《芯片设计自动化与智能优化》
SAT & BDD

The slides are based on Prof. Weikang Qian’s lecture notes at SJTU and Prof. Rob Rutenbar’s lecture notes at UIUC

Yibo Lin

Peking University

Outline

- Satisfiability (SAT)
- Binary decision diagram (BDD)
  - Optional

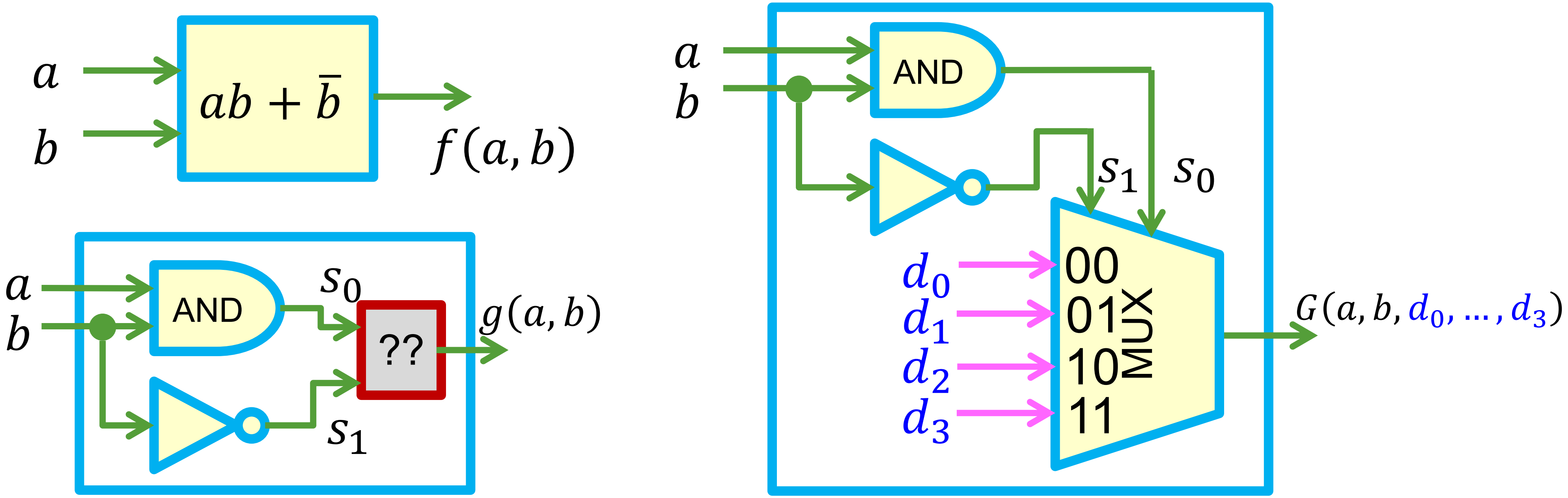
Satisfiability

- Called **SAT** for short
  - Given an appropriate representation of function $F(x_1, x_2, \dots, x_n)$, find an assignment of the variables $(a_1, a_2, \dots, a_n)$ so that $F(a_1, a_2, \dots, a_n) = 1$.
  - Note: could have many satisfying solution; **any one** is fine.
  - However, if there are no satisfying assignments at all, prove it and return this info.
    - We call this **unSAT**.
- Something you can do with BDDs, can do **easier** with SAT.
  - SAT is aimed at scenarios where you just need **one satisfying assignment**…
  - … or prove that there is **no** such satisfying assignment.

Example: Network Repair

Specified

Implemented



- We want to find (d_0, d_1, d_2, d_3) so that $(\forall ab\ z)(d_0, d_1, d_2, d_3) = 1$
- To repair the network, we only need **one satisfying assignment** for (d_0, d_1, d_2, d_3).
- If **unSAT**, the network repair is impossible!

Standard SAT Form: CNF

- **Conjunctive Normal Form (CNF)**
  – It is just standard **Product-of-Sums** form.

$$\Phi = \underbrace{(a + c)}_{\text{clause}}(b + \underbrace{c}_{\text{positive literal}})(\bar{a} + \bar{b} + \underbrace{\bar{c}}_{\text{negative literal}})$$

- Terminology
  – Each sum is called a **clause**.
  – Each variable in true form is called a **positive literal**.
  – Each variable in complement form is called a **negative literal**.
- Why CNF is useful?
  – Need only determine that **one** clause evaluates to “0” to know whole formula = “0”.
  – Of course, to satisfy the whole formula, you must make **all** clauses identically “1”.

Assignment to a CNF Formula

- An **assignment** gives values to **some**, **not necessarily all**, of variables x_i in $(x_1, x_2, \dots, x_n)$.
  - **Complete** assignment: assigns value to all variables.
  - **Partial** assignment: some, not all, variables have values.
- Given an assignment, we can evaluate **status** of the clauses.
- There are three status:
  - **Conflicting**: Clause = 0
  - **Satisfied**: Clause=1
  - **Unsolved**: Clause unknown
- Example: $a = 0$, $b = 1$, but c and d unassigned.

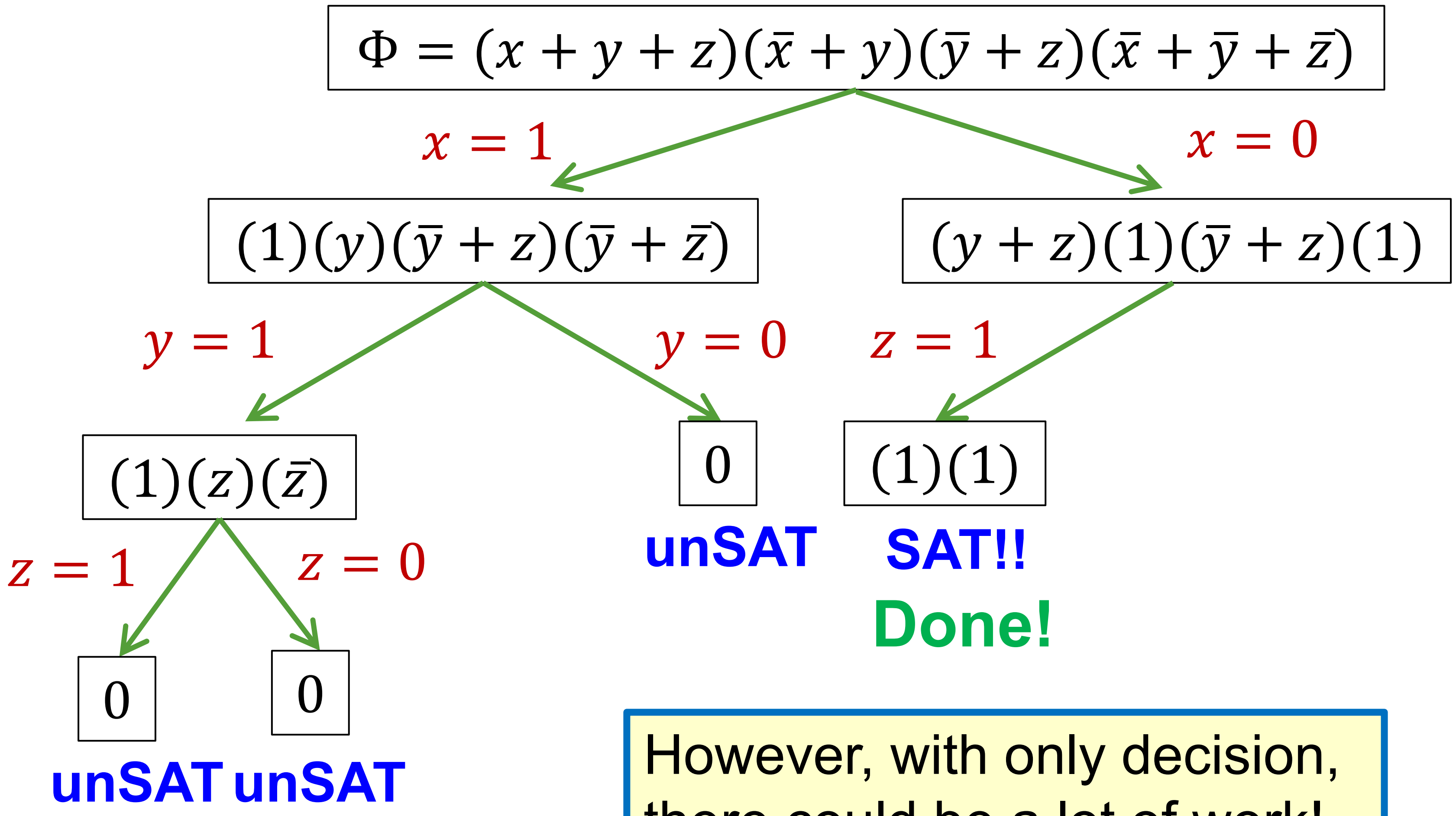
$$\Phi = \underbrace{(a + \bar{b})}_{\text{Conflicting}}\underbrace{(\bar{a} + b + \bar{c})}_{\text{Satisfied}}\underbrace{(a + c + d)}_{\text{Unsolved}}\underbrace{(\bar{a} + \bar{b} + \bar{c})}_{\text{Satisfied}}$$

How to Solve SAT Problem?

- **Recursively!**

- Idea #1: **Decision**
  - Select a variable and **assign** its value; **simplify** CNF formula as far as you can.
  - Hope you can decide if it is SAT or unSAT, without any further work.
  - If you cannot, pick another variable.

Decision: Example



However, with only decision, there could be a lot of work!

How to Solve SAT Problem?

- Idea #2: **Deduction**
  - Look at the newly simplified clauses.
  - Based on **structure of clauses**, and **values of partial assignment**, we can **deduct** the values of some unassigned variables so that SAT is **possible**.
  - With new values deducted, simplify the CNF as far as you can.
  - Do deduction and simplification **iteratively** until nothing simplifies. If you can decide SAT or unSAT, great.
  - If you cannot, then you have to recurse some more, back up to Decision.

Deduction: Example

$$\boxed{\Phi = (x + y + z)(\bar{x} + y)(\bar{y} + z)(\bar{x} + \bar{y} + \bar{z})}$$

$$\Big\downarrow {\scriptstyle x = 1}$$

$$\boxed{(1)(y)(\bar{y} + z)(\bar{y} + \bar{z})}$$ Deduction: $y = 1$

$$\Big\downarrow \textbf{Simplify}$$

$$\boxed{(1)(z)(\bar{z})}$$ Deduction: $z = 1$

$$\Big\downarrow \textbf{Simplify}$$

$$\boxed{(1)(0)}$$

unSAT

BCP: Boolean Constraint Propagation

- To do "**deduction**", use **Boolean Constraint Propagation (BCP).**
  - Given a set of **fixed** variable assignments, you "**deduce**" about other necessary assignments by "**propagating constraints**".
    - What constraints? Each clause should be **satisfied**.
- Most famous BCP strategy is "**Unit Clause Rule**"
  - A clause is said to be "**unit**" if it has **exactly one** **unassigned literal**.
  - Unit clause has **exactly one** way to be satisfied, i.e., pick polarity that makes clause="1".
  - This choice is called an "**implication**".

Example: Unit Clause Rule

$$\Phi = (a + c)(b + c)(\bar{a} + \bar{b} + \bar{c})$$

- Assume $a = 1, b = 1$
- We can deduct that $c = 0$.

BCP Example

Partial Assignment is $x_9 = 0, x_{10} = 0,$
$x_{11} = 0, x_{12} = 1, x_{13} = 1$

$$\Phi = \omega_1 \omega_2 \cdots \omega_9$$

$$\omega_1 = \bar{x}_1 + x_2$$
$$\omega_2 = \bar{x}_1 + x_3 + x_9$$
$$\omega_3 = \bar{x}_2 + \bar{x}_3 + x_4$$
$$\omega_4 = \bar{x}_4 + x_5 + x_{10}$$
$$\omega_5 = \bar{x}_4 + x_6 + x_{11}$$
$$\omega_6 = \bar{x}_5 + \bar{x}_6$$
$$\omega_7 = x_1 + x_7 + \bar{x}_{12}$$
$$\omega_8 = x_1 + x_8$$
$$\omega_9 = \bar{x}_7 + \bar{x}_8 + \bar{x}_{13}$$

Simplify

$$\omega_1 = \bar{x}_1 + x_2$$
$$\omega_2 = \bar{x}_1 + x_3$$
$$\omega_3 = \bar{x}_2 + \bar{x}_3 + x_4$$
$$\omega_4 = \bar{x}_4 + x_5$$
$$\omega_5 = \bar{x}_4 + x_6$$
$$\omega_6 = \bar{x}_5 + \bar{x}_6$$
$$\omega_7 = x_1 + x_7$$
$$\omega_8 = x_1 + x_8$$
$$\omega_9 = \bar{x}_7 + \bar{x}_8$$

No SAT
No BCP
Now what?

BCP Example (cont.)

- Next: Assign a variable to a value
  - Assign $x_1 = 1$

$\omega_1 = \bar{x}_1 + x_2$

$\omega_2 = \bar{x}_1 + x_3$

$\omega_3 = \bar{x}_2 + \bar{x}_3 + x_4$

$\omega_4 = \bar{x}_4 + x_5$

$\omega_5 = \bar{x}_4 + x_6$

$\omega_6 = \bar{x}_5 + \bar{x}_6$

$\omega_7 = x_1 + x_7$

$\omega_8 = x_1 + x_8$

$\omega_9 = \bar{x}_7 + \bar{x}_8$

Simplify

$$\left.\begin{aligned}\omega_1 &= x_2\\ \omega_2 &= x_3\end{aligned}\right\} \xrightarrow{\textbf{Implication}} \begin{aligned}x_2 &= 1\\ x_3 &= 1\end{aligned}$$

$\omega_3 = \bar{x}_2 + \bar{x}_3 + x_4$

$\omega_4 = \bar{x}_4 + x_5$

$\omega_5 = \bar{x}_4 + x_6$

$\omega_6 = \bar{x}_5 + \bar{x}_6$

$\omega_7 = 1$

$\omega_8 = 1$

$\omega_9 = \bar{x}_7 + \bar{x}_8$

BCP Example (cont.)

- Assign implied values
  - Assign $x_2 = 1, x_3 = 1$

$\omega_1 = x_2$

$\omega_2 = x_3$

$\omega_3 = \bar{x}_2 + \bar{x}_3 + x_4$

$\omega_4 = \bar{x}_4 + x_5$

$\omega_5 = \bar{x}_4 + x_6$

$\omega_6 = \bar{x}_5 + \bar{x}_6$

$\omega_7 = 1$

$\omega_8 = 1$

$\omega_9 = \bar{x}_7 + \bar{x}_8$

Simplify

$\omega_1 = 1$

$\omega_2 = 1$

$\omega_3 = x_4$ $\xrightarrow{\text{Implication}}$ $x_4 = 1$

$\omega_4 = \bar{x}_4 + x_5$

$\omega_5 = \bar{x}_4 + x_6$

$\omega_6 = \bar{x}_5 + \bar{x}_6$

$\omega_7 = 1$

$\omega_8 = 1$

$\omega_9 = \bar{x}_7 + \bar{x}_8$

BCP Example (cont.)

- Assign implied values
  - Assign $x_4 = 1$

$$\omega_1 = 1$$
$$\omega_2 = 1$$
$$\omega_3 = x_4$$
$$\omega_4 = \bar{x}_4 + x_5$$
$$\omega_5 = \bar{x}_4 + x_6$$
$$\omega_6 = \bar{x}_5 + \bar{x}_6$$
$$\omega_7 = 1$$
$$\omega_8 = 1$$
$$\omega_9 = \bar{x}_7 + \bar{x}_8$$

Simplify $\Rightarrow$

$$\omega_1 = 1$$
$$\omega_2 = 1$$
$$\omega_3 = 1$$
$$\left.\begin{aligned}\omega_4 &= x_5\\ \omega_5 &= x_6\end{aligned}\right\} \xrightarrow{\text{Implication}} \begin{aligned}x_5 &= 1\\ x_6 &= 1\end{aligned}$$
$$\omega_6 = \bar{x}_5 + \bar{x}_6$$
$$\omega_7 = 1$$
$$\omega_8 = 1$$
$$\omega_9 = \bar{x}_7 + \bar{x}_8$$

BCP Example (cont.)

- Assign implied values
  - Assign $x_5 = 1, x_6 = 1$

$$\omega_1 = 1 \qquad\qquad \omega_1 = 1$$
$$\omega_2 = 1 \qquad\qquad \omega_2 = 1$$
$$\omega_3 = 1 \qquad\qquad \omega_3 = 1$$
$$\omega_4 = x_5 \qquad\qquad \omega_4 = 1$$
$$\omega_5 = x_6 \qquad\qquad \omega_5 = 1$$
$$\omega_6 = \bar{x}_5 + \bar{x}_6 \qquad\qquad \omega_6 = 0$$
$$\omega_7 = 1 \qquad\qquad \omega_7 = 1$$
$$\omega_8 = 1 \qquad\qquad \omega_8 = 1$$
$$\omega_9 = \bar{x}_7 + \bar{x}_8 \qquad\qquad \omega_9 = \bar{x}_7 + \bar{x}_8$$

Simplify

Conflicting!

unSAT

BCP Example: Summary

- We start from partial assignment:
  $x_9 = 0, x_{10} = 0, x_{11} = 0,$
  $x_{12} = 1, x_{13} = 1$
- Next we assign $x_1 = 1$.
- After that, by BCP, we get implications:
  $x_2 = 1, x_3 = 1$
  $x_4 = 1$
  $x_5 = 1, x_6 = 1$
- Finally, we obtain a conflicting clause → unSAT

$$\Phi = \omega_1 \omega_2 \cdots \omega_9$$
$$\omega_1 = \bar{x}_1 + x_2$$
$$\omega_2 = \bar{x}_1 + x_3 + x_9$$
$$\omega_3 = \bar{x}_2 + \bar{x}_3 + x_4$$
$$\omega_4 = \bar{x}_4 + x_5 + x_{10}$$
$$\omega_5 = \bar{x}_4 + x_6 + x_{11}$$
$$\omega_6 = \bar{x}_5 + \bar{x}_6$$
$$\omega_7 = x_1 + x_7 + \bar{x}_{12}$$
$$\omega_8 = x_1 + x_8$$
$$\omega_9 = \bar{x}_7 + \bar{x}_8 + \bar{x}_{13}$$

When Does BCP Finish?

- Three cases when BCP finishes:
  - **SAT**: Find a SAT assignment, all clauses resolve to “1”. Return the assignment.
  - **Unresolved**: One or more clauses unresolved.
    - What’s next? Pick another unassigned variable, and recurse more.
  - **unSAT**: Find conflict, one or more clauses evaluate to “0”.
    - What’s next? You need to **undo** one of the previous variable assignments, try again…

DPLL Algorithm

- What we have covered is the basic idea behind the famous SAT-solving algorithm -- **Davis-Putnam-Logemann-Loveland (DPLL) Algorithm**.
  - Davis and Putnam published the basic recursive framework in 1960.
  - Davis, Logemann, and Loveland found smarter BCP, e.g., unit-clause rule, in 1962.
- Big ideas
  - A complete, systematic search of variable assignments.
  - Use CNF form for efficiency.
  - BCP makes search stop earlier, "**resolving**" more assignments without recursing more.

SAT: Huge Progress Last ~20 Years

- DPLL is only the start…
- SAT has been subject of intense work and **great progress**.
  - Efficient data structures for clauses (so can search them fast).
  - Efficient variable selection heuristics (so search smart, find lots of implications).
  - Efficient BCP mechanisms (because SAT spends MOST of its time here).
  - Learning mechanisms (find patterns of variables that NEVER lead to SAT, avoid them).
- Results: Good SAT codes that can do huge problems, fast.
  - 50,000 variables; 50,000,000 clauses

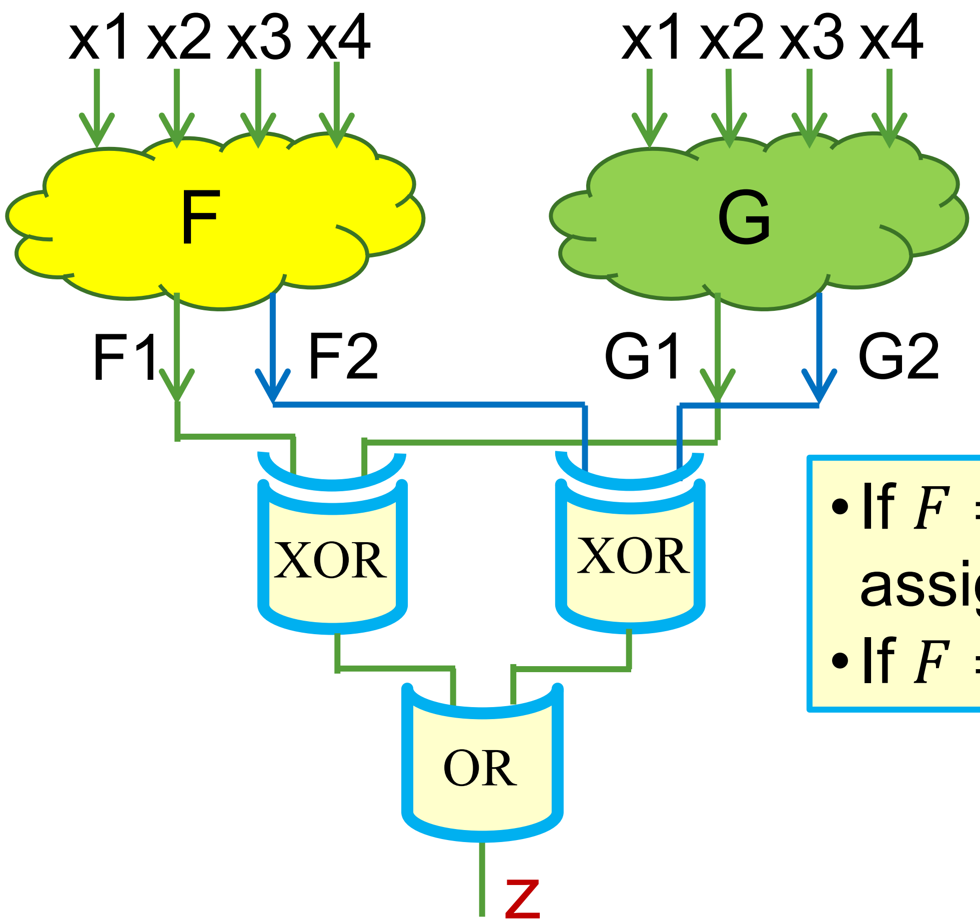
SAT Solvers

- Many good solvers available online, open source.

- Examples
  - **MiniSAT**, from Niklas Eén, Niklas Sörensson in Sweden.
  - **CHAFF**, from Sharad Malik and students, Princeton University.
  - **GRASP**, from Joao Marques-Silva and Karem Sakallah, University of Michigan.
  - **Z3 theorem prover**, from Microsoft Research
  - …and many others too.

Application of SAT in EDA

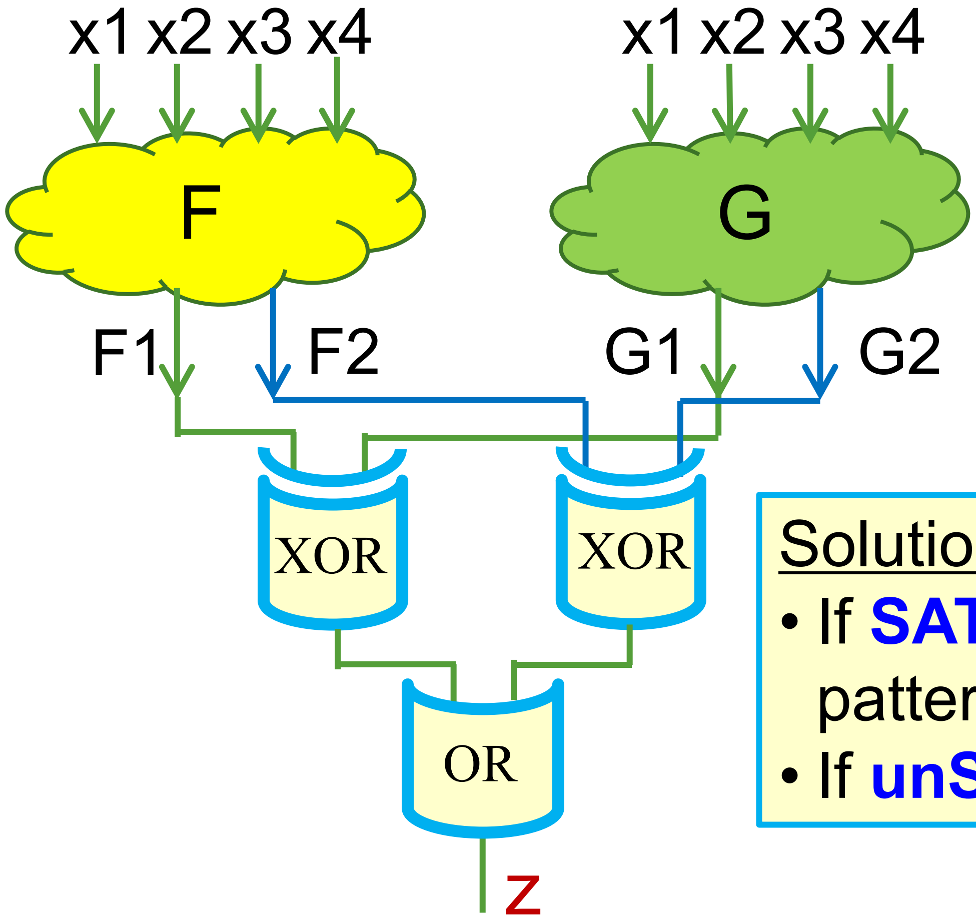
- Do these two logic networks implement the **same** Boolean function?



- If $F \neq G$ → some input assignment lets $z = 1$: **SAT!**
- If $F = G$ → $z \equiv 0$: **unSAT**!

Application of SAT in EDA

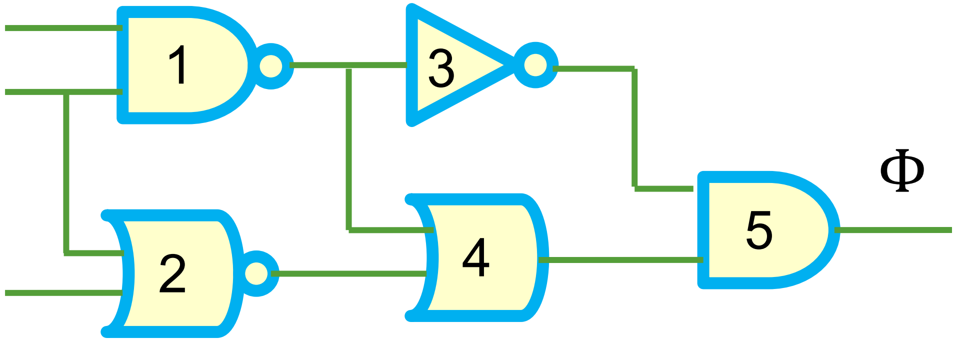
- Do these two logic networks implement the **same** Boolean function?



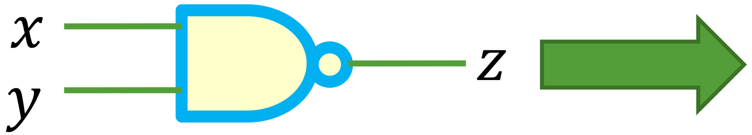
Solution: Do **SAT** on this new network
- If **SAT**: networks not same, and this pattern makes them give different outputs.
- If **unSAT**: yes, same!

Related Question: Circuits → CNF

- How do we start with a gate-level description and get CNF?
  - Isn't this hard? No – it's really easy.



- Idea: build up CNF one gate at a time.
  - We build **gate consistency function** (or **gate satisfiability function**): $\Phi_z(x,y,z) = z \overline{\oplus} f(x,y)$

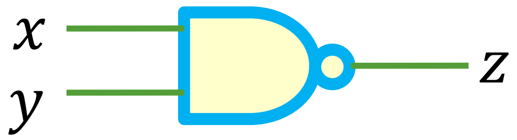


$$\Phi_z = z \overline{\oplus} \overline{xy}$$

$$\Phi_z = (x+z)(y+z)(\bar{x}+\bar{y}+\bar{z})$$

Gate Consistency Function

- **Gate consistency function:** $\Phi_z(x, y, z) = z \overline{\oplus} f(x, y)$
  - It is "1" just for combinations of inputs and the output that are "consistent" with what gate actually does.



$$\Phi_z = (x + z)(y + z)(\bar{x} + \bar{y} + \bar{z})$$

Consistent input: $x = 0, y = 0, z = 1 \Rightarrow \Phi_z = 1$

Inconsistent input: $x = 1, y = 1, z = 1 \Rightarrow \Phi_z = 0$

Rules for ALL Kinds of Basic Gates

$$z = x$$

$$(\bar{x} + z)(x + \bar{z})$$

$$z = \bar{x}$$

$$(x + z)(\bar{x} + \bar{z})$$

Rules for ALL Kinds of Basic Gates

$$z = \mathrm{NOR}(x_1, x_2, \dots, x_n)$$

$$\left[\prod_{i=1}^{n} (\bar{x}_i + \bar{z})\right]\left[\left(\sum_{i=1}^{n} x_i\right) + z\right]$$

$$z = \mathrm{OR}(x_1, x_2, \dots, x_n)$$

$$\left[\prod_{i=1}^{n} (\bar{x}_i + z)\right]\left[\left(\sum_{i=1}^{n} x_i\right) + \bar{z}\right]$$

$$z = \mathrm{NAND}(x_1, x_2, \dots, x_n)$$

$$\left[\prod_{i=1}^{n} (x_i + z)\right]\left[\left(\sum_{i=1}^{n} \bar{x}_i\right) + \bar{z}\right]$$

$$z = \mathrm{AND}(x_1, x_2, \dots, x_n)$$

$$\left[\prod_{i=1}^{n} (x_i + \bar{z})\right]\left[\left(\sum_{i=1}^{n} \bar{x}_i\right) + z\right]$$

Rules for ALL Kinds of Basic Gates

- XOR/XNOR gates are rather **unpleasant** for SAT solver.
  - They have rather large gate consistency functions.
  - Even small 2-input gates create a lot of terms.

$$z = x \oplus y$$

$$\Phi_z = z \overline{\oplus} (x \oplus y)$$
$$= (\bar{x} + \bar{y} + \bar{z})(x + y + \bar{z})$$
$$(x + \bar{y} + z)(\bar{x} + y + z)$$

$$z = x \overline{\oplus} y$$

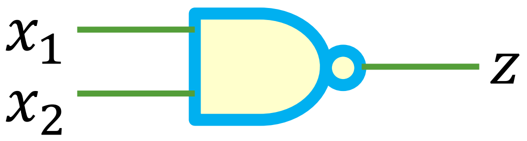
$$\Phi_z = z \overline{\oplus} (x \overline{\oplus} y)$$
$$= (x + y + z)(\bar{x} + \bar{y} + z)$$
$$(x + \bar{y} + \bar{z})(\bar{x} + y + \bar{z})$$

Example: Apply the Rule

$$z = \mathrm{NAND}(x_1, x_2, \ldots, x_n)$$

$$\left[\prod_{i=1}^{n}(x_i + z)\right]\left[\left(\sum_{i=1}^{n}\bar{x}_i\right) + \bar{z}\right]$$

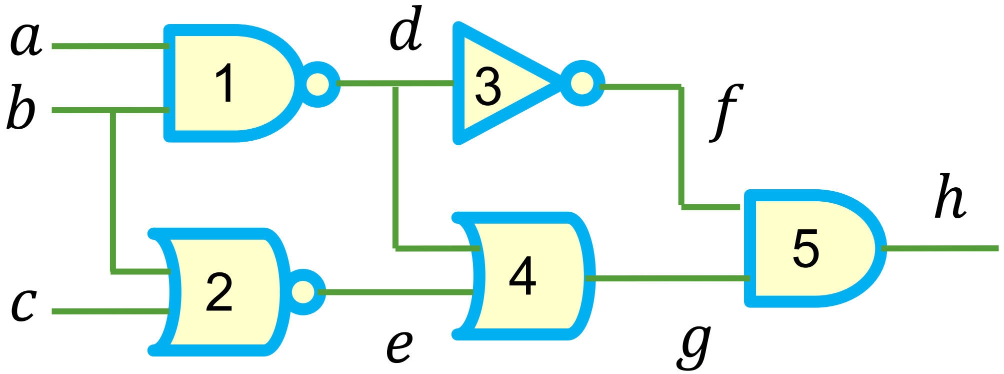
Example: $n = 2$



$$\Phi_z = (x_1 + z)(x_2 + z)(\bar{x}_1 + \bar{x}_2 + \bar{z})$$

Circuits → CNF

- For a network: label each wire, build all gate consistency functions.



$\Phi_d = ?$

$\Phi_e = ?$

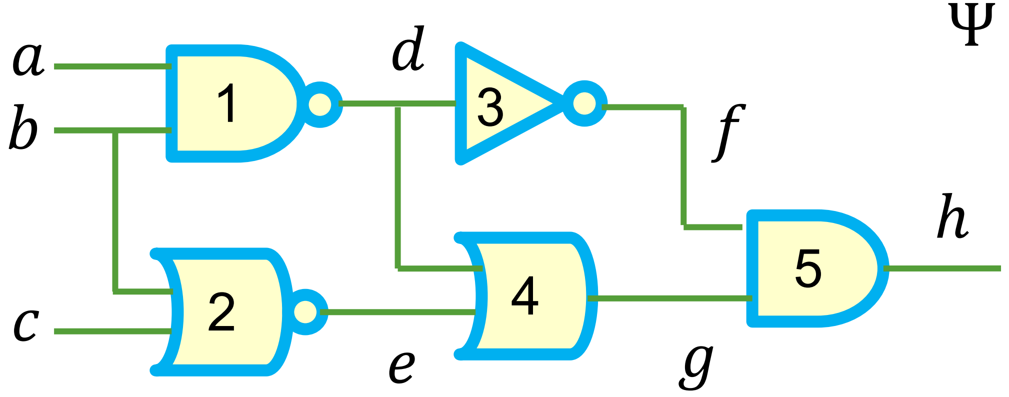
$\Phi_f = ?$

$\Phi_g = ?$

$\Phi_h = ?$

Circuits → CNF

- SAT CNF for network is simple:
  - $\Psi = \boxed{(Output\ Var)} \cdot \prod_{k\ is\ gate\ output\ wire} \Phi_k$
  - Any pattern of that satisfies the function, also makes the gate network output=1.



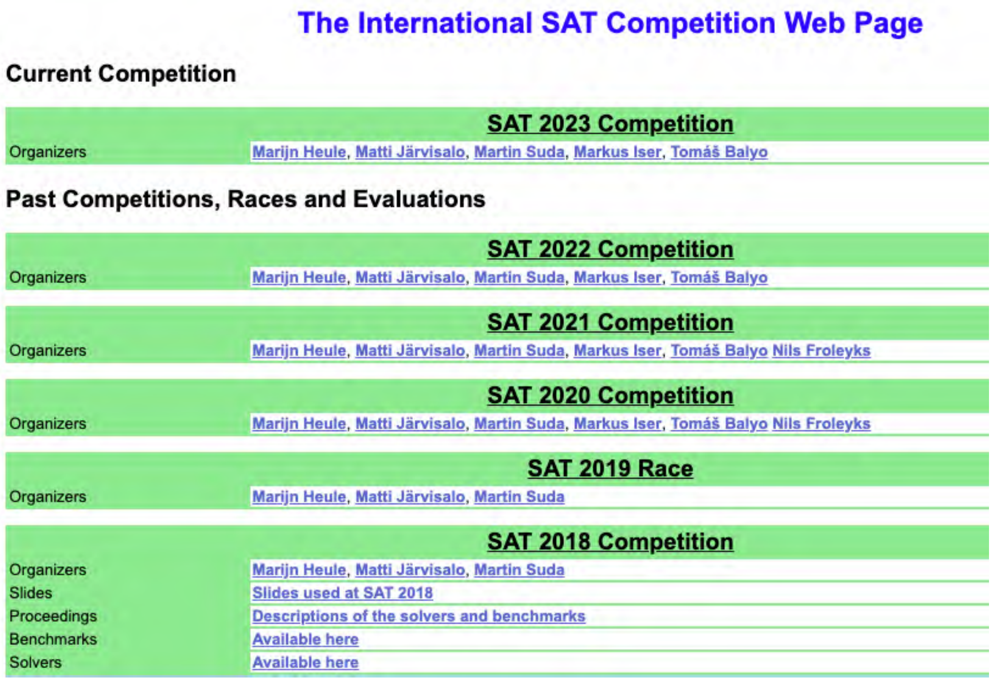
$$\Psi = h \cdot \Phi_d \cdot \Phi_e \cdot \Phi_f \cdot \Phi_g \cdot \Phi_h$$

SAT: Summary

- SAT provides “just solve it” apps.
  - Reason is **scalability**: can do very large problems faster, more reliably.
  - Still, SAT, not guaranteed to find a solution in reasonable time or space.

- 50 years old, but still the big idea: **DPLL**
  - Many recent engineering advances make it amazingly fast.

- SAT contest
  - http://www.satcompetition.org/



The International SAT Competition Web Page

Current Competition

	SAT 2023 Competition
Organizers	Marijn Heule, Matti Järvisalo, Martin Suda, Markus Iser, Tomáš Balyo

Past Competitions, Races and Evaluations

	SAT 2022 Competition
Organizers	Marijn Heule, Matti Järvisalo, Martin Suda, Markus Iser, Tomáš Balyo
	SAT 2021 Competition
Organizers	Marijn Heule, Matti Järvisalo, Martin Suda, Markus Iser, Tomáš Balyo Nils Froleyks
	SAT 2020 Competition
Organizers	Marijn Heule, Matti Järvisalo, Martin Suda, Markus Iser, Tomáš Balyo Nils Froleyks
	SAT 2019 Race
Organizers	Marijn Heule, Matti Järvisalo, Martin Suda
	SAT 2018 Competition
Organizers	Marijn Heule, Matti Järvisalo, Martin Suda
Slides	Slides used at SAT 2018
Proceedings	Descriptions of the solvers and benchmarks
Benchmarks	Available here
Solvers	Available here

Binary Decision Diagrams (BDD)

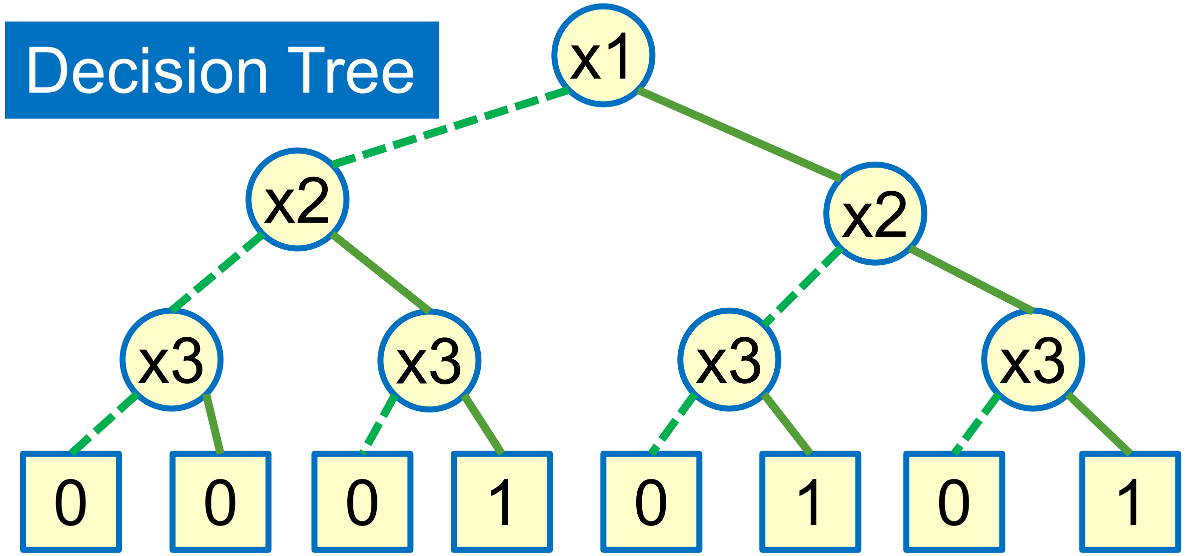
- Originally studied by several people
- … got **practically useful** in 1986
  - Randal Bryant of CMU made breakthrough on **Reduced Ordered BDD (ROBDD)**.
- References

  - Bryant, Randal E. "Graph-based algorithms for boolean function manipulation." *Computers, IEEE Transactions on* 100.8 (1986): 677-691.
  - Brace, Karl S., Richard L. Rudell, and Randal E. Bryant. "Efficient implementation of a BDD package." *27th ACM/IEEE design automation conference*. IEEE, 1990.

Binary Decision Diagrams for Truth Tables

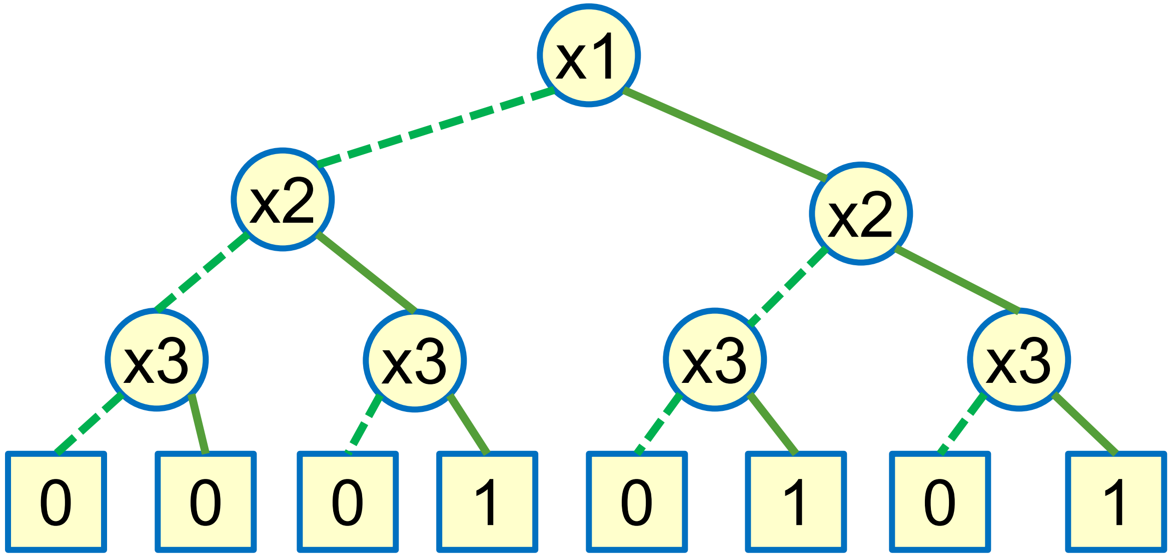
- Big Idea #1: **Binary Decision Diagram**
  - Turn a truth table for the Boolean function into a **Decision Diagram**.
  - In simplest case, graph is just a **tree**.
  - By convention, don't draw arrows on the edges, we know where they go.

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

- **Vertex** represents a **variable**.
- **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.
- 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

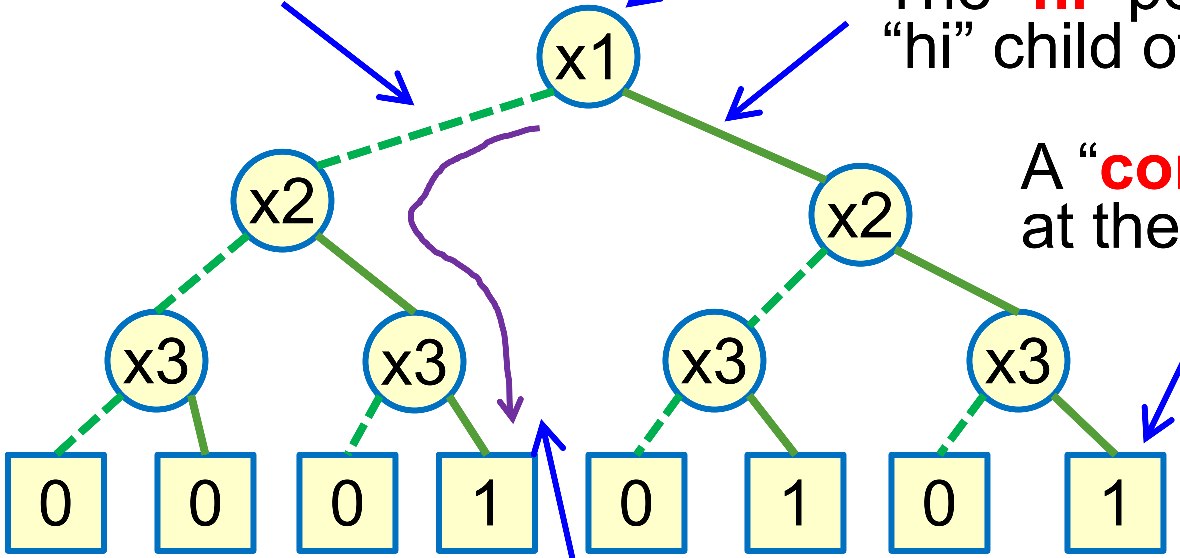
Binary Decision Diagrams: Some Terminology

A "**variable**" vertex

The "**lo**" pointer to "lo" child of vertex

The "**hi**" pointer to "hi" child of vertex

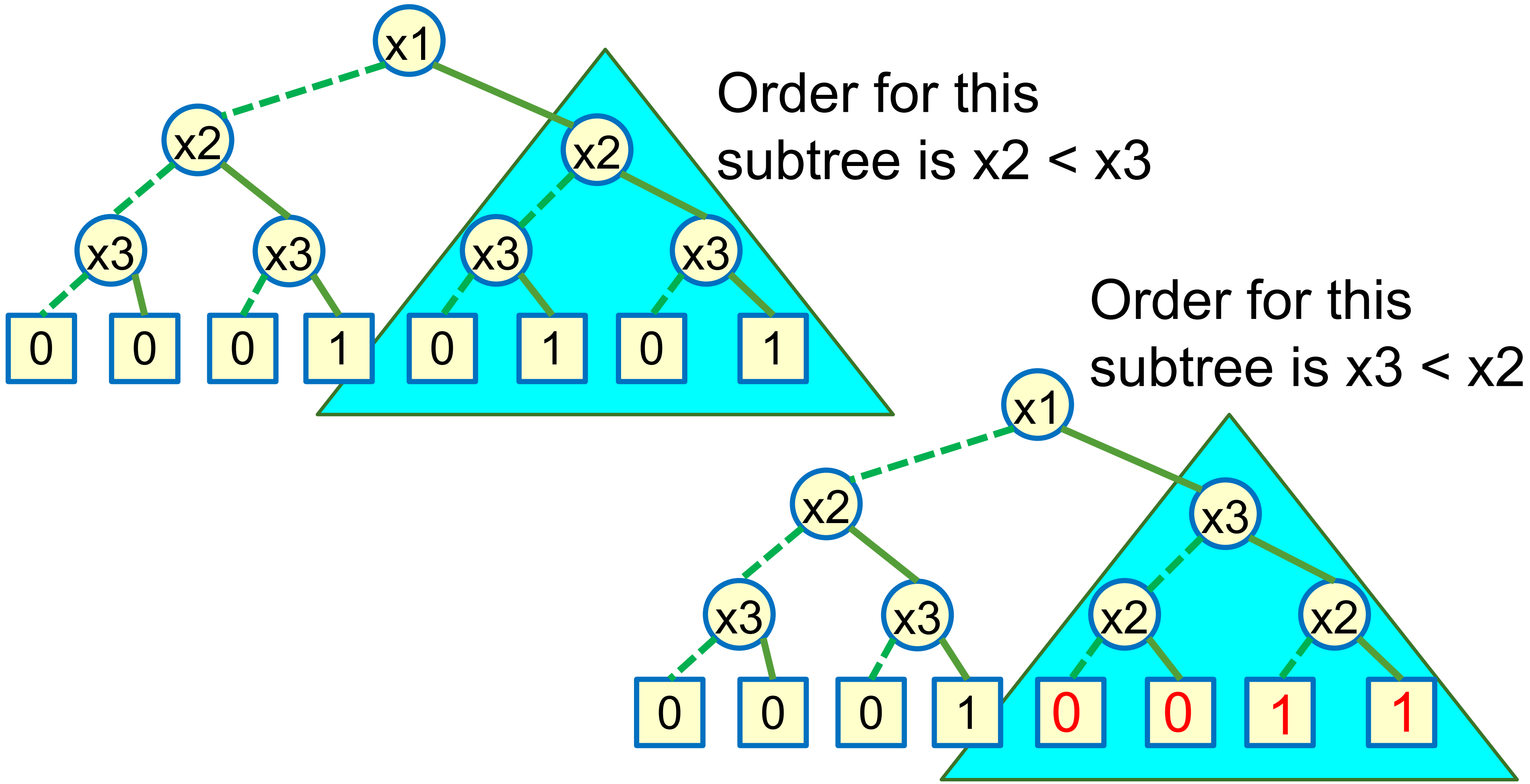
A "**constant**" vertex at the leaf of the tree



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.



Binary Decision Diagrams: Observations

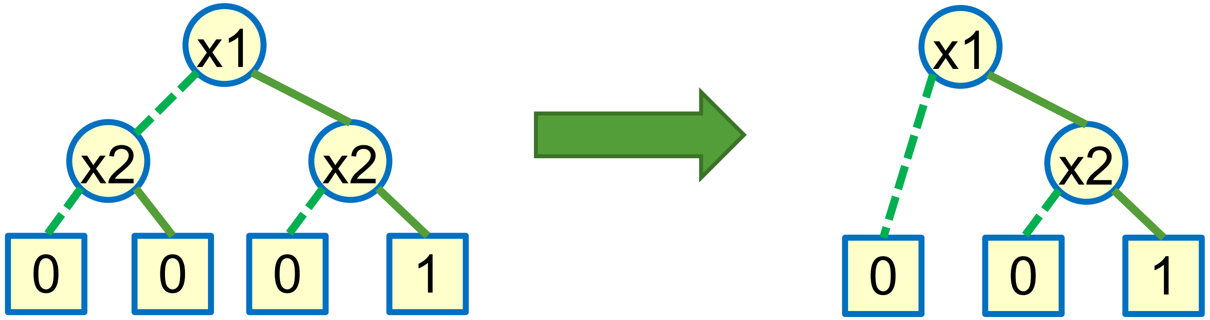
- Each path from root to leaf traverses variables in **some** order.
- Each such path constitutes **a row** of the truth table, i.e., a decision about what output is when variables take particular values.
- However, we have not yet specified anything about the **order** of decisions.
- The decision diagram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

Terminology: Canonical form

- Representation that does not depend on gate implementation of a Boolean function.
- Same function of same variables always produces this exact **same** representation.
- Example: a truth table is **canonical** (up to variable order).
- We want a canonical form data structure.

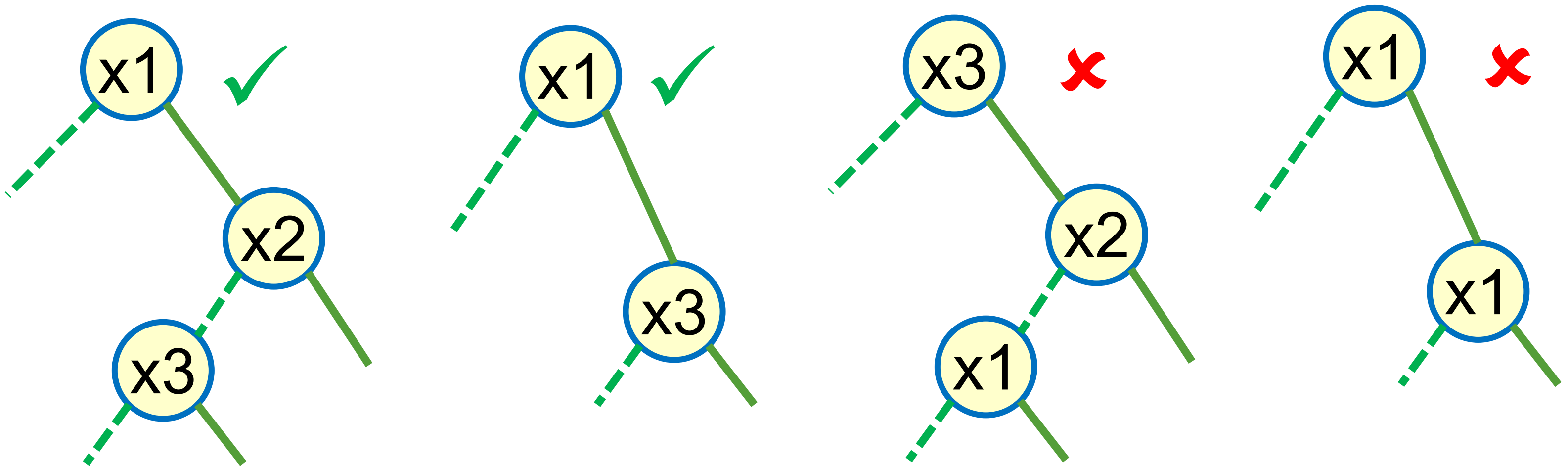
Binary Decision Diagrams

- 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

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

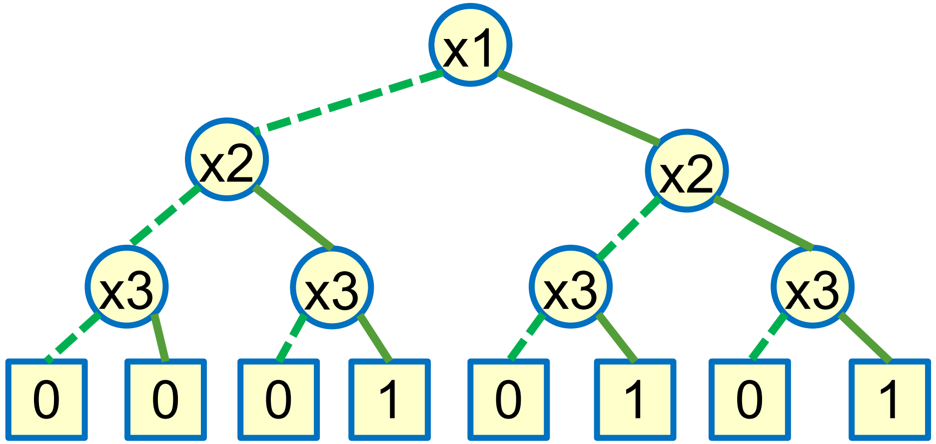


- Property: No **conflicting** assignments along path (see each var **at most once** on path).

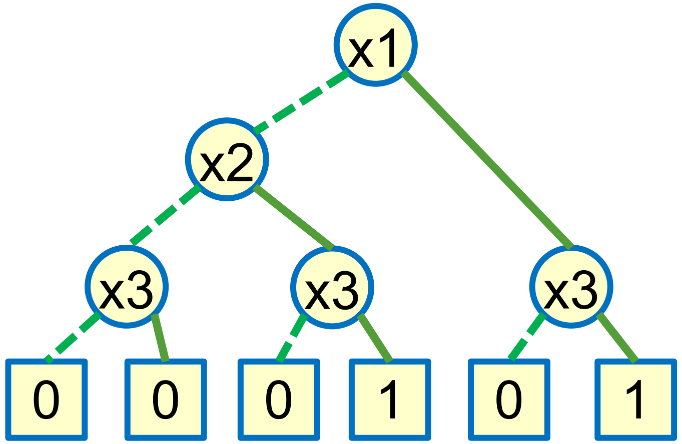
Binary Decision Diagrams

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

Original Decision Diagram

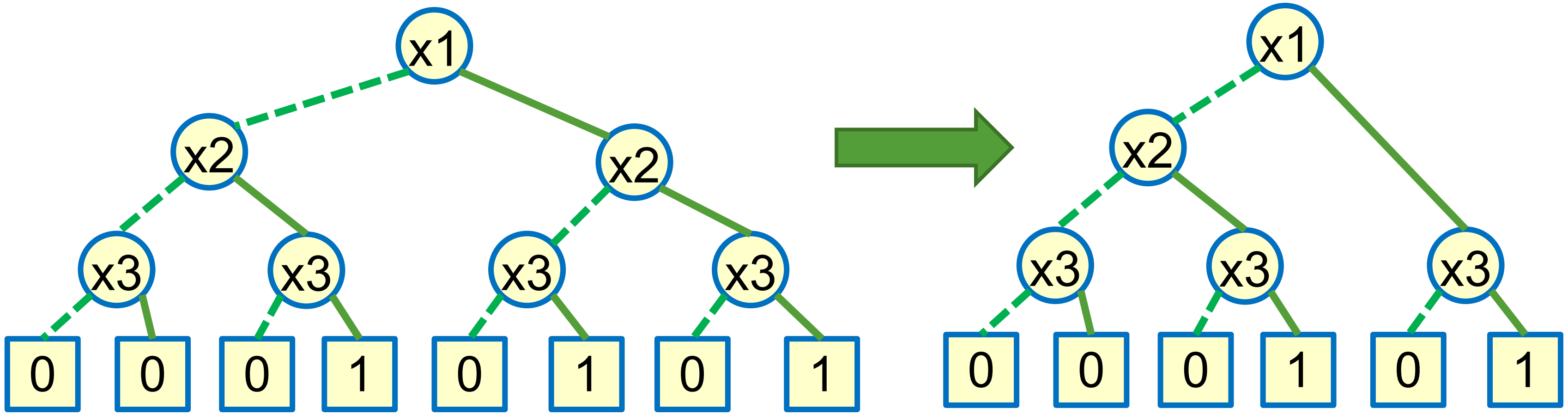


Equivalent, but Different Decision Diagram



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.

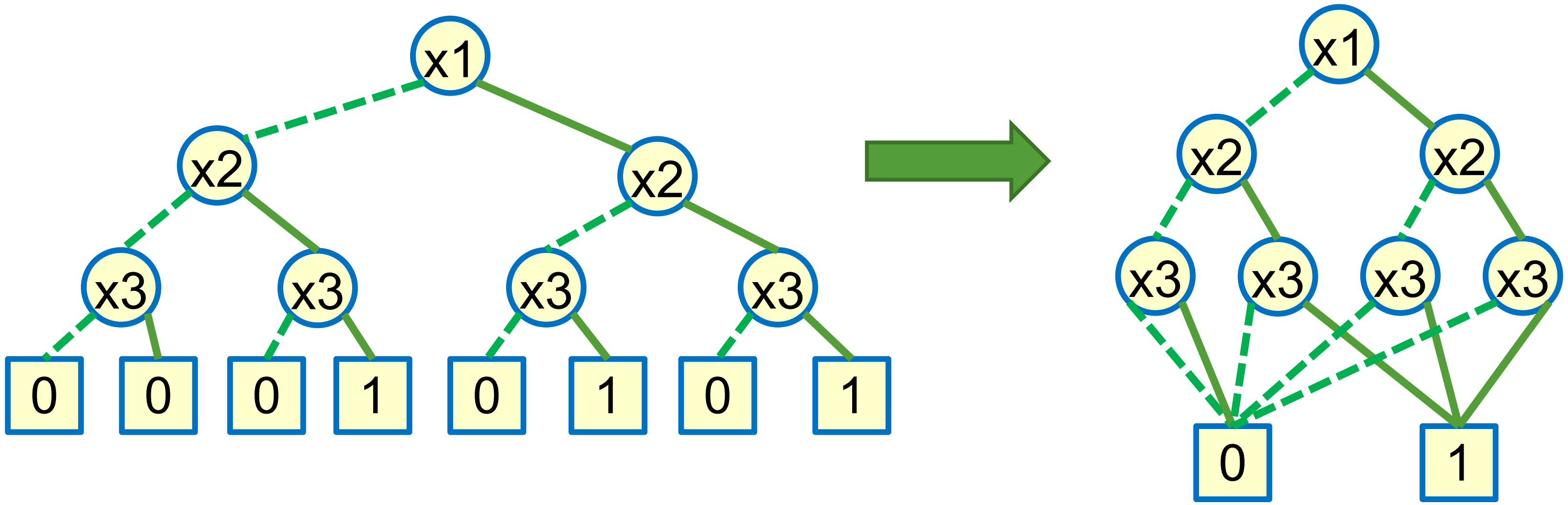


Binary Decision Diagrams: Reduction

- 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.

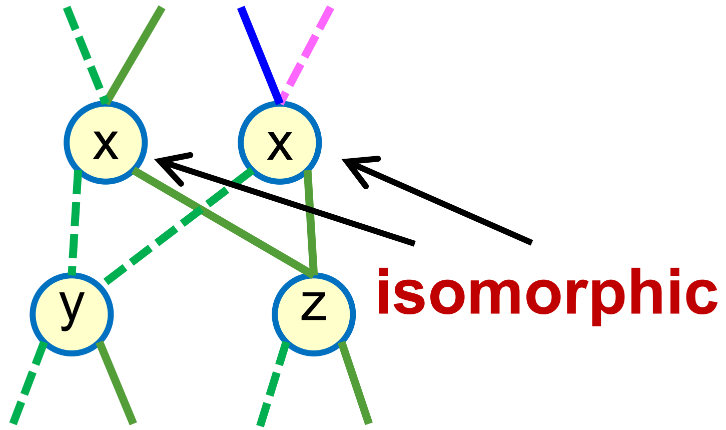
Reduction Rules

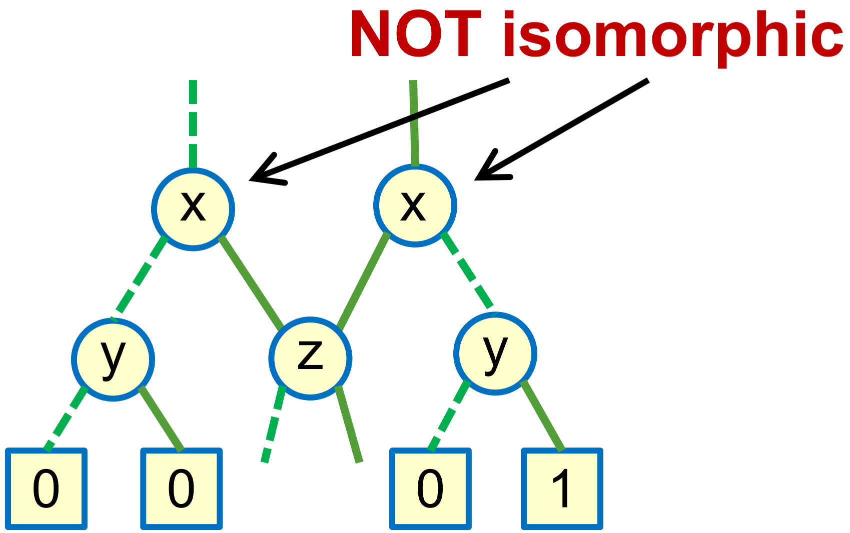
- **Reduction Rule 1**: **Merge equivalent <u>leaves</u>**
  - Just keep one copy of each **constant leaf** – anything else is totally wasteful.
  - Redirect all edges that went into the redundant leaves into this one kept node.
- Apply Rule 1 to our example...



Reduction Rules

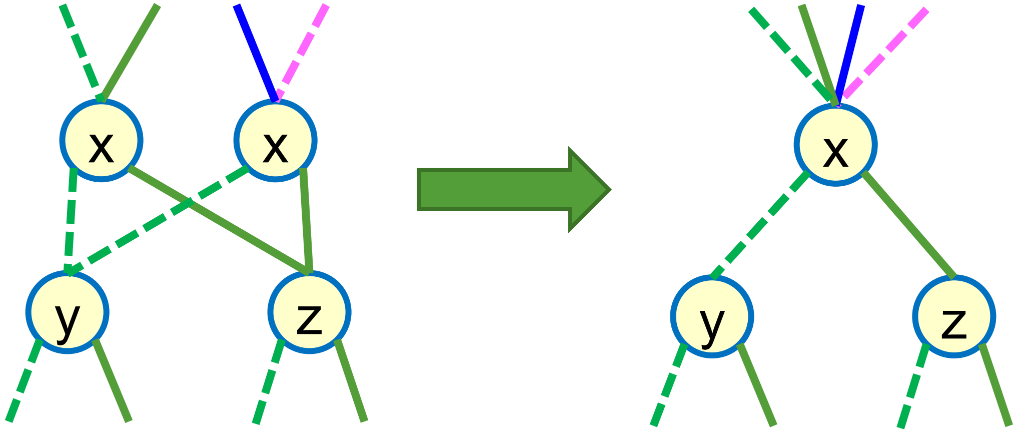
- **Reduction Rule 2**: **Merge isomorphic nodes**
- Isomorphic = 2 nodes with **same** variable and **identical** children
  - Cannot tell these nodes apart from how they contribute to decisions in graph.
  - Note: means exact same physical child nodes, not just children with same label





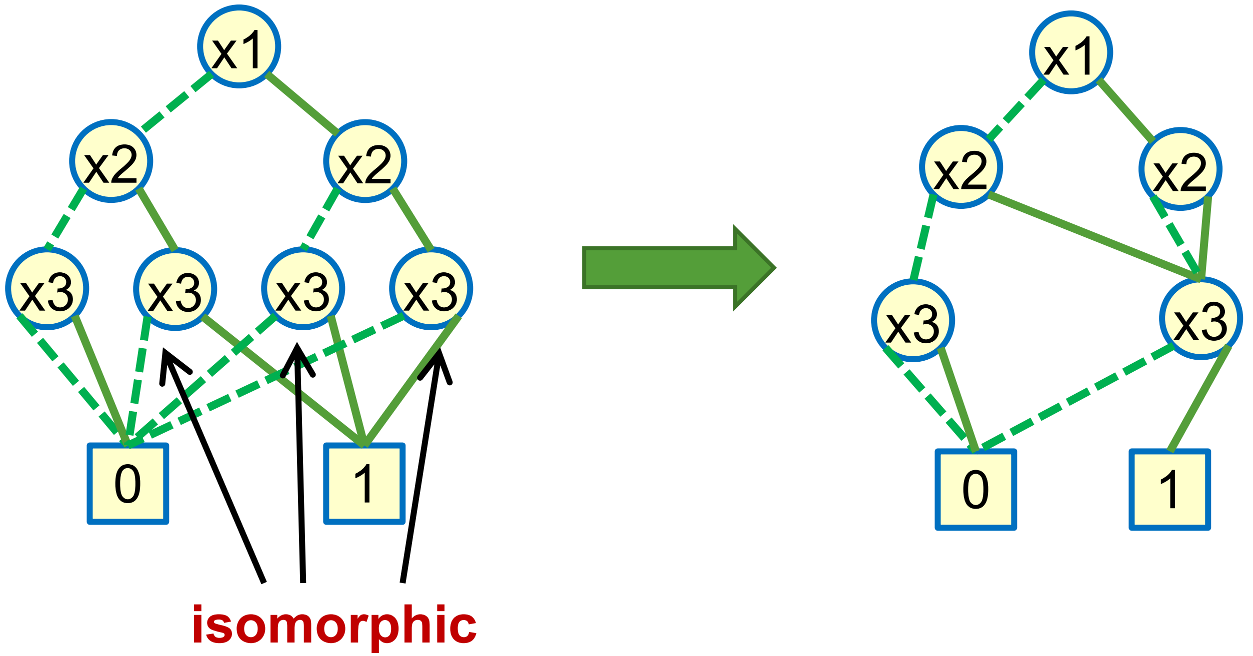
Steps of Merge isomorphic nodes

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 into left as well.



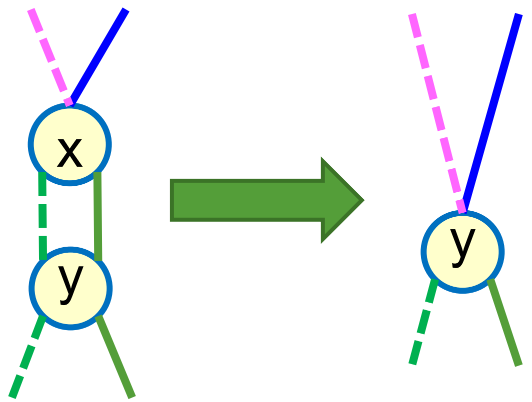
Reduction Rules

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



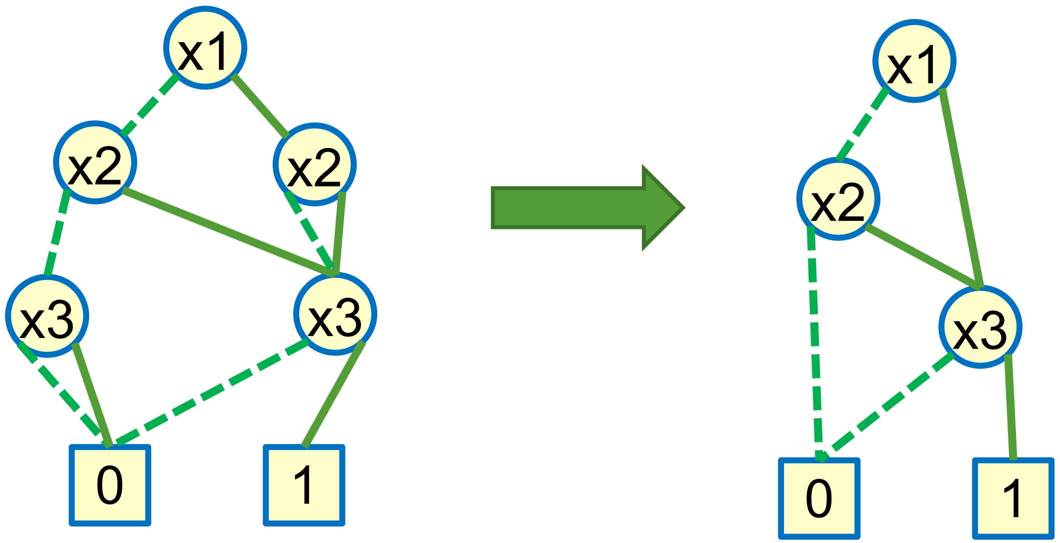
Reduction Rules

- **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, merging redundant nodes, 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…

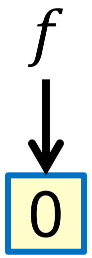
Binary Decision Diagrams: Big Results

- 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**.

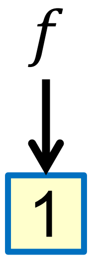
BDDs: Representing Simple Things

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

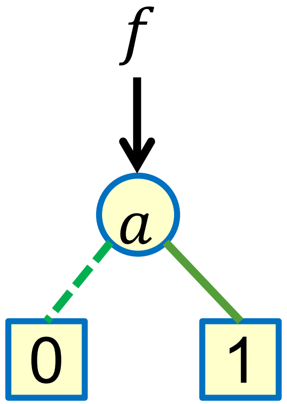
ROBDD for $f(a, b, \dots, z) = 0$



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

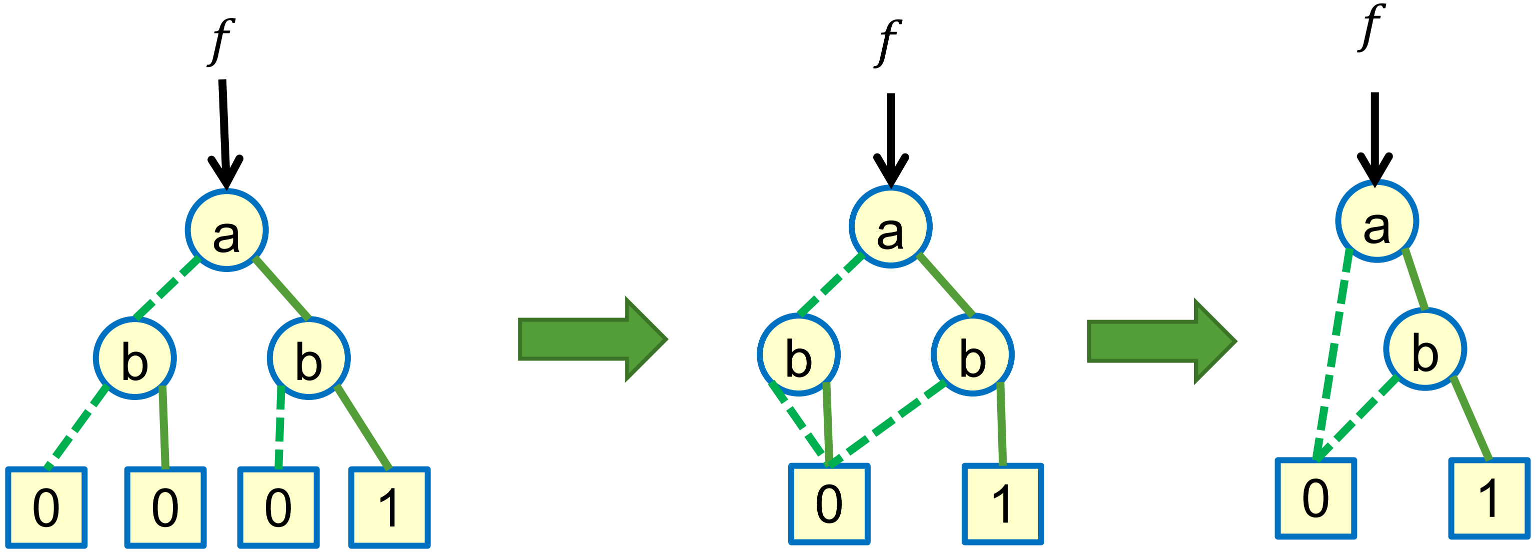


ROBDD for $f(a, b, \dots, z) = a$



ROBDD for AND

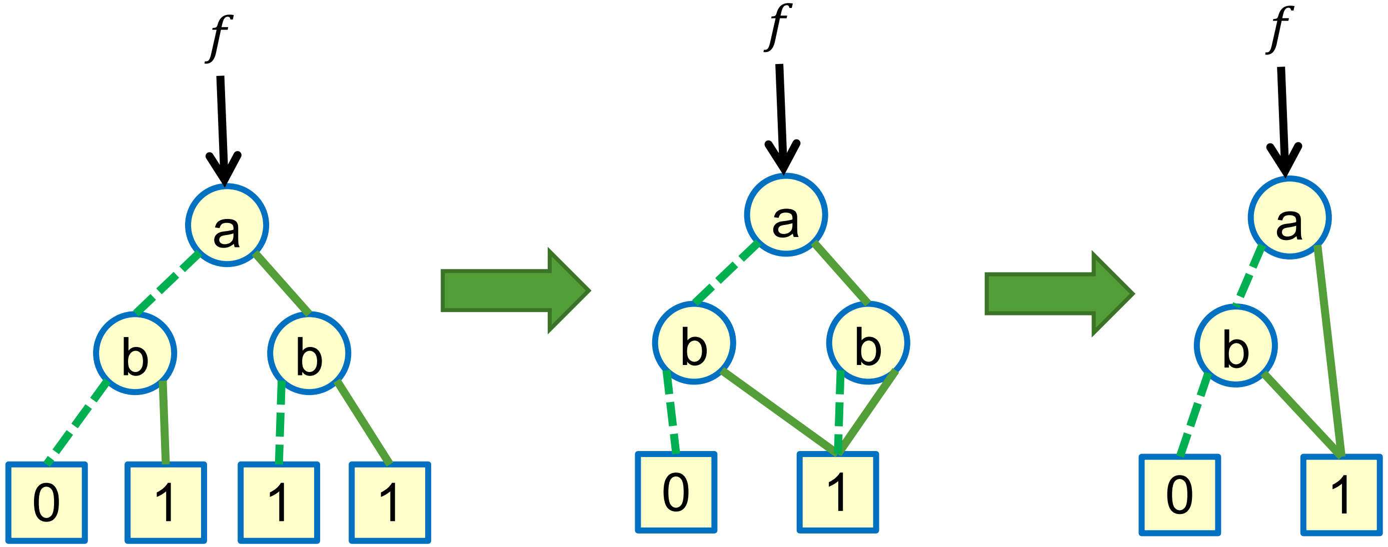
$$f(a, b) = ab$$



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

ROBDD for OR

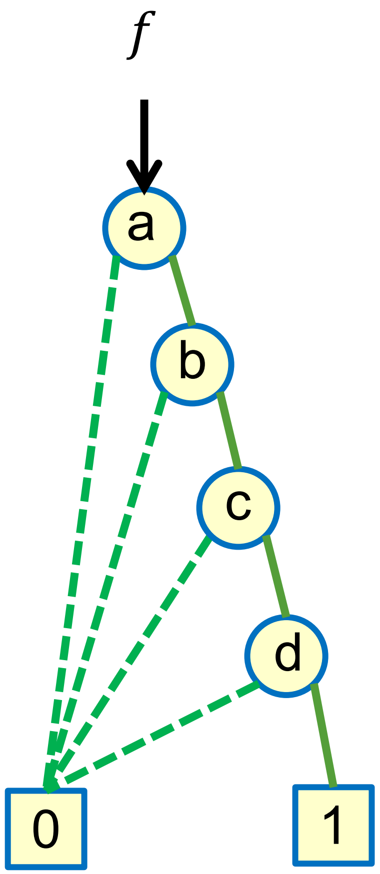
$$f(a,b) = a + b$$



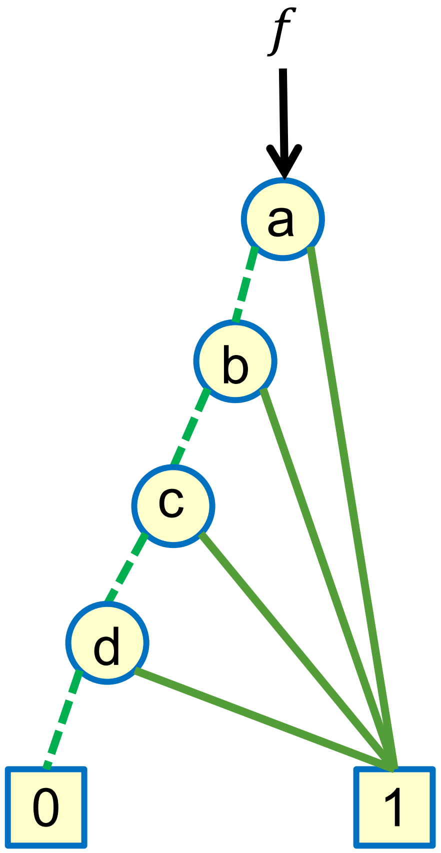
Same graph for $f(a, b, \dots, z) = a + b$

ROBDD for AND/OR on Multiple Inputs

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

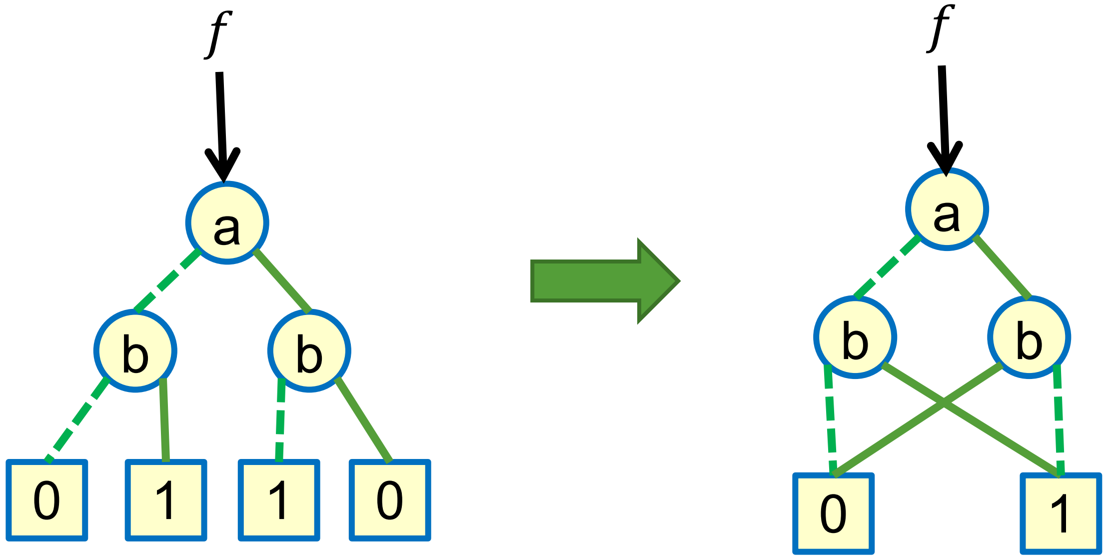


$$f(a,b,c,d) = a + b + c + d$$



ROBDD for XOR

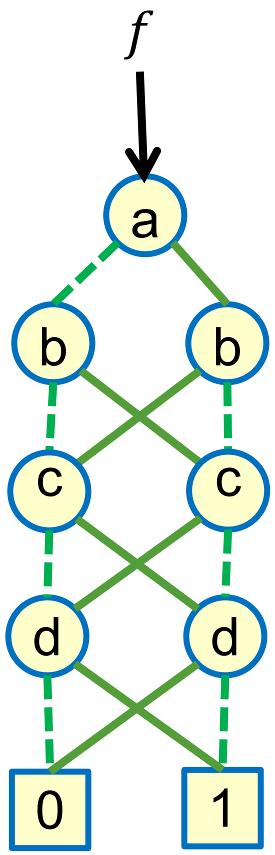
$$f(a, b) = a \oplus b$$



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

ROBDD for XOR on Multiple Inputs

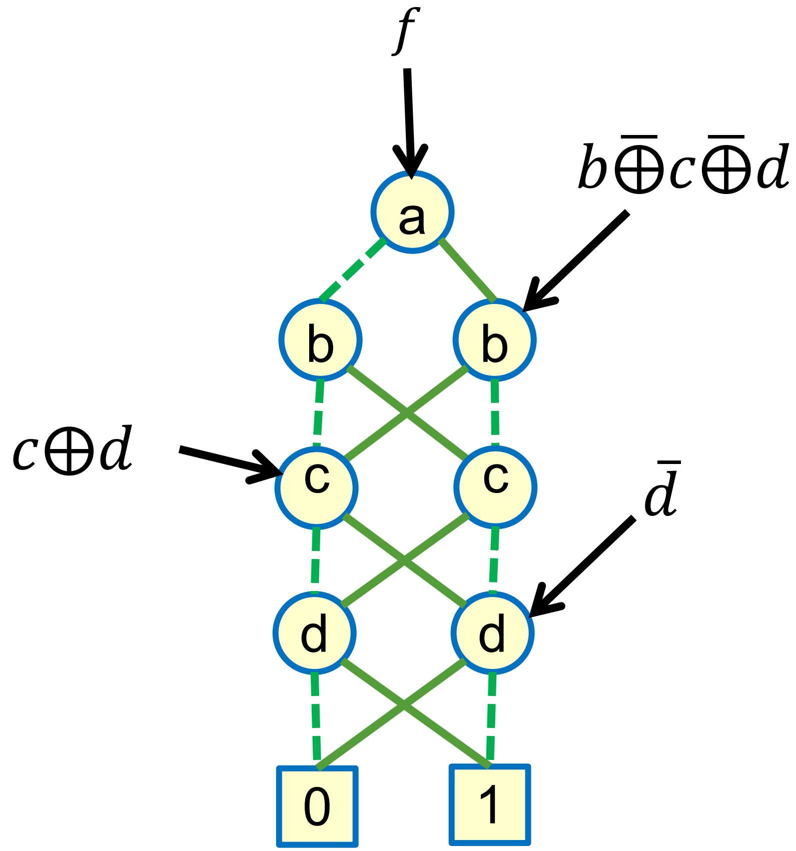
$$f(a, b, c, d) = a \oplus b \oplus c \oplus d$$



Sharing in BDDs

- Very important technical point:
  - **Every** BDD node (not ***just*** root) represents **some** Boolean function in a **canonical** way.
  - BDDs 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

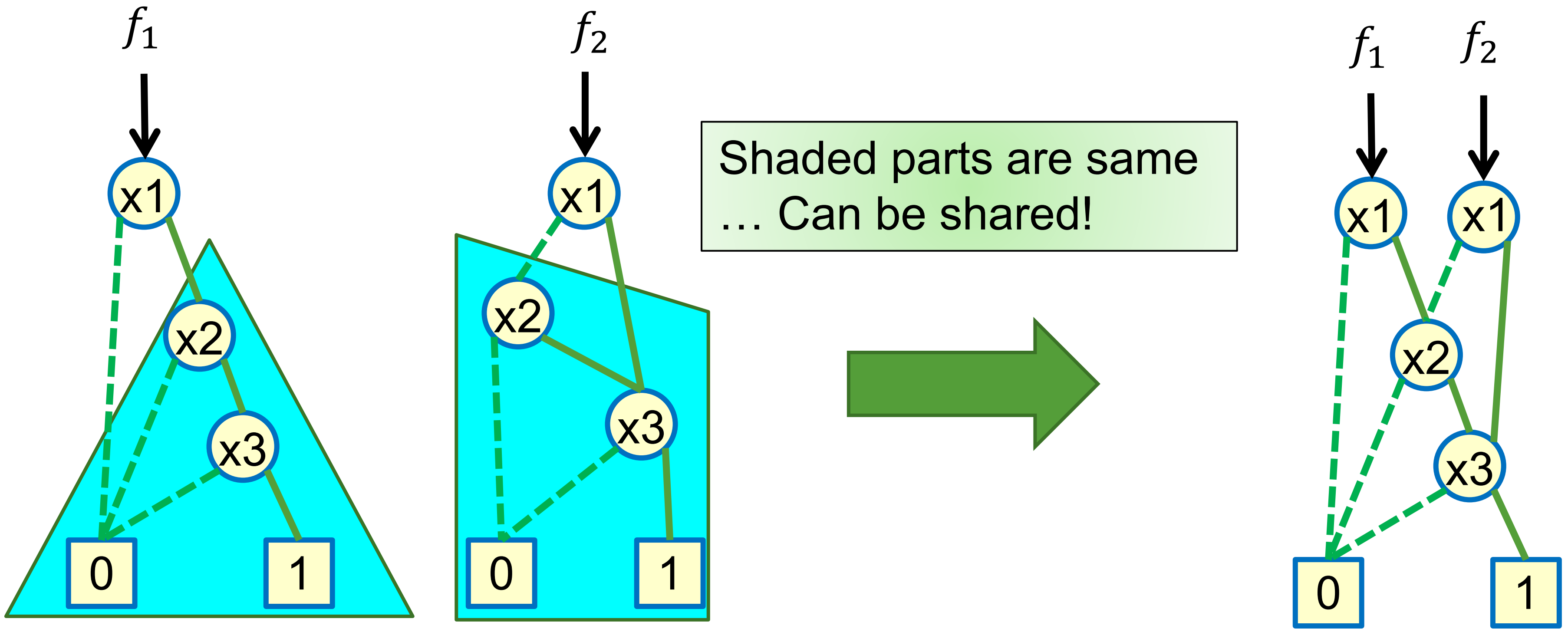
- If we are building BDDs for multiple function,
  - …then there may be **same subgraphs** among different BDDs.
  - Don’t represent same things multiple times; share them!

- 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

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

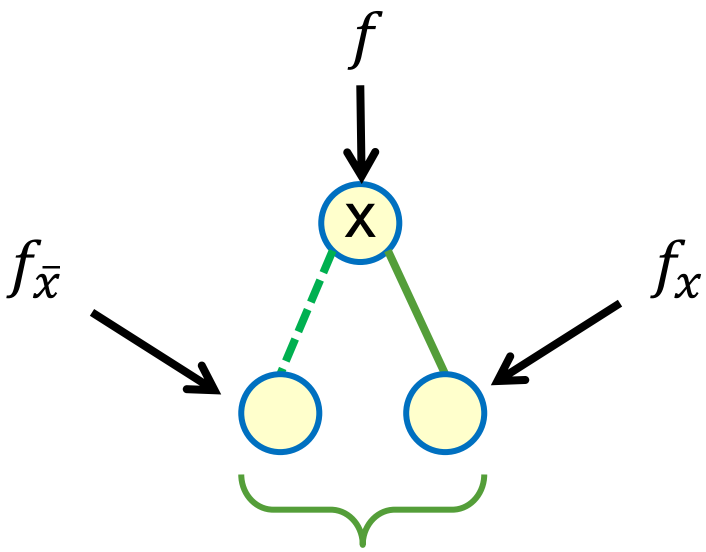


Multi-Rooted BDD: Example

- 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

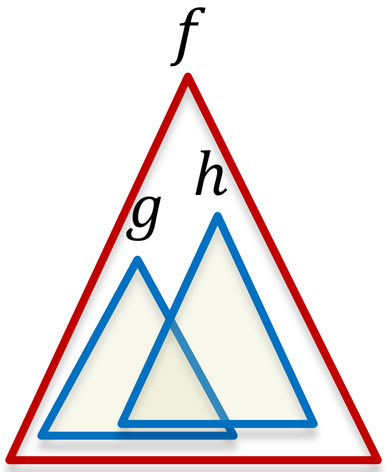
BDD and Cofactors



What are these two functions?

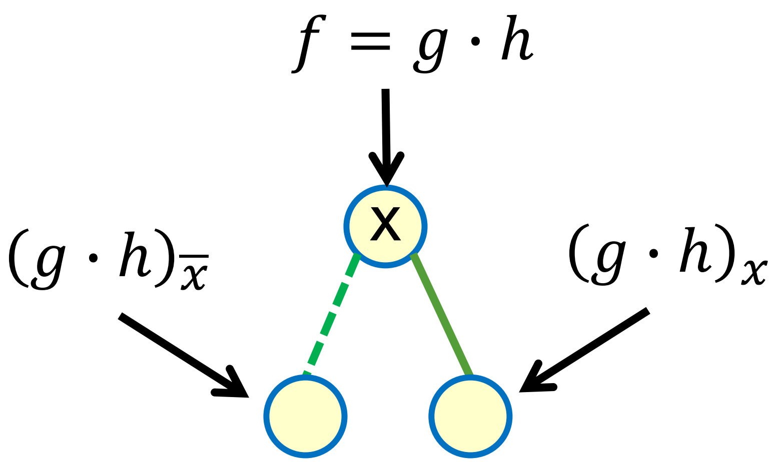
How Are BDDs Really Implemented?

- Recursively!
  - Cofactor and divide-and-conquer are two keys.
- <u>Note</u>: Boolean function can be decomposed: $f = op(g, h)$
  - op can be either AND, OR, XOR, NOT, …
- <u>Idea</u>: 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:

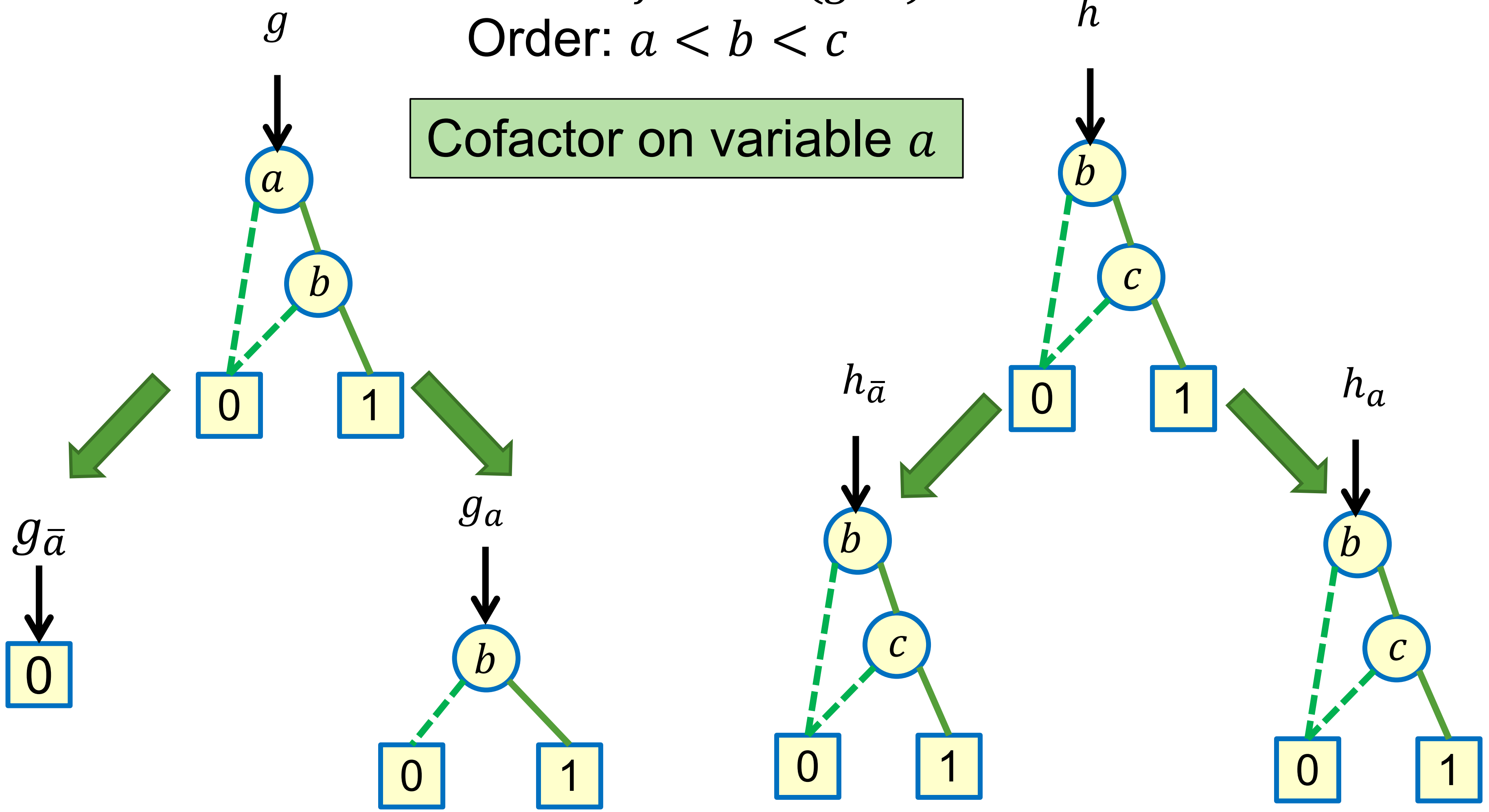
$$negCofBDD(g,x) = g_{\overline{x}} = \begin{cases} g & if\ x < root(g) \\ lo(g) & if\ x = root(g) \end{cases}$$

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

Example of OP

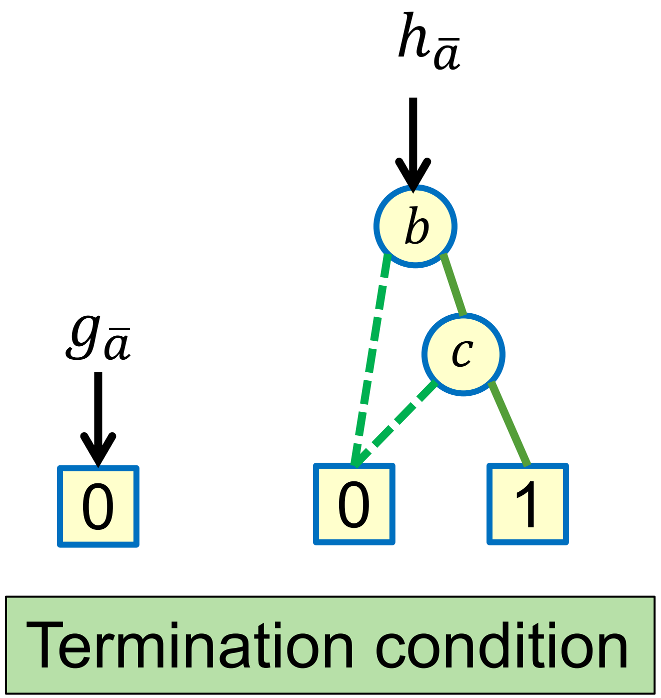
Obtain $f = OR(g, h)$.
Order: $a < b < c$

Cofactor on variable a

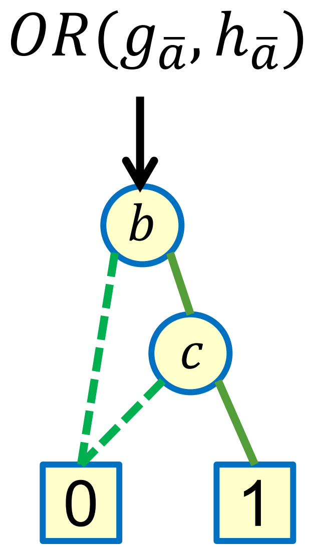


Example of OP (cont.)

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

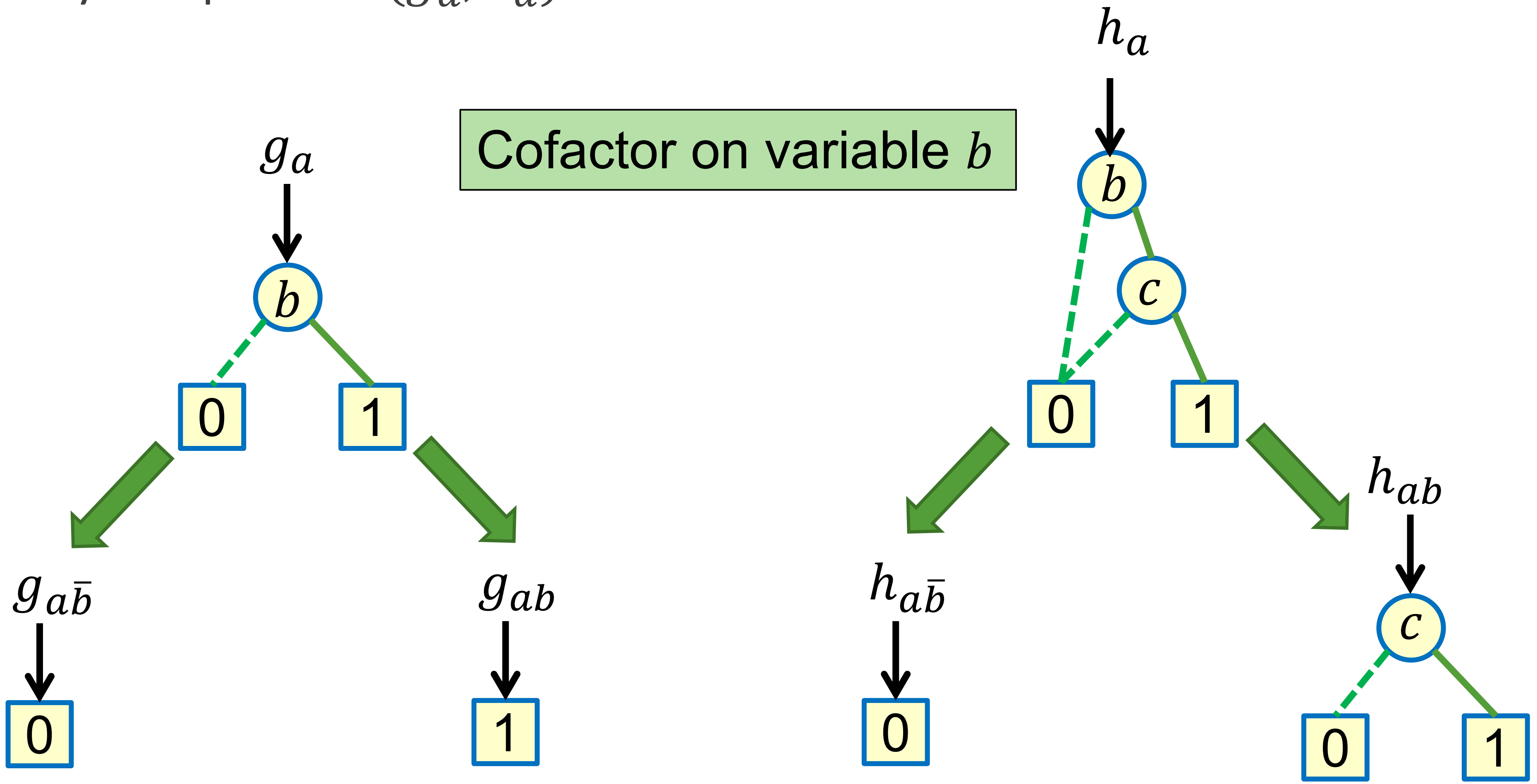


We obtain:



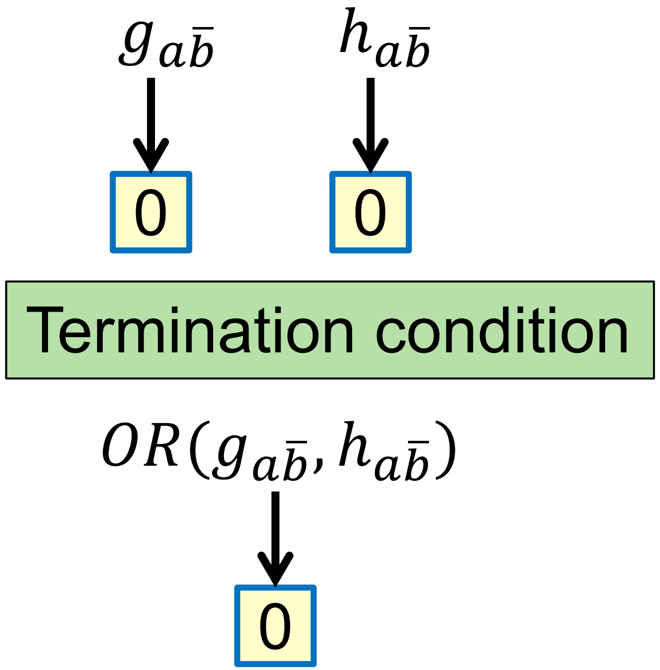
Example of OP (cont.)

- Recursively compute $OR(g_a, h_a)$

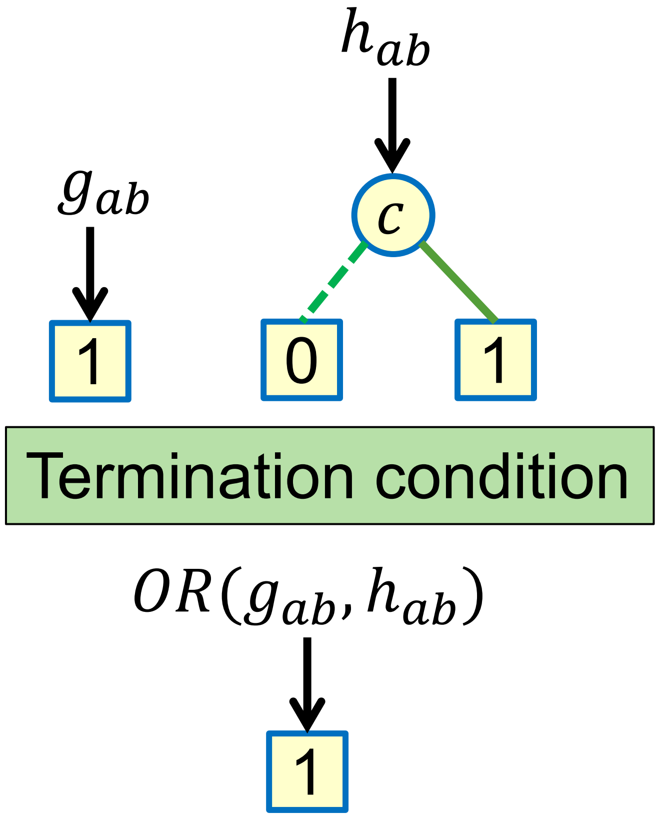


Example of OP (cont.)

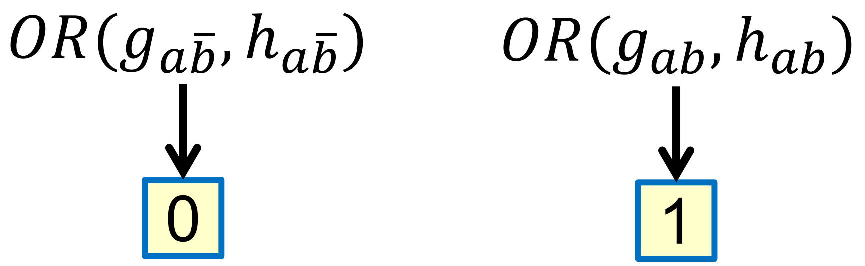
Recursively compute $OR(g_{a\bar{b}}, h_{a\bar{b}})$



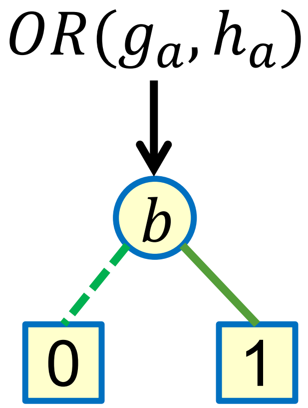
Recursively compute $OR(g_{ab}, h_{ab})$



Example of OP (cont.)

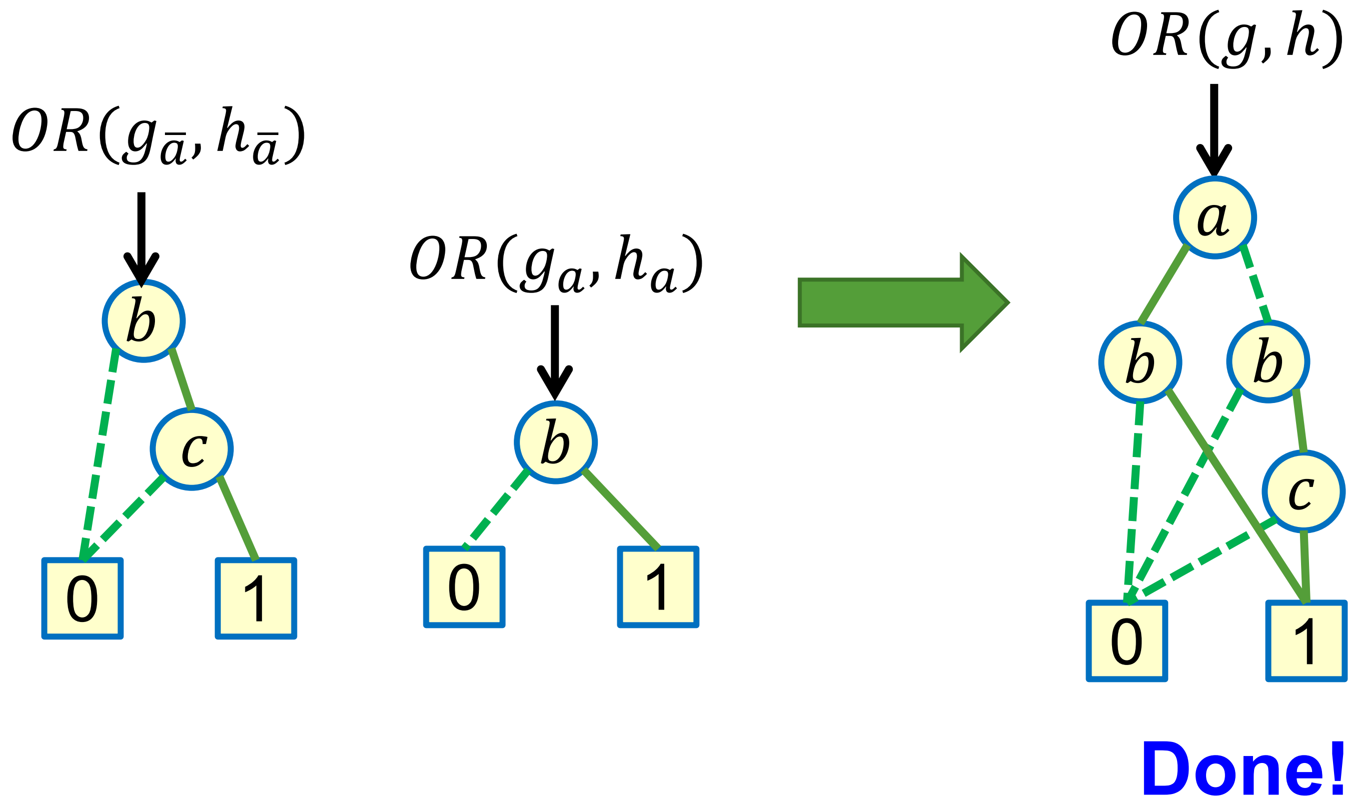


- Based on the recursion results, obtain $OR(g_a, h_a)$
  - **<u>Note</u>**: we cofactor on b.



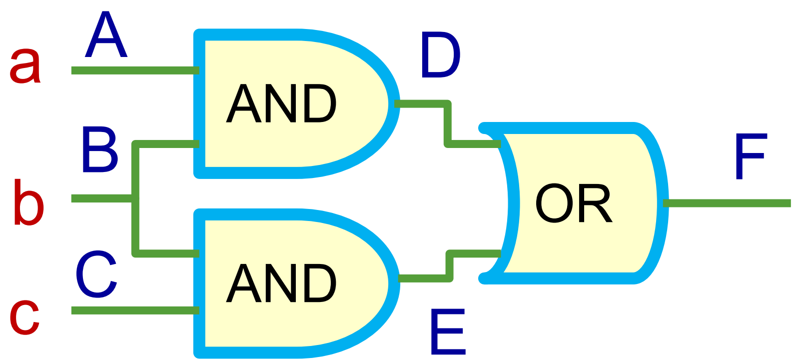
Example of OP (cont.)

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



BDDs: Build Up Incrementally…

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

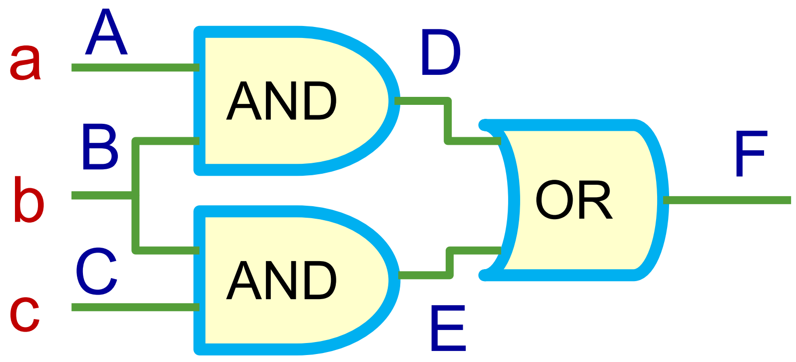


  – Each **input** is a BDD, each **gate** becomes an **operator** *op* that produces a new **output** BDD.
  – Build BDD for F as a **script** of **calls** to basic BDD operators.
  – Stick to a global ordering.

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)

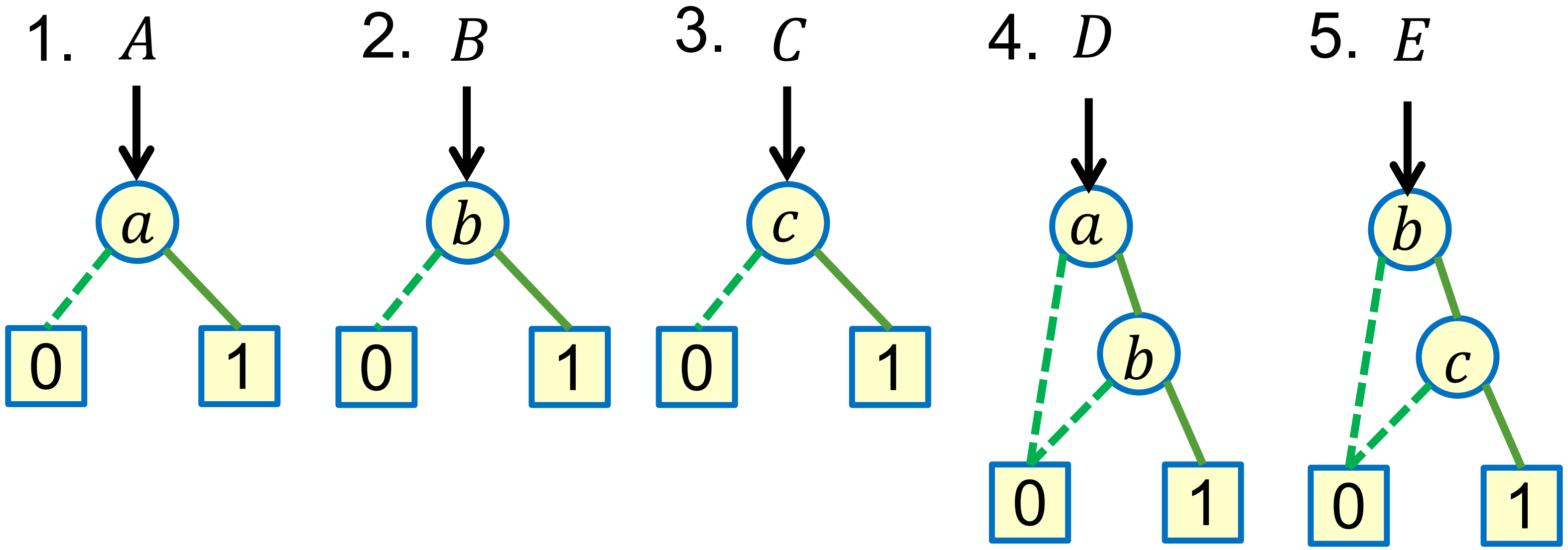
Example: Build BDD Incrementally



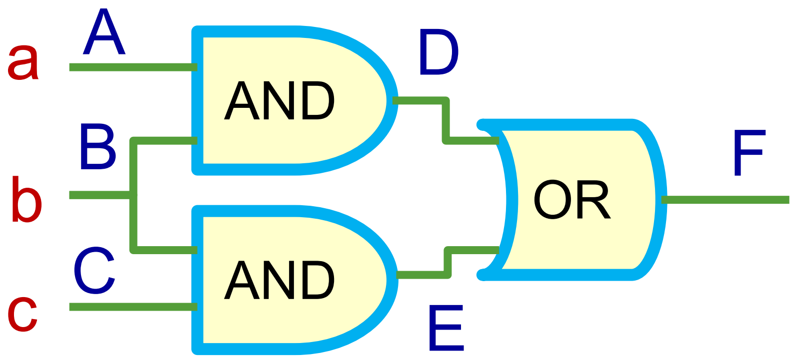
Global ordering: $a < b < c$

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)



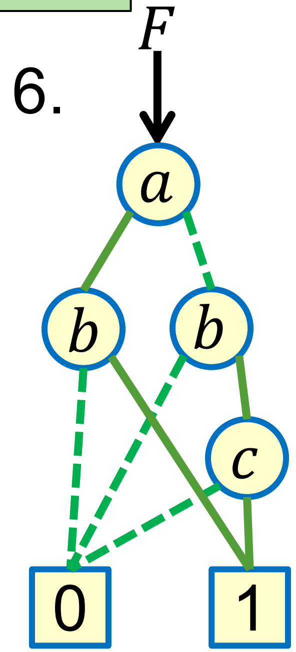
Example: Build BDD Incrementally



Global ordering: $a < b < c$

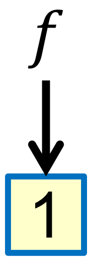
BDD operator script

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)



Application of BDD: Tautology checking

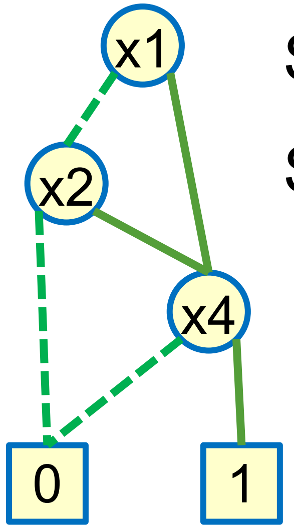
- Solution:
  - Build BDD for f.
  - Check if the BDD is just the BDD for $f = 1$.



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 NO.
  - If yes, any path from root to "1" leaf is a solution.



SAT? Yes.

SAT pattern:

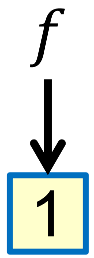
(x_1, x_2, x_3, x_4)
$= (0,1,*,1)$
$(1,*,*,1)$

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

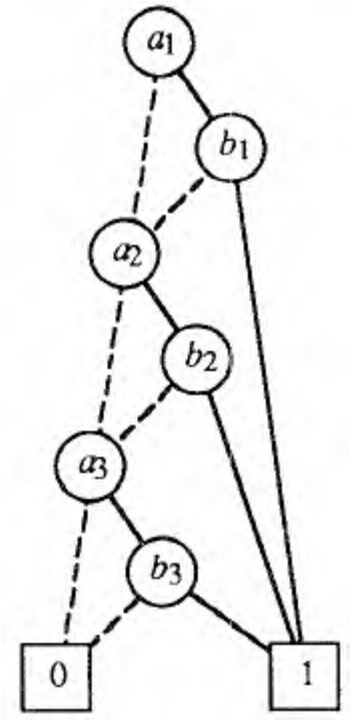
- What inputs make functions F and G give **different answers**?

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

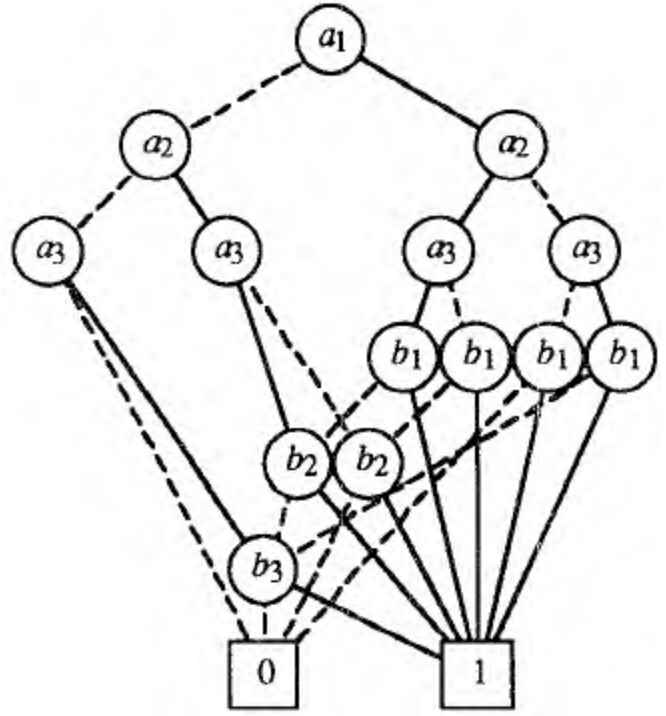
BDDs: Seem Too Good To Be True?!

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

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



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



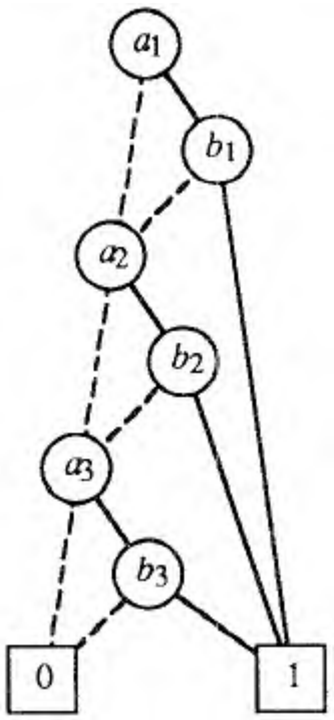
Variable Ordering: How to Handle?

- **Variable ordering heuristics**: make nice BDDs for reasonable problems.
- **Characterization**: know which problems never make simple BDDs (e.g., multipliers)
- **Dynamic ordering**: let the BDD software package pick the order on the fly.

Variable Ordering: Intuition

- Rules of thumb for BDD ordering
  - **Related inputs** should be near each other in order.
  - **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**:

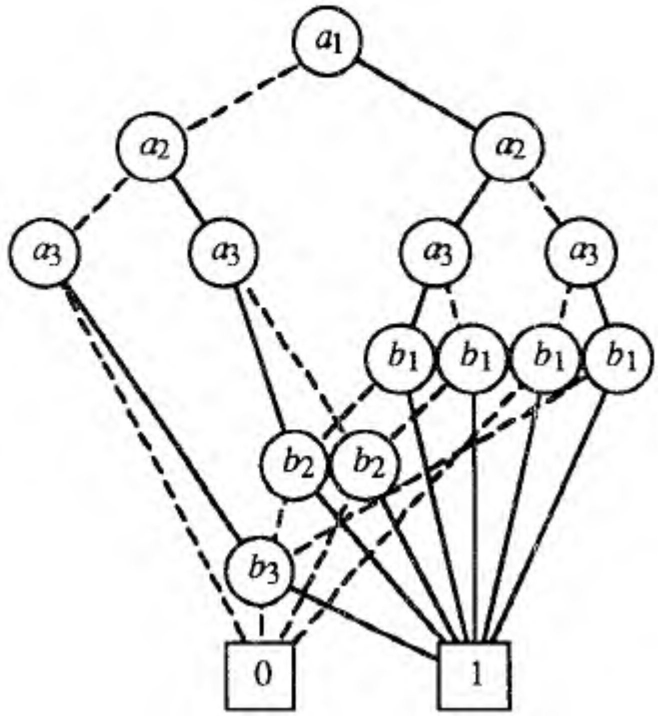
  $a1 < b1 < a2 < b2 < a3 < b3$
  - Why?
    - a_i and b_i together can determine the function value



Variable Ordering: Intuition

- Rules of thumb for BDD ordering
  - **Related inputs** should be near each other in order.
  - **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 **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.
  - People spend a lot of effort to find orderings that work …

BDD Summary

- Reduced, Ordered, Binary Decision Diagrams, ROBDDs
  - Canonical form – a data structure – for Boolean functions.
  - Two Boolean functions 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 or 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.

BDD versus SAT Functionality

- BDD
  - Often work well for many problems.
  - But no guarantee always work.
  - Can build BDD to **represent function** Φ.
    - Can do a big set of Boolean manipulations.
    - But sometimes cannot build BDD with reasonable computer resources (run out of memory SPACE)
  - Problem size **smaller** than SAT.

- SAT
  - Often work well for many problems.
  - But no guarantee always work.
  - Can **solve for SAT** (y/n) on function Φ.
    - Does not support big set of operators.
    - But sometimes cannot find SAT with reasonable computer resources (run out of TIME doing search)
  - Problem size **larger** than BDD.