As a result of sharing, the BDD can have multiple "entry points", or **roots**.
  - Called a **multi-rooted BDD**.

Multi-Rooted BDD: Example

- Build BDDs for two functions

$f_1(x_1, x_2, x_3) = x_1 x_2 x_3 \qquad f_2(x_1, x_2, x_3) = (x_1 + x_2) x_3$



Multi-Rooted BDD: Example

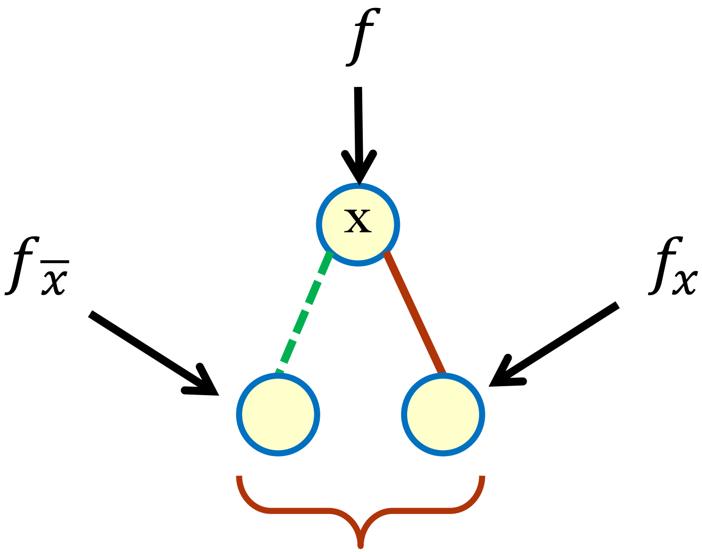
- Sharing among several separate BDDs reduces the size of BDD!

- Real example: Adders
  - Separately
    - **4-bit** adder: **51** nodes
    - **64-bit** adder: **12,481** nodes
  - Shared
    - **4-bit** adder: **31** nodes
    - **64-bit** adder: **571** nodes

Outline

- Binary Decision Diagram (BDD): Introduction
- BDD Reduction
- BDDs for Common Functions
- Sharing in BDD
- Real Way to Build BDD
- Applications and Issues of BDD

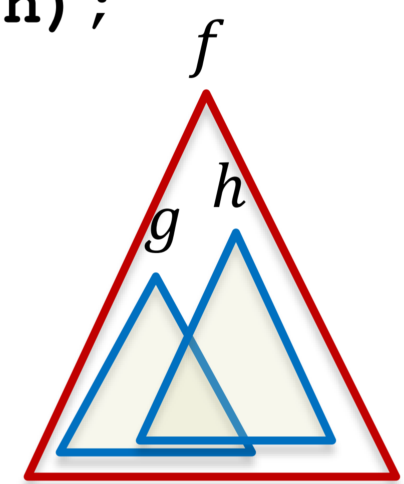
BDD and Cofactors



What are these two functions?

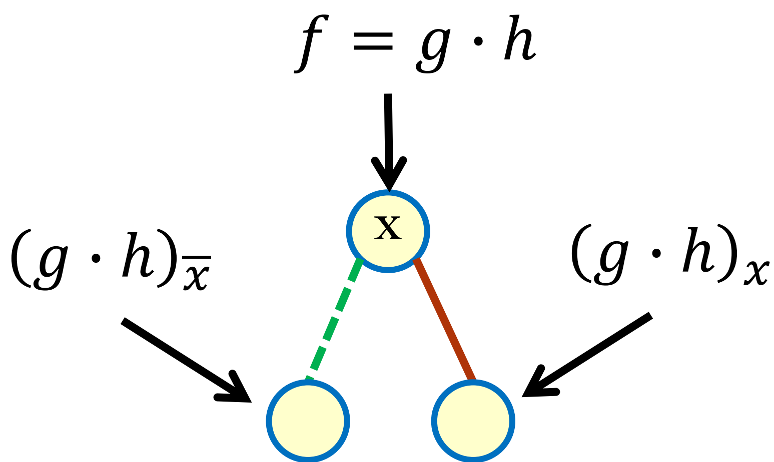
How Are BDDs Really Implemented?

- Recursively!
  - Cofactor and divide-and-conquer are two keys.
- Note: Boolean function can be decomposed: $f = op(g, h)$
  - op can be either AND, OR, XOR, NOT, …
- Idea: build ROBDD for g and ROBDD for h, then build ROBDD for f from the previous two ROBDDs.
  - op looks like: **`BDD op(BDD g, BDD h);`**
  - BDDs for g, h, and f can share.
  - Start from the base cases: ROBDDs for constants 0 and 1 and a single variable.



How to Implement OP?

- Example: op = AND
- BDD and cofactors:



- Therefore, we only need to obtain BDDs for $(g \cdot h)_{\overline{x}}$ and $(g \cdot h)_x$
  - Property of cofactors:
    - $(g \cdot h)_{\overline{x}} = g_{\overline{x}} \cdot h_{\overline{x}}$
    - $(g \cdot h)_x = g_x \cdot h_x$

Since we are given BDDs for g and h, it is easy to get BDDs for $g_{\overline{x}}$, g_x, $h_{\overline{x}}$, and h_x. We **recursively** apply op on $(g_{\overline{x}}, h_{\overline{x}})$ and (g_x, h_x) first.

Algorithm for Implementing OP

```
BDD op(BDD g, BDD h) {
    if (g is a leaf or h is a leaf) // termination condition:
                                    // either g = 0 or 1, or h = 0 or 1
        return proper BDD;
    var x = min(root(g), root(h)) // get the lowest order var
    BDD fLo = op( negCofBDD(g, x), negCofBDD(h, x) );
    BDD fHi = op( posCofBDD(g, x), posCofBDD(h, x) );
    return combineBDD(x, fLo, fHi);
}
```

Note:

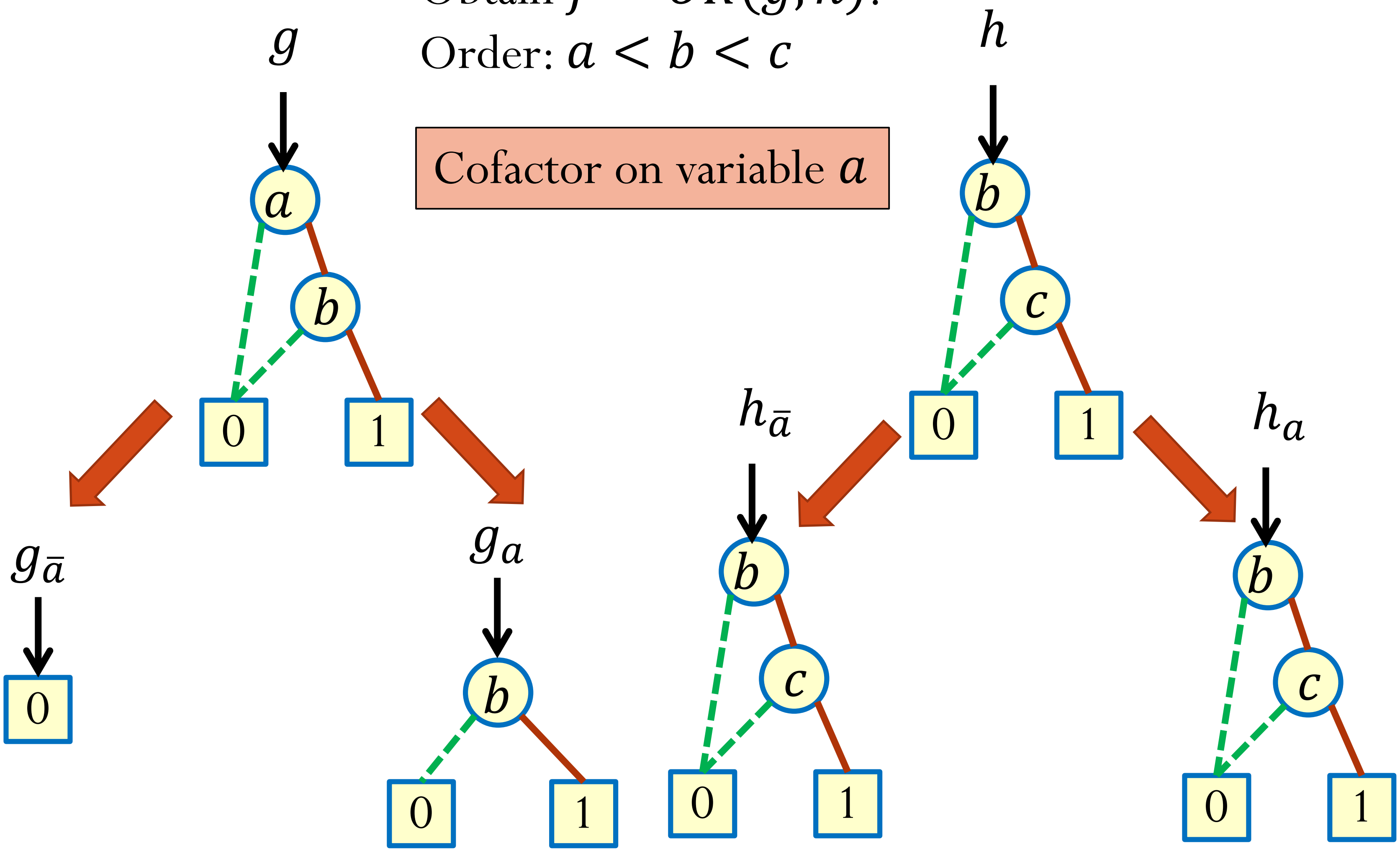
$$negCofBDD(g, x) = g_{\overline{x}} = \begin{cases} g & if\ x < root(g) \\ lo(g) & if\ x = root(g) \end{cases}$$

$$posCofBDD(g, x) = g_{x} = \begin{cases} g & if\ x < root(g) \\ hi(g) & if\ x = root(g) \end{cases}$$

Example of OP

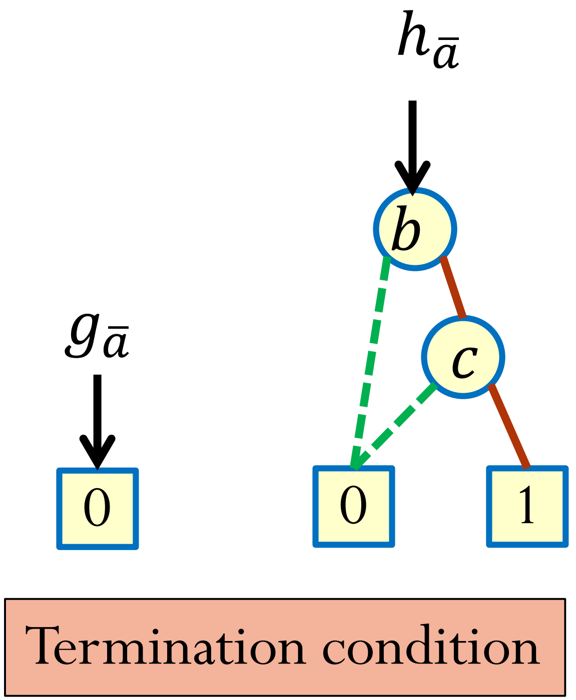
Obtain $f = OR(g, h)$.

Order: $a < b < c$

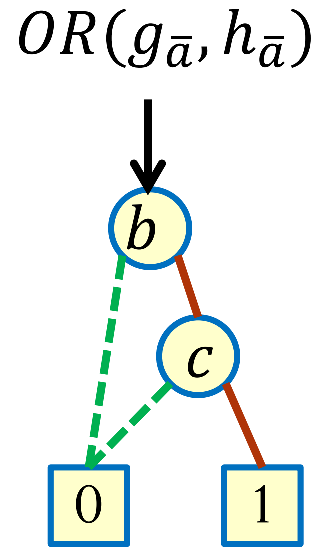


Example of OP (cont.)

- Recursively compute $OR(g_{\bar{a}}, h_{\bar{a}})$

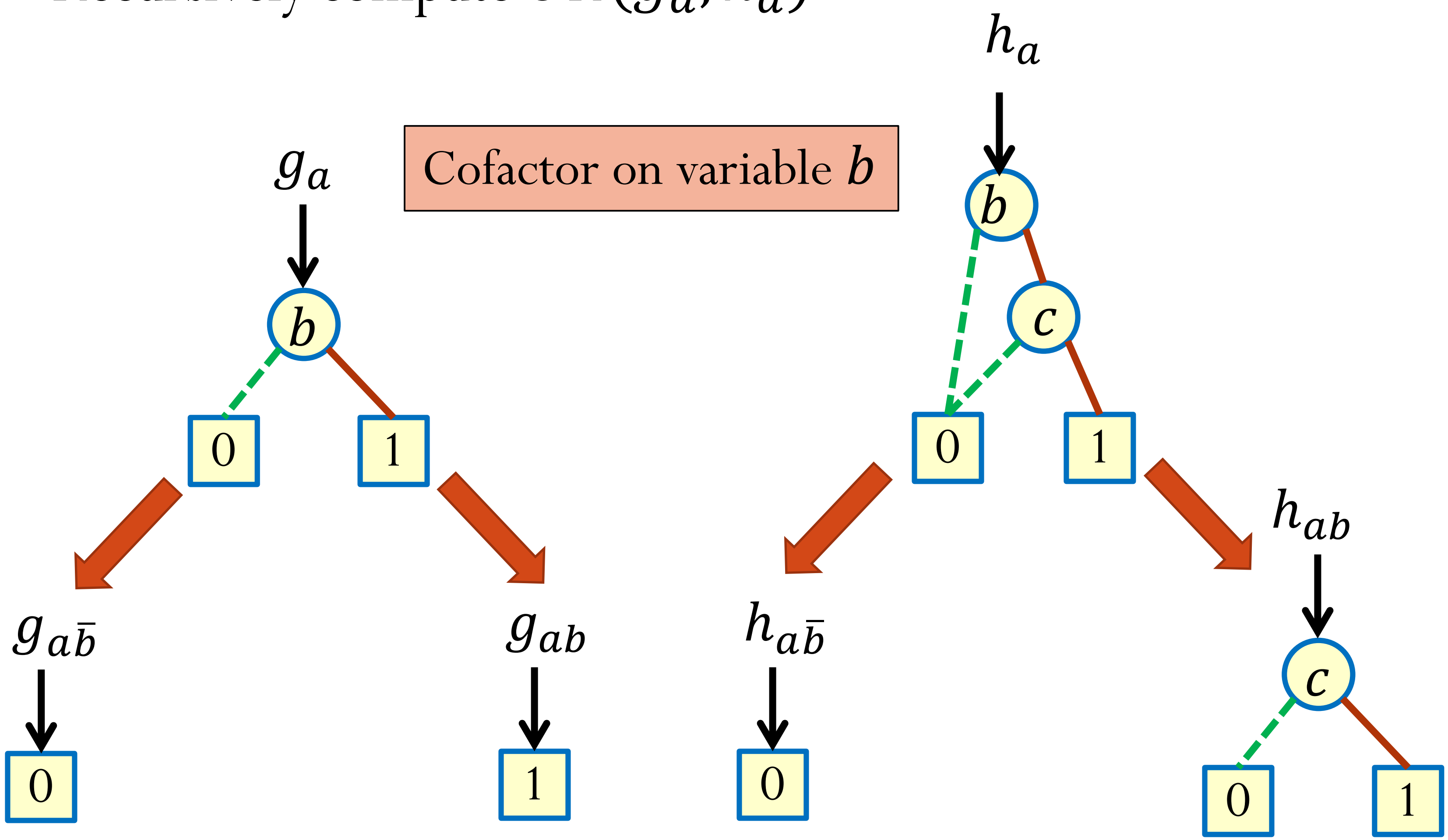


We obtain:



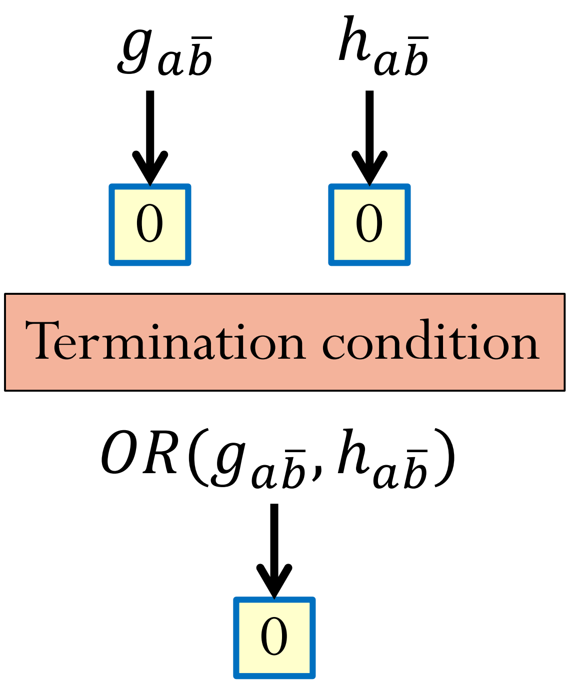
Example of OP (cont.)

- Recursively compute $OR(g_a, h_a)$

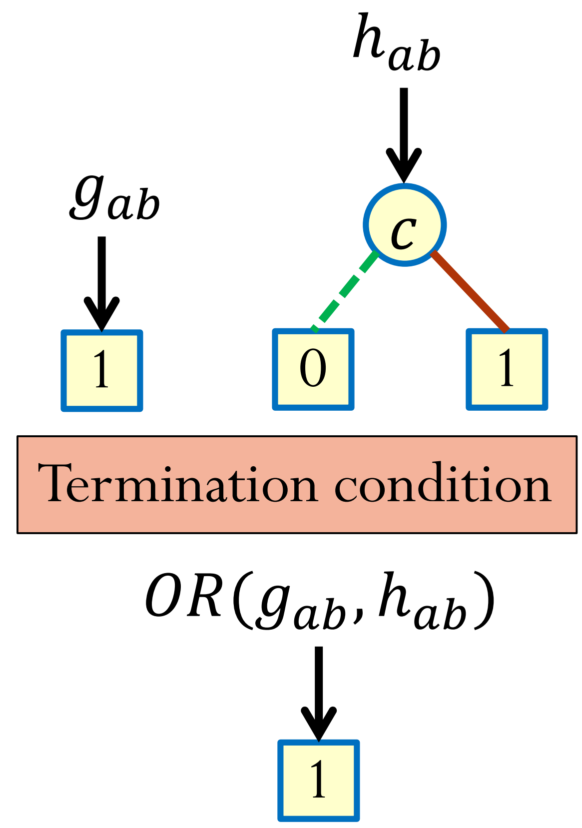


Example of OP (cont.)

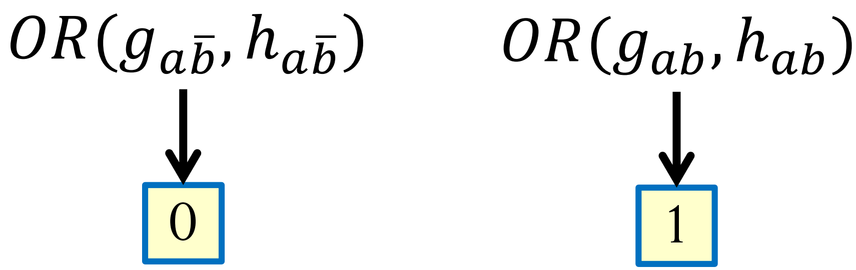
Recursively compute $OR(g_{a\bar{b}}, h_{a\bar{b}})$



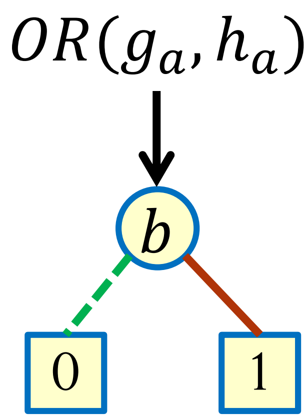
Recursively compute $OR(g_{ab}, h_{ab})$



Example of OP (cont.)

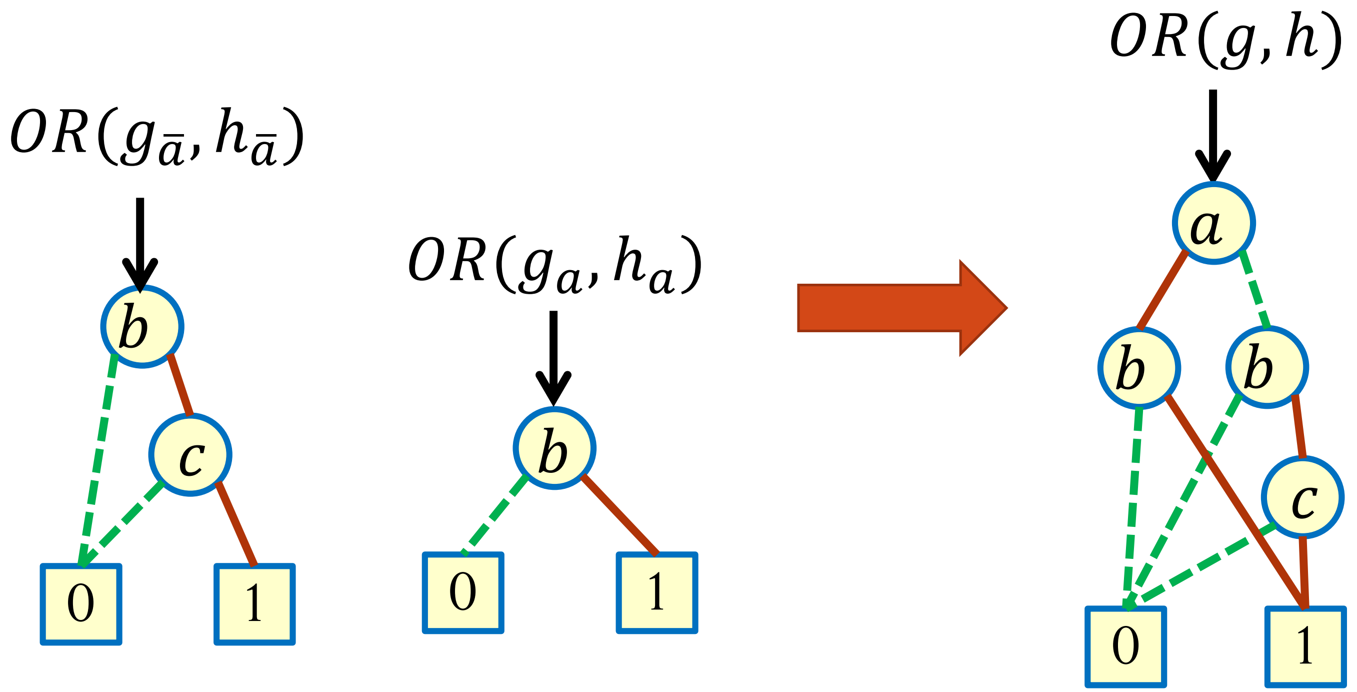


- Based on the recursion results, obtain $OR(g_a, h_a)$
  - **<u>Note</u>**: we cofactor on b.



Example of OP (cont.)

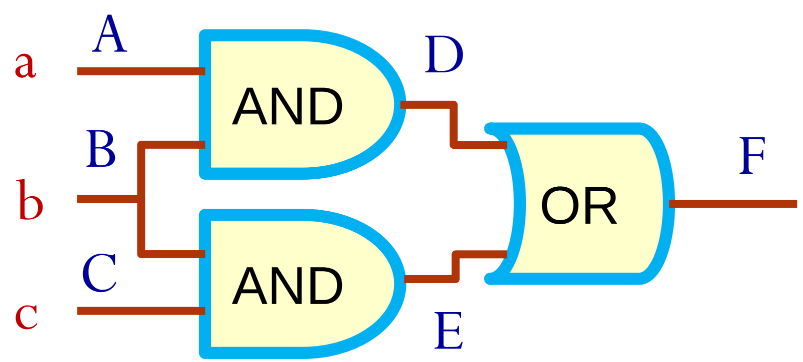
- Based on the recursion results, obtain $OR(g, h\ )$
  - **<u>Note</u>**: we cofactor on a.



Done!

BDDs: Build Up Incrementally…

- For a gate-level network, build the BDD for the output incrementally.

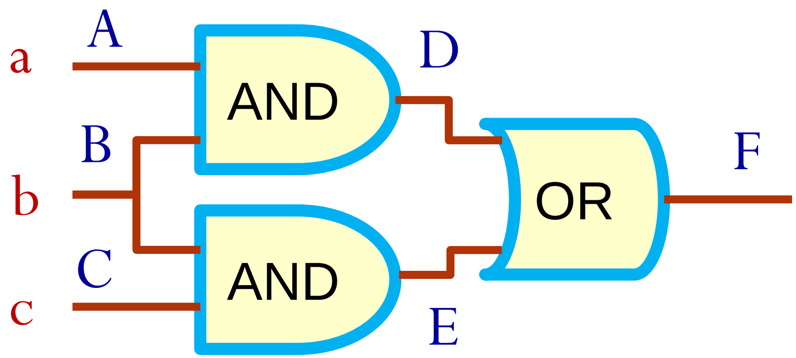


- Each **input** is a BDD. Each **gate** becomes an **operator** *op* that produces a new **output** BDD.
- Build BDD for F as a **script** of **calls** to basic BDD operators.
- Stick to a global ordering.

BDD operator script

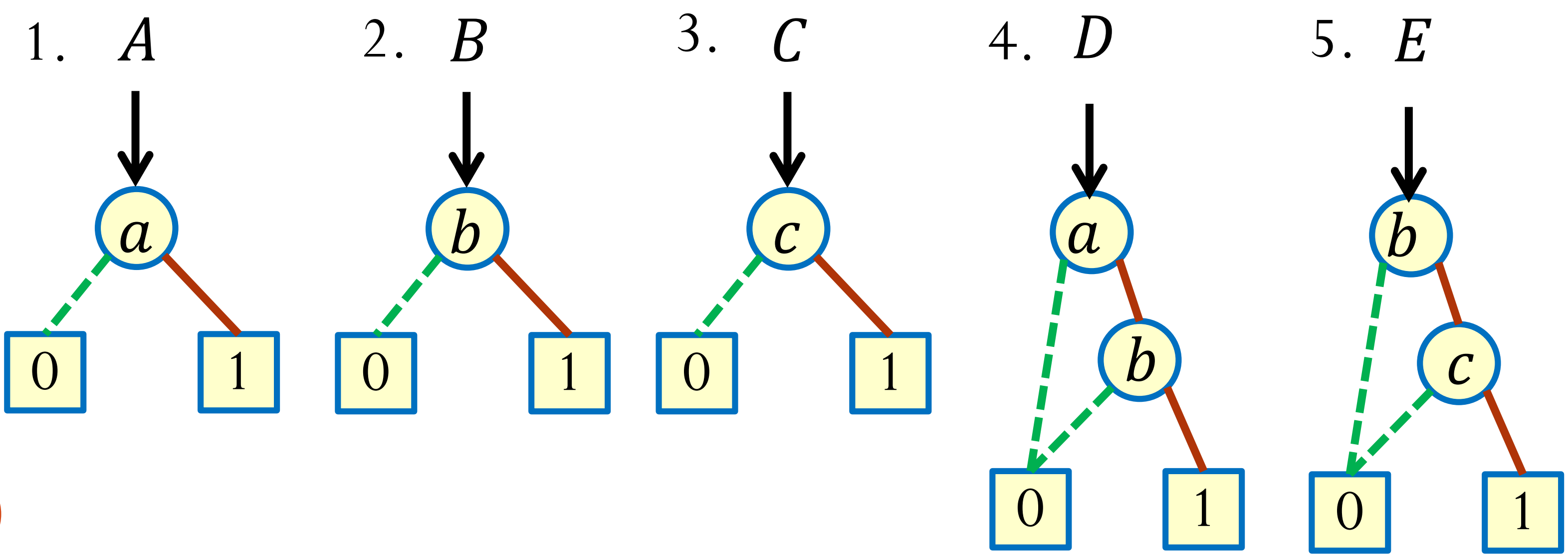
1. A = CreateVar("a")
2. B = CreateVar("b")
3. C = CreateVar("c")
4. D = AND(A, B)
5. E = AND(B, C)
6. F = OR(D, E)

Example: Build BDD Incrementally

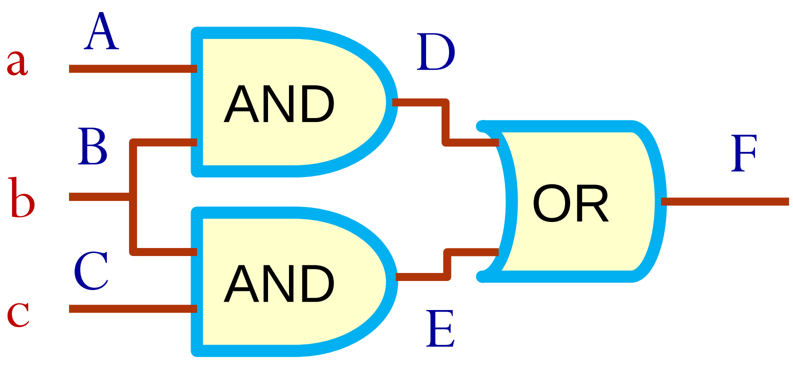


BDD operator script
1. A = CreateVar("A")
2. B = CreateVar("B")
3. C = CreateVar("C")
4. D = AND(A, B)
5. E = AND(B, C)
6. F = OR(D, E)

Global ordering: $a < b < c$



Example: Build BDD Incrementally

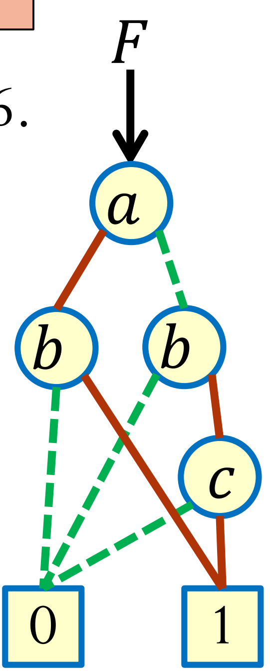


Global ordering: $a < b < c$

BDD operator script

1. A = CreateVar("A")
2. B = CreateVar("B")
3. C = CreateVar("C")
4. D = AND(A, B)
5. E = AND(B, C)
6. F = OR(D, E)

6.

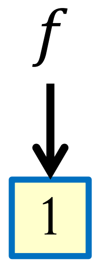


Outline

- Binary Decision Diagram (BDD): Introduction
- BDD Reduction
- BDDs for Common Functions
- Sharing in BDD
- Real Way to Build BDD
- Applications and Issues of BDD

Application of BDD
Tautology checking

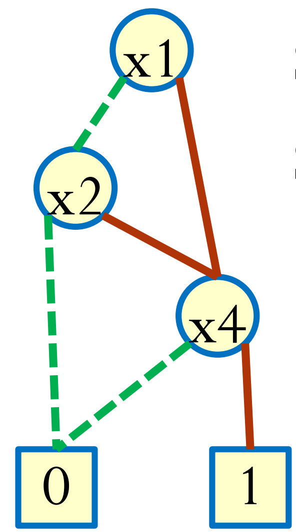
- <u>Solution</u>:
  - Build BDD for f.
  - Check if the BDD is just the BDD for $f = 1$.



Application of BDD
Satisfiability (SAT)

- Satisfiability (SAT): Does there exist an input pattern for variables that lets F = 1? If yes, return one pattern.
  - Recall: In network repair problem, we want to find (d_0, d_1, d_2, d_3) so that $(\forall ab\ z)(d_0, d_1, d_2, d_3) = 1$

- <u>Solution</u>:
  - If the BDD for F is not the BDD for $f = 0$. Then, SAT answer is yes.
  - Otherwise, no.
  - If yes, any path from root to "1" leaf is a solution.



SAT? Yes.

SAT pattern?

(x_1, x_2, x_3, x_4)
$= (0,1,*,1)$
$(1,*,*,1)$

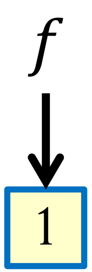
Application of BDD

Comparing Logic Implementations

- Are two given Boolean functions F and G the same?

- <u>Solution #1</u>:
  - Build BDD for F. Build BDD for G
  - Compare pointers to roots of F and G
  - If and only if pointers are **same**, F = G.

- <u>Solution #2</u>:
  - Build BDD for function $F \overline{\oplus} G$
  - Check if the BDD is just the BDD for $f = 1$.



Application of BDD
Comparing Logic Implementations

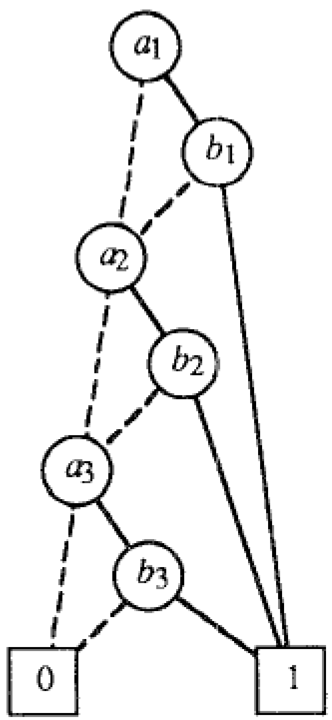
- What inputs make functions F and G give **different answers**?

- Solution:
  - Build BDD for $H = F \oplus G$.
  - Ask "**SAT**" question for H.

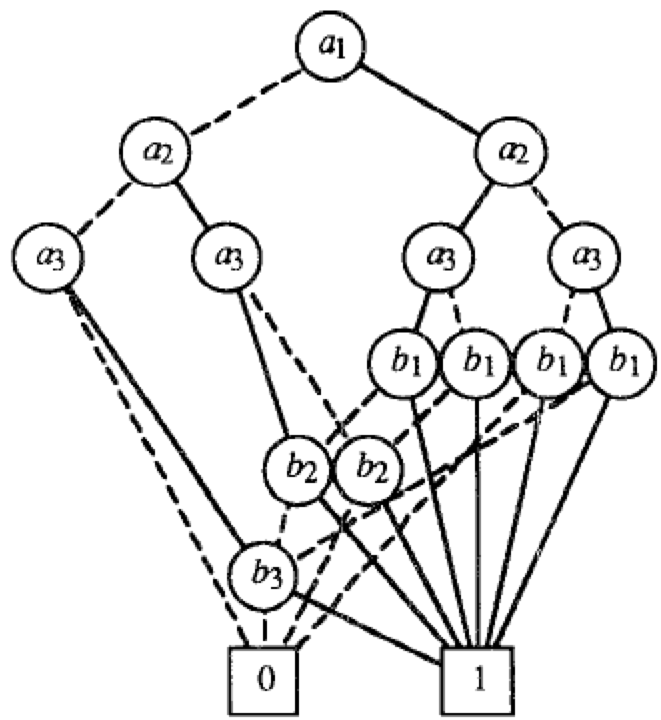
BDDs: Seem Too Good To Be True?!

- Problem : Variable ordering **matters**.
- Example: $a1 \cdot b1 + a2 \cdot b2 + a3 \cdot b3$

Good ordering: $a1 < b1 < a2 < b2 < a3 < b3$



Bad ordering: $a1 < a2 < a3 < b1 < b2 < b3$

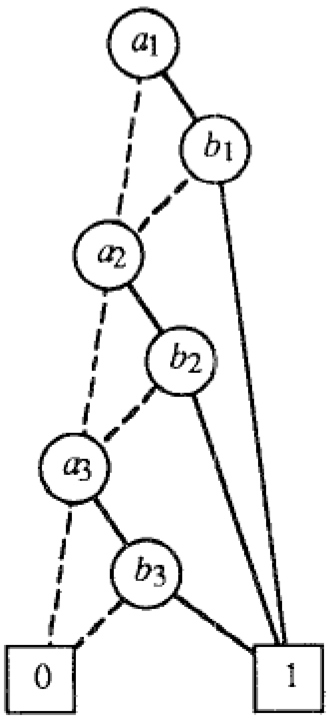


Variable Ordering: How to Handle?

- **Variable ordering heuristics**: how to order to get smaller BDDs?
- **Characterization**: which problems never make simple BDDs (e.g., multipliers)?
- **Dynamic ordering**: let the BDD program pick the order on the fly.

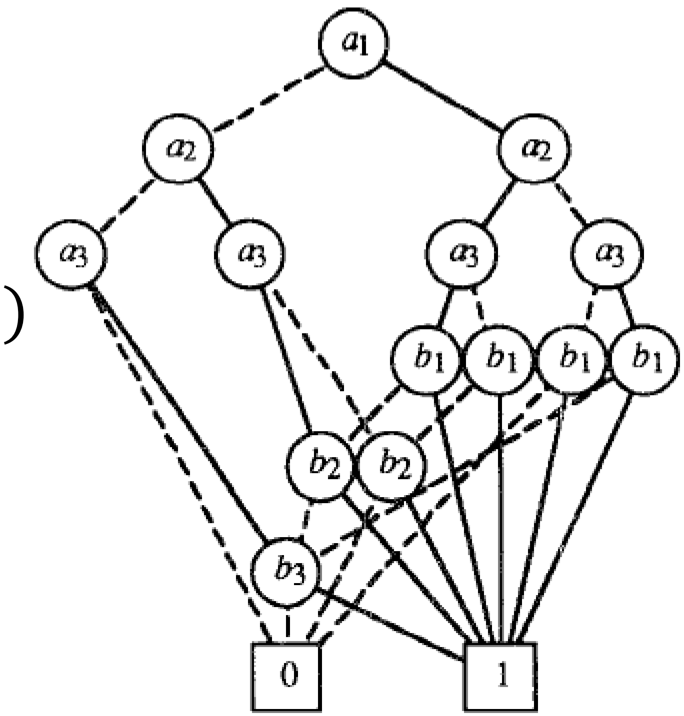
Variable Ordering: Intuition

- Rules of thumb for BDD ordering
  - **Groups** of inputs that can determine function by themselves should be (i) close together, and (ii) near top of BDD.
- Example: $a1 \cdot b1 + a2 \cdot b2 + a3 \cdot b3$
  - **Good ordering**:
    $a1 < b1 < a2 < b2 < a3 < b3$
  - Why?
    - a_i and b_i together can sometimes determine the function value



Variable Ordering: Intuition

- Rules of thumb for BDD ordering
  - **Groups** of inputs that can determine function by themselves should be (i) close together, and (ii) near top of BDD.
- Example: $a1 \cdot b1 + a2 \cdot b2 + a3 \cdot b3$
  - **Bad ordering**:
    $a1 < a2 < a3 < b1 < b2 < b3$
  - Why?
    - We need to remember $(a1, a2, a3)$ before we see any b's.



Variable Ordering: Practice

- Arithmetic circuits are important logic; how are their BDDs?
  - Many **carry chain circuits** have easy **linear sized** ROBDD orderings: Adders, Subtractors, Comparators.
  - Rule is to **alternate** variables in the BDD order: a0, b0, a1, b1, a2, b2, …, an, bn.
- Are all arithmetic circuits easy?
  - No! Multiplication is exponential in number of nodes for **any** order.
- General experience with BDDs
  - Many tasks have reasonable ROBDD sizes; algorithms are practical to about 100M nodes.

BDD Summary

- Reduced, Ordered, Binary Decision Diagrams, ROBDDs
  - Canonical form – a data structure – for Boolean functions.
  - Two Boolean functions are the same if and only if they have identical BDD.
  - A Boolean function is just a pointer to the root node of the BDD graph.
  - Every node in a (shared) BDD represents some function.
  - Basis for much of today's general manipulation of Boolean stuff.

- Problems
  - Variable ordering matters; sometimes BDD is just too big.
  - Often, we just want to know **SAT** – don't need to build the whole function.