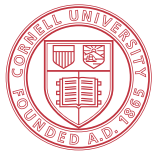
ECE 6775
High-Level Digital Design Automation
Fall 2024

Binary Decision Diagrams (BDDs)



Cornell University

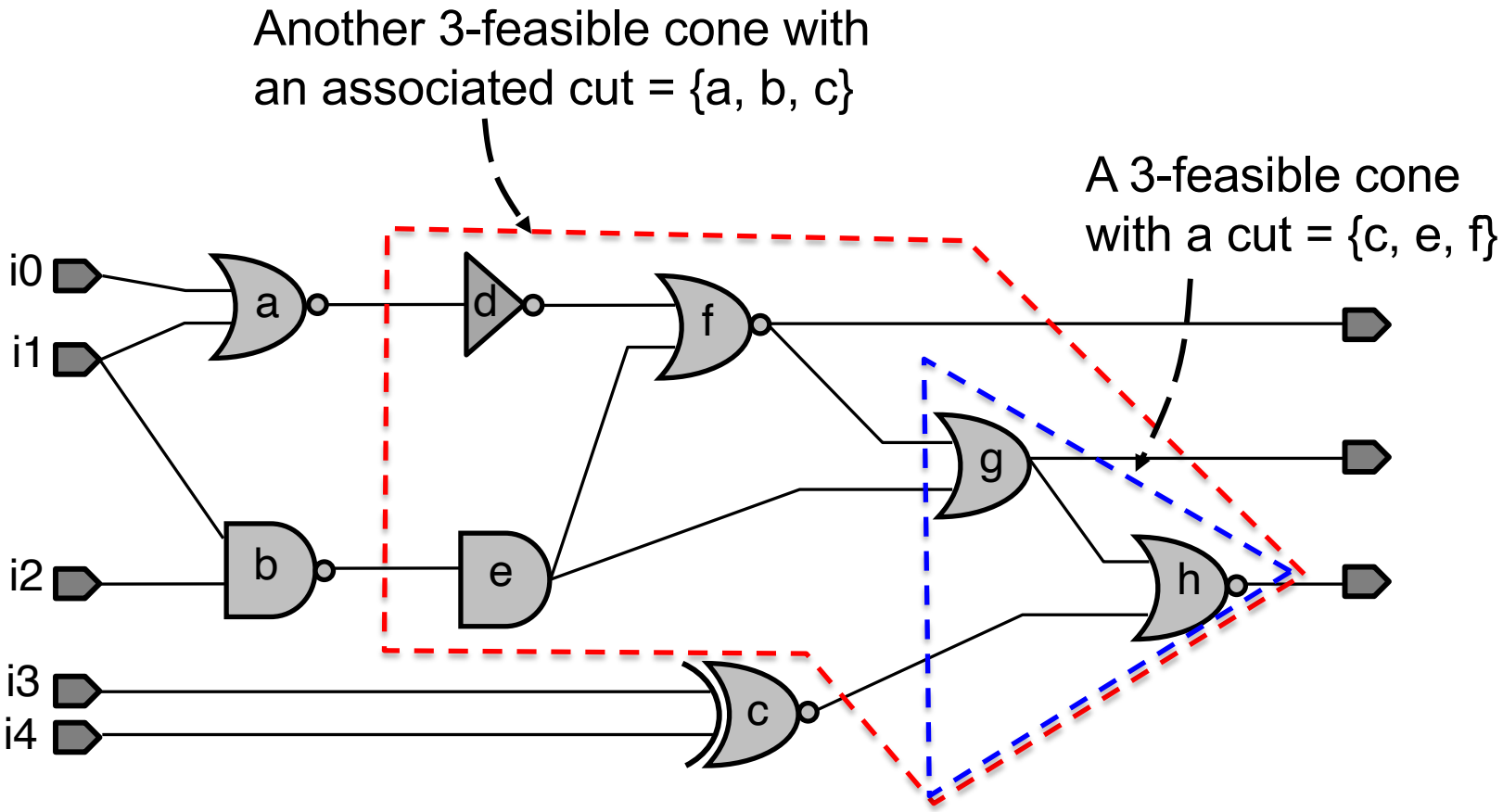
CSL

Announcements

- Lab 2 will be released tomorrow

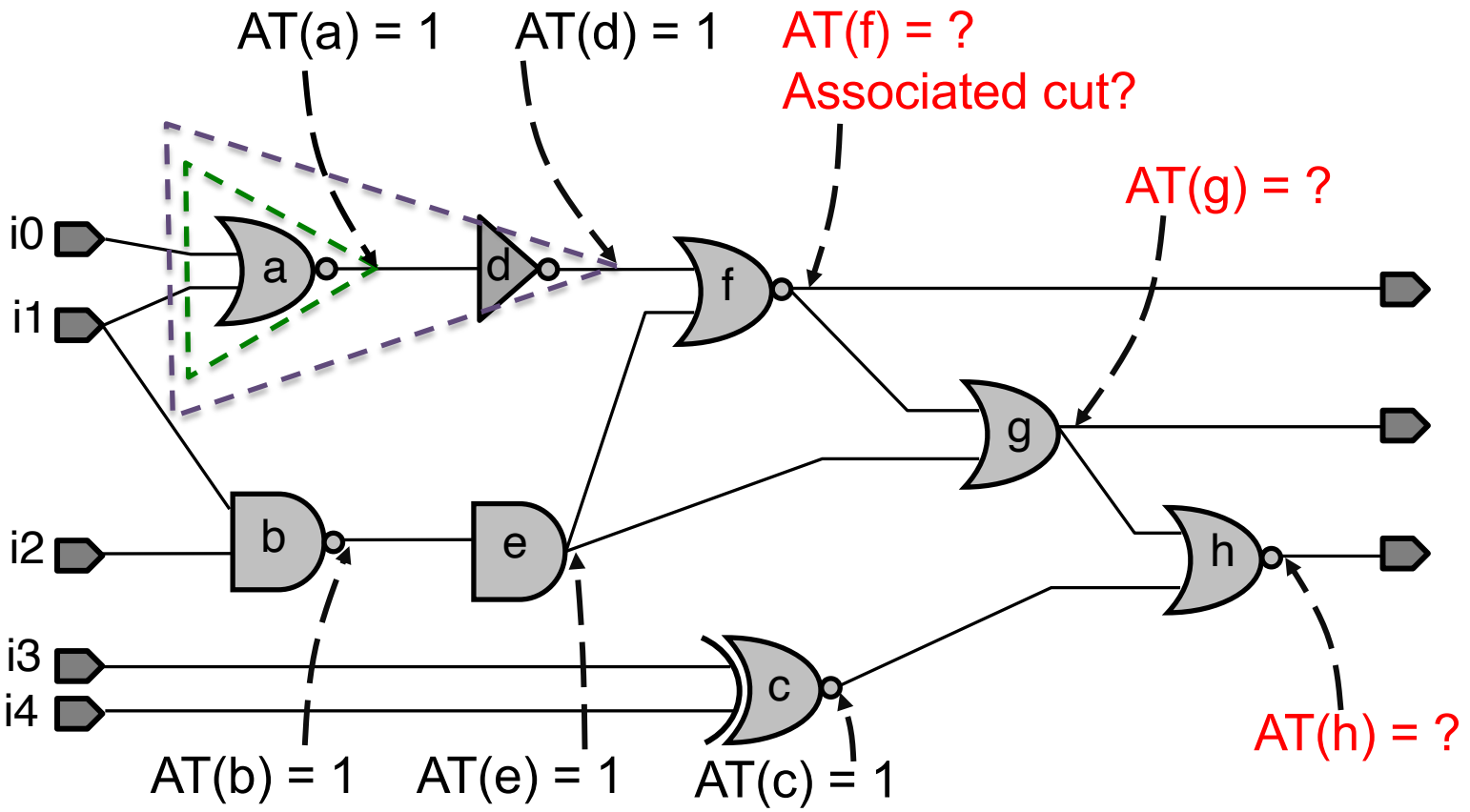
FPGA LUT Mapping Revisited

- Cone C_v : a subgraph rooted on a node v
  - K-feasible cone: #inputs(C_v) ≤ K (**Can occupy a K-input LUT**)
  - K-feasible cut: The set of input nodes of a K-feasible C_v



Timing Analysis with LUT Mapping

- Assumptions
  - K=3
  - All inputs arrive at time 0
  - Unit delay model: 3-input LUT delay = 1; Zero delay on wire
- Question: **Minimum arrival time** (AT) of each gate output?



Agenda

- Introduction to BDD: A canonical graph-based representation of Boolean functions

IEEE TRANSACTIONS ON COMPUTERS, VOL. C-35, NO. 8, AUGUST 1986 677

Graph-Based Algorithms for Boolean Function Manipulation

RANDAL E. BRYANT, MEMBER, IEEE

Abstract—**In this paper we present a new data structure for representing Boolean functions and an associated set of manipulation algorithms. Functions are represented by directed, acyclic graphs in a manner similar to the representations introduced by Lee [1] and Akers [2], but with further restrictions on the ordering of decision variables in the graph. Although a function requires, in the worst case, a graph of size exponential in the number of arguments, many of the functions encountered in typical applications have a more reasonable representation. Our algorithms have time complexity proportional to the sizes of the graphs being operated on, and hence are quite efficient as long as the graphs do not grow too large. We present experimental results from applying these algorithms to problems in logic design verification that demonstrate the practicality of our approach.**

Index Terms—**Boolean functions, binary decision diagrams, logic design verification, symbolic manipulation.**

I. Introduction

BOOLEAN Algebra forms a cornerstone of computer science and digital system design. Many problems in digital logic design and testing, artificial intelligence, and combinatorics can be expressed as a sequence of operations on Boolean functions. Such applications would benefit from efficient algorithms for representing and manipulating Boolean functions symbolically. Unfortunately, many of the tasks one would like to perform with Boolean functions, such as testing whether there exists any assignment of input variables such that a given Boolean expression evaluates to 1 (satisfiability), or two Boolean expressions denote the same function (equivalence) require solutions to NP-complete or co NP-complete problems [3]. Consequently, all known approaches to performing these operations require, in the worst case, an amount of computer time that grows exponentially with the size of the problem. This makes it difficult to compare the relative efficiencies of different approaches to representing and manipulating Boolean functions. In the worst case, all known approaches perform as poorly as the naive approach of representing functions by their truth tables and defining all of the desired operations in terms of their effect on truth table entries. In practice, by utilizing more clever representations and manipulation algorithms, we can often avoid these exponential computations.

Manuscript received November 28, 1984; revised June 11, 1985. This work was supported in part by the Defense Advanced Research Projects Agency under Orders 3771 and 3597.

The author is with the Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213.

IEEE Log Number 8609399.

A variety of methods have been developed for representing and manipulating Boolean functions. Those based on classical representations such as truth tables, Karnaugh maps, or canonical sum-of-products form [4] are quite impractical—*every* function of n arguments has a representation of size 2^n or more. More practical approaches utilize representations that, at least for many functions, are not of exponential size. Example representations include as a reduced sum of products [4] (or equivalently as sets of prime cubes [5]) and factored into unate functions [6]. These representations suffer from several drawbacks. First, certain common functions still require representations of exponential size. For example, the even and odd parity functions serve as worst case examples in all of these representations. Second, while a certain function may have a reasonable representation, performing a simple operation such as complementation could yield a function with an exponential representation. Finally, none of these representations are *canonical forms*, i.e., a given function may have many different representations. Consequently, testing for equivalence or satisfiability can be quite difficult.

Due to these characteristics, most programs that process a sequence of operations on Boolean functions have rather erratic behavior. They proceed at a reasonable pace, but then suddenly "blow up," either running out of storage or failing to complete an operation in a reasonable amount of time.

In this paper we present a new class of algorithms for manipulating Boolean functions represented as directed acyclic graphs. Our representation resembles the binary decision diagram notation introduced by Lee [1] and further popularized by Akers [2]. However, we place further restrictions on the ordering of decision variables in the vertices. These restrictions enable the development of algorithms for manipulating the representations in a more efficient manner.

Our representation has several advantages over previous approaches to Boolean function manipulation. First, most commonly encountered functions have a reasonable representation. For example, all symmetric functions (including even and odd parity) are represented by graphs where the number of vertices grows at most as the square of the number of arguments. Second, the performance of a program based on our algorithms when processing a sequence of operations degrades slowly, if at all. That is, the time complexity of any single operation is bounded by the product of the graph sizes for the functions being operated on. For example, complementing a function requires time proportional to the size of the function graph, while combining two functions with a binary operation (of which intersection, subtraction, and testing for

> One of the only really fundamental data structures that came out in the last twenty-five years
>
> Donald Knuth, 2008

Ideal Representation of a Boolean Function

- We wish to find a representation with the following characteristics
  - **Compact** in terms of size
  - **Efficient** to compute the output with the given inputs and efficient to manipulate and modify
  - *Ideally*, a **canonical** representation
    - Equivalent functions have the same unique form (under certain restrictions)

Example: Voting Function

- A Boolean voting function
  - An *n*-ary Boolean function $f(x_1, x_2, \ldots, x_n)$ evaluates to 1 if 50% or more ($\geq \lceil n/2 \rceil$) of its inputs are set to 1
  - Examples:
    - f(0,0) = 0
    - f(0,1) = 1
    - f(0,0,1) = 0
    - f(1,0,1) = 1

- How to formally represent this function?
  - Truth table
  - Karnaugh map
  - Sum of Products (SOP)
  - …

Truth Table and Canonical Sum

x	y	z	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	1
1	1	1	1

Truth table is canonical

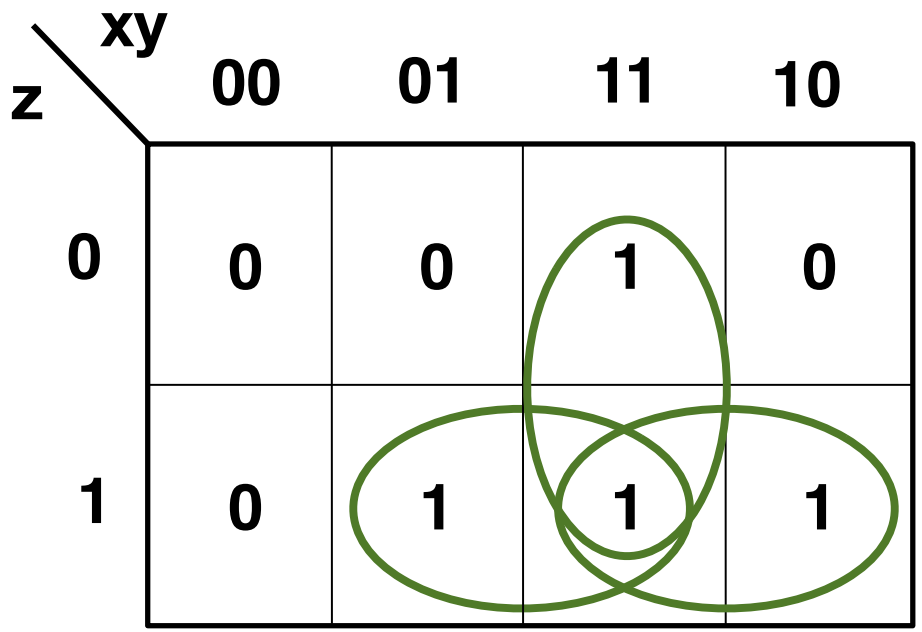
But 2^n table entries are required!

Canonical sum of products (SOP)

xyz' + xy'z + xyz + x'yz
(4 minterms)

Is it a compact form?

Karnaugh Map and Minimized SOP



Minimized SOP (3 terms): xy + xz + yz

What about *n* inputs? (esp. where *n* is large)

Note: K-map only handles up to 6 inputs, and the solution is not necessarily unique

Complexity of SOP Representation

- An n-input Boolean voting function has at least C(n, n/2) prime implicants

- Growth rate of C(*n*, *k*) in terms of *n*
  - For *k*=1, C(*n*,1) = *n*
  - For *k*=2, C(*n*,2) = *n*(*n*-1)/2
  - For *k*=3, C(*n*,3) = *n*(*n*-1)(*n*-2)/6
  - …
  - For k=*n*/2, C(*n*, *n*/2) = $\frac{n!}{[(n/2)!]^2} \in \Theta(2^n n^{-0.5})$
    (uses Stirling formula)

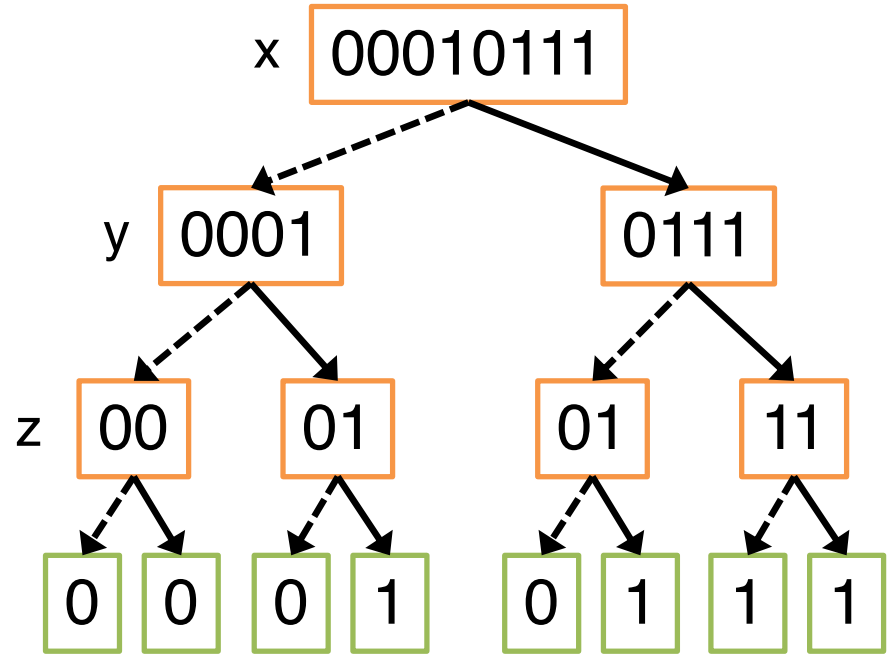
Co-factors and Shannon Expansion

- The co-factor of a Boolean function $f(x_1, x_2, \ldots, x_n)$ is the result of simplifying the function with respect to a specific variable, either by setting that variable to 1 or 0
  - $f_{x_i=1}$ denotes the **positive co-factor** with respect to x_i, which is obtained by substituting $x_i = 1$ into the original function f
    - For example, $f_{x_1=1} = f(1, x_2, \ldots, x_n)$
  - $f_{x_i=0}$ denotes the **negative co-factor** with respect to x_i, which is obtained by substituting $x_i = 0$ into f

- The **Shannon expansion** with respect to a variable x_i:
  $f(x_1, x_2, \ldots, x_n) = x_i \bullet f_{x_i=1} + x_i' \bullet f_{x_i=0}$

Boolean Function in a Decision Tree

	x	y	z	f
x=0, y=0	0	0	0	0
	0	0	1	0
x=0, y=1	0	1	0	0
	0	1	1	1
x=1	1	0	0	0
	1	0	1	1
	1	1	0	1
	1	1	1	1



Shannon Expansion

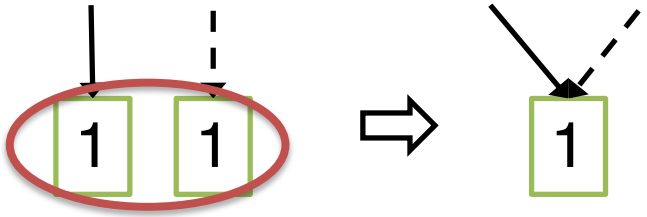
$f(x, y, z) = x' \bullet f_{x=0} + x \bullet f_{x=1}$

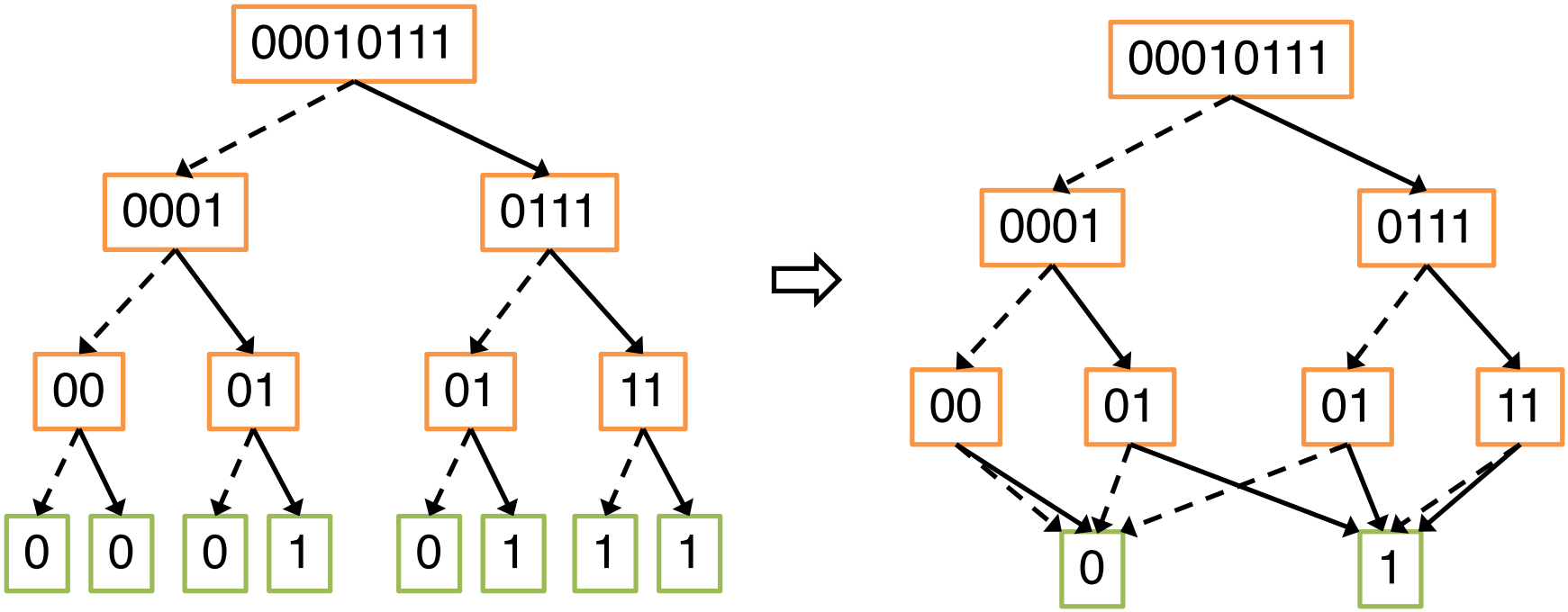
$= x' \bullet f(0, y, z) + x \bullet f(1, y, z)$

- Nonterminal node in orange
  - Follow dashed line for value 0
  - Follow solid line for value 1
- Terminal (leaf) node in green
  - Function value determined by leaf values

Reduction Rule #1

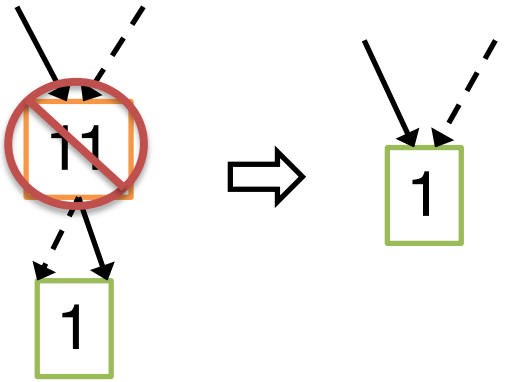
- Merge equivalent leaves

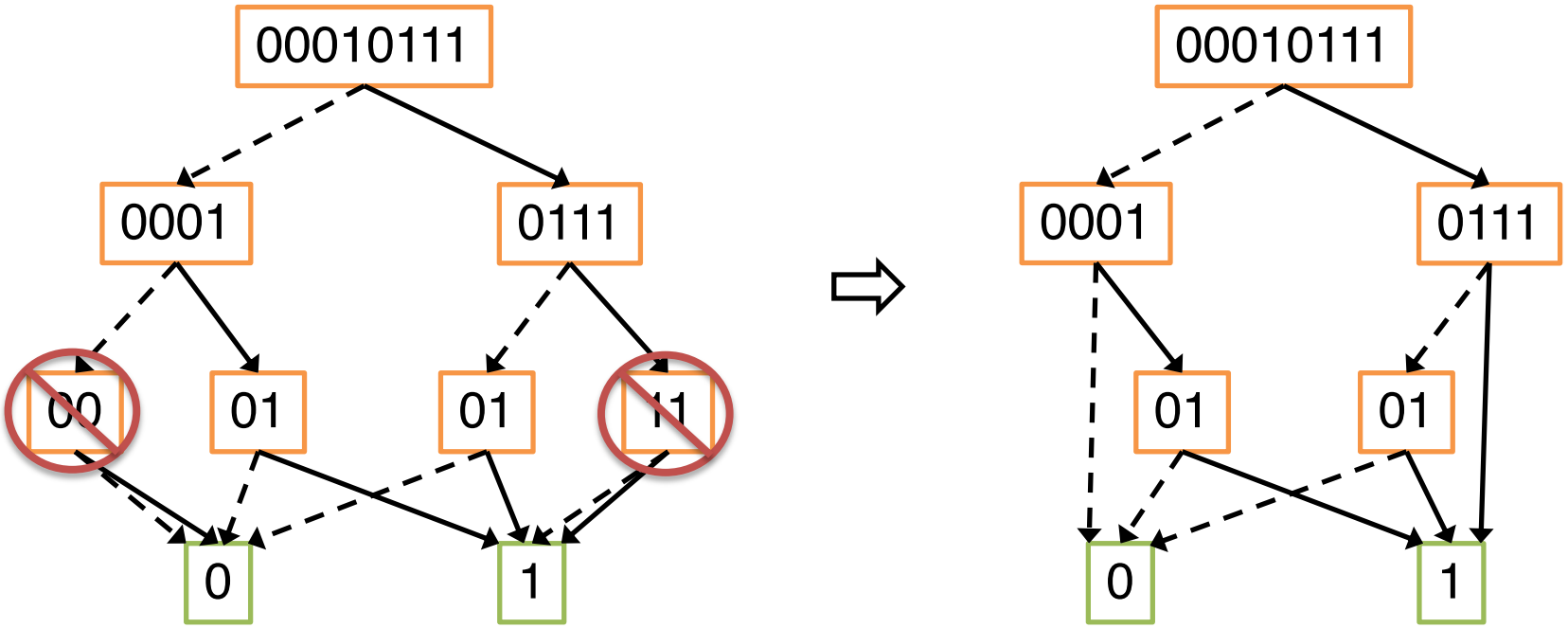




Reduction Rule #2

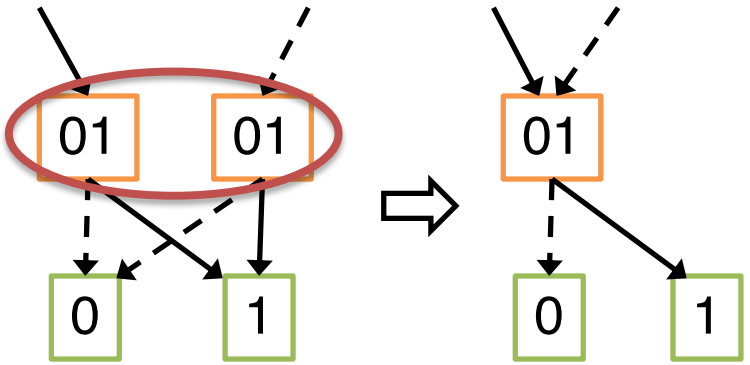
- Remove redundant tests
  - If a node v has the same left child as its right child, it's deemed redundant
    - i.e., left(v) = right(v)

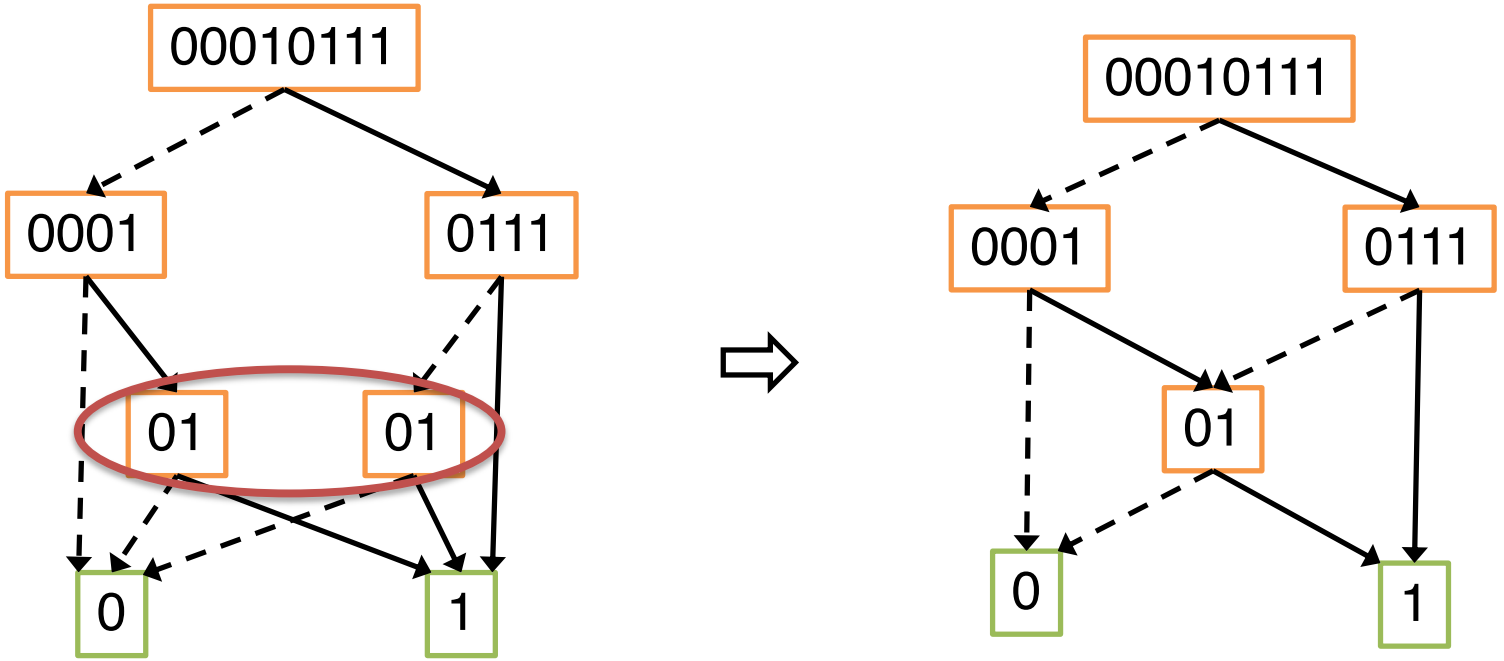




Reduction Rule #3

- Merge isomorphic nodes (i.e., nodes with the same structure)
  - u and v are isomorphic, when left(u) = left(v) and right(u) = right(v)





Efficient BDD Construction

- BDDs are usually directly constructed bottom up, avoiding the reduction steps

- One approach is using a hash table called unique table, which contains the IDs of the Boolean functions whose BDDs have been constructed [1]
  - A new function is added if its associated ID is not already in the unique table

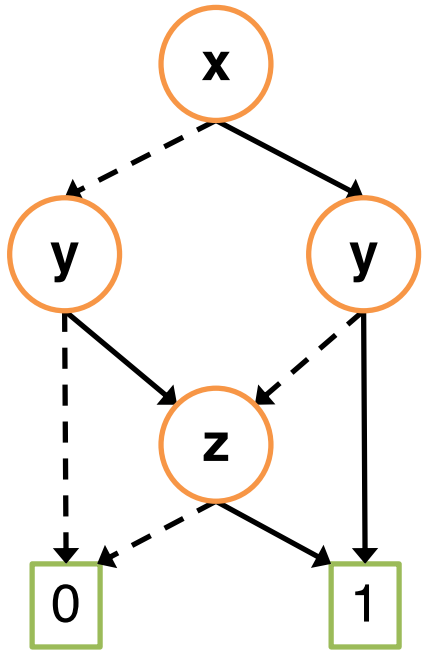
[1] K. Brace, R. Rudell, and R. Bryant, Efficient Implementation of a BDD Package, DAC'91.

BDDs History

- Initially proposed by Lee in 1959, and later Akers in 1976
  - Idea of representing Boolean function as a rooted DAG with a decision at each vertex
- Popularized by Bryant in 1986
  - Further restrictions + efficient algorithms to make a useful data structure (ROBDD)
  - **BDD = ROBDD since then**

ROBDDs

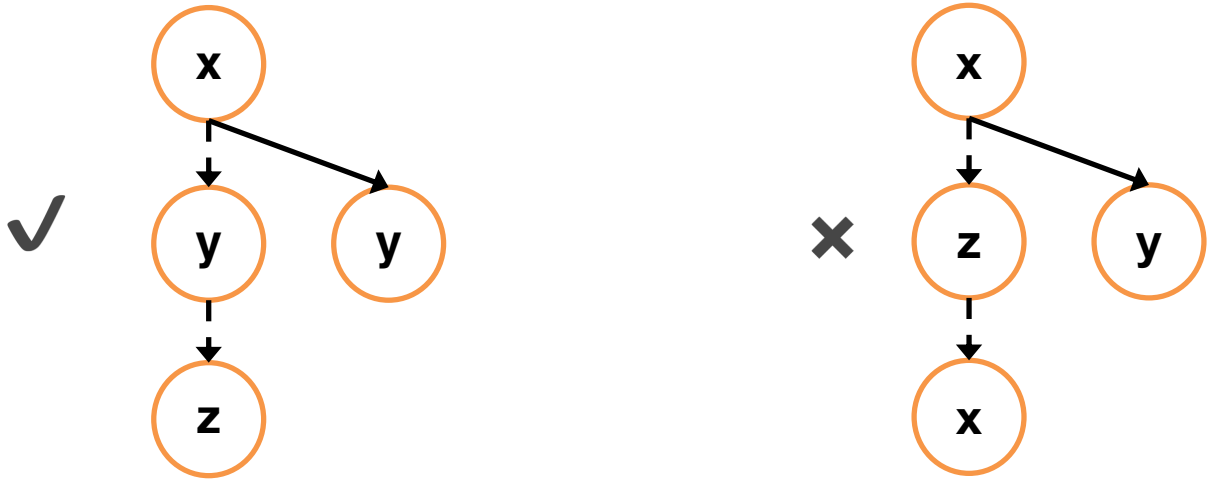
- Reduced and Ordered (**RO**BDD)
  - Directed acyclic graph (DAG)
    - Two children per node
    - Two terminals 0, 1
  - **Ordered**:
    - Co-factoring variables (splitting variables) always follow the same order along all paths $x_1 < x_2 < x_3 < \ldots < x_n$
  - **Reduced:**
    - Any node with two identical children is removed (rule #2)
    - Two nodes with isomorphic BDDs are merged (rules #1 and #3)



3-input voting function in BDD form

More on Variable Ordering

- Follow a total ordering to variables
  - e.g., x < y < z
- Variables must appear in the same ascending order along all paths

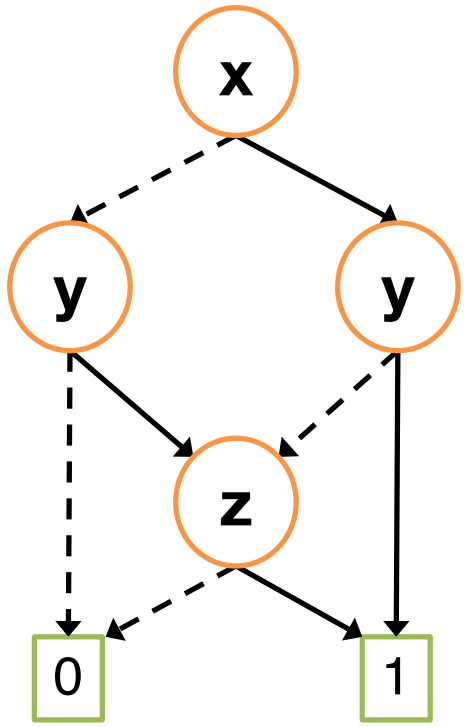


Canonical Representation

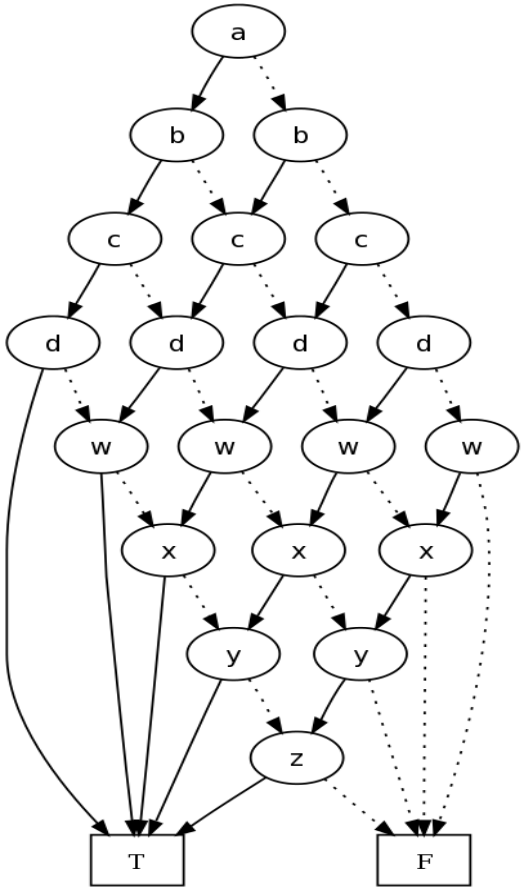
- BDD is a *canonical* representation of Boolean functions
  - Given the same variable order, two functions equivalent if and only if they have the same BDD form
    - “0” unique unsatisifable function
    - “1” unique tautology

More Virtues of BDDs

- There are many, but to list a few more:
  - Can represent an exponential number of paths with a DAG
  - Can evaluate an *n*-ary Boolean function in at most *n* steps
    - By tracing paths to the 1 node, we can count or enumerate all solutions to equation f = 1
  - Every BDD node (not just root) represent some Boolean function in a canonical way
    - A BDD can be multi-rooted representing multiple Boolean functions sharing subgraphs



BDD Representation of Voting Function



- 8-input voting function in BDD with only 20 nonterminal nodes
- In contrast to 70 prime implicants in SOP form

Diagram generated by www.cs.uc.edu/~weaversa/BDD_Visualizer.html

Example Application: Equivalence Checking

```
bool P(bool x, bool y) { return ~(~x & ~y); }
```

& means bitwise AND in C; ~ is negation

$P \stackrel{?}{=} Q$

Is P equivalent to Q?

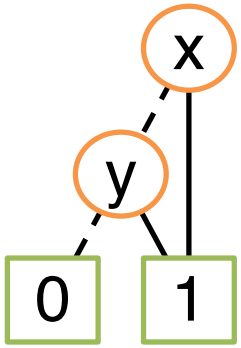
```
bool Q(bool x, bool y) { return x ^ y; }
```

^ means bitwise XOR in C

- Either prove equivalence or find counterexample(s)
  - Counterexamples: Input values (x, y) for which the two programs produce different results

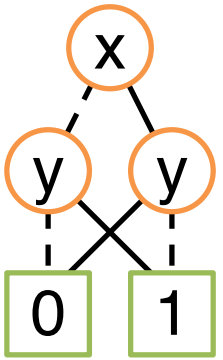
Equivalence Checking using BDDs

P = ~(~x & ~y)
= x | y



BDD of P

Q = x ^ y



BDD of Q

P $\overset{?}{=}$ Q

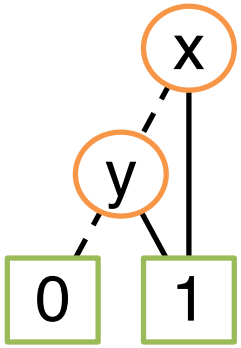
Checking equivalence of P and Q by constructing the BDD of (P==Q)

Let **S** denote the function of (P==Q), i.e., P XNOR Q

Exercise: first derive $S_{x=1}$ and $S_{x=0}$ before constructing the BDD of S

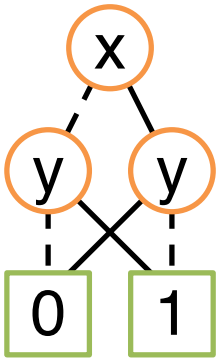
Equivalence Checking using BDDs

P = ~(~x & ~y)
= x | y



BDD of P

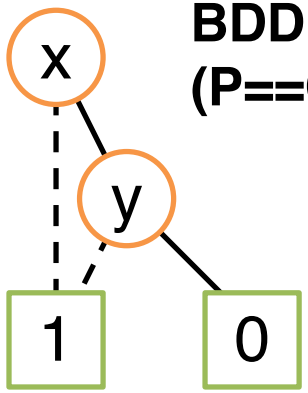
Q = x ^ y



BDD of Q

P $\stackrel{?}{=}$ Q

Checking equivalence of P and Q by constructing the BDD of (P==Q)



BDD of (P==Q)

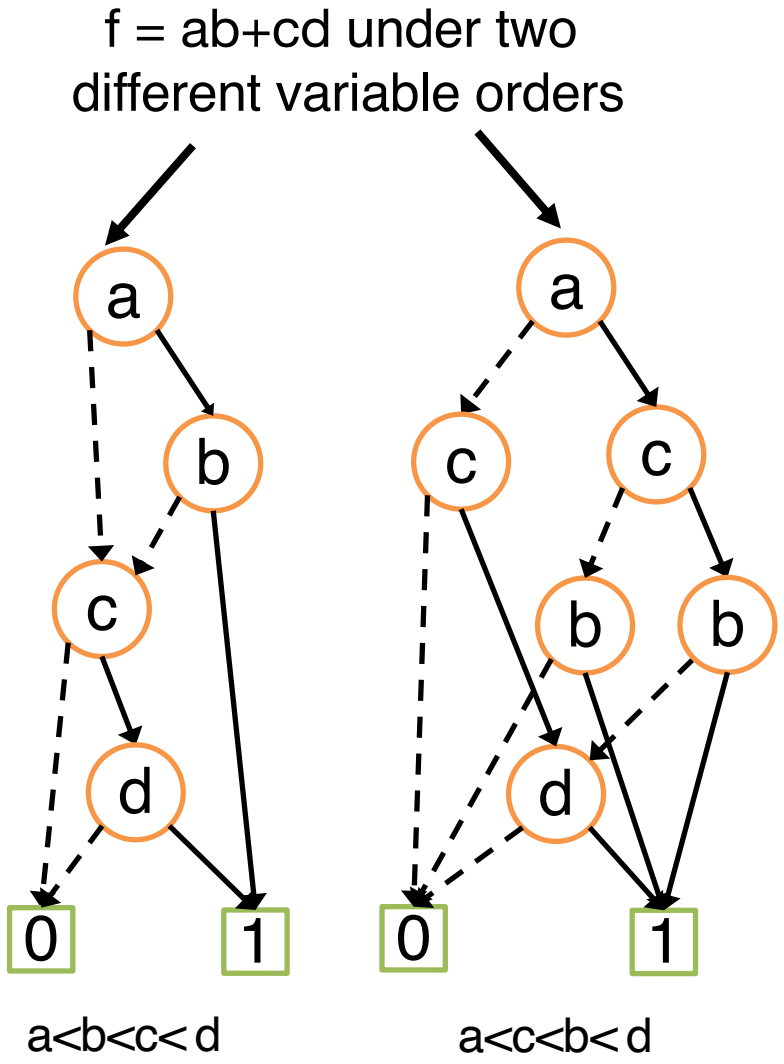
Counterexample

Setting x = 1 & y = 1 leads to a false output

Hence P ≠ Q

BDD Limitations

- NP-hard problem to construct the optimal order for a given BDD
  - Extensive research in ordering algorithms
- No efficient BDD exists for some functions regardless of the order
- Existing heuristics work well enough on many combinational functions from real circuits



Same function, two different orderings, different graphs

Next Lecture

- Front-end compilation and CDFG

Acknowledgements

- These slides contain/adapt materials from / developed by
  - Prof. Randal Bryant (CMU)