EECS 219C: Computer-Aided Verification

Binary Decision Diagrams (BDDs)

Sanjit A. Seshia

EECS, UC Berkeley

Boolean Function Representations

- Syntactic: e.g.: CNF, DNF (SOP), Circuit
- Semantic: e.g.: Truth table, Binary Decision Tree, BDD

Reduced Ordered BDDs

- Introduced by Randal E. Bryant in mid-80s
  - IEEE Transactions on Computers 1986 paper is one of the most highly cited papers in EECS
- Useful data structure to represent Boolean functions
  - Applications in logic synthesis, verification, program analysis, AI planning, …
- Commonly known simply as BDDs
  - Lee [1959] and Akers [1978] also presented BDDs, but not **RO**BDDs
- Many variants of BDDs have also proved useful
- Links to coding theory (trellises), etc.

RoadMap for this Lecture

- Cofactor of a Boolean function
- From truth table to BDD
- Properties of BDDs
- Operating on BDDs
- Variants

Cofactors

- A Boolean function F of n variables $x_1, x_2, \ldots, x_n$

$$F : \{0,1\}^n \rightarrow \{0,1\}$$

- Suppose we define new Boolean functions of n-1 variables as follows:

$$F_{x_1}(x_2, \ldots, x_n) = F(1, x_2, x_3, \ldots, x_n)$$
$$F_{x_1'}(x_2, \ldots, x_n) = F(0, x_2, x_3, \ldots, x_n)$$

- F_{x_i} and $F_{x_i'}$ are called cofactors of F. F_{x_i} is the positive cofactor, and $F_{x_i'}$ is the negative cofactor

Shannon Expansion

- $F(x_1, \ldots, x_n) = x_i \,.\, F_{x_i} + x_i' \,.\, F_{x_i'}$

- Proof?

Shannon expansion with many variables

- F(x, y, z, w) =

  $xy\ F_{xy} + x'y\ F_{x'y} + xy'\ F_{xy'} + x'y'\ F_{x'y'}$

Properties of Cofactors

- Suppose you construct a new function H from two existing functions F and G: e.g.,
  - H = F’
  - H = F.G
  - H = F + G
  - Etc.

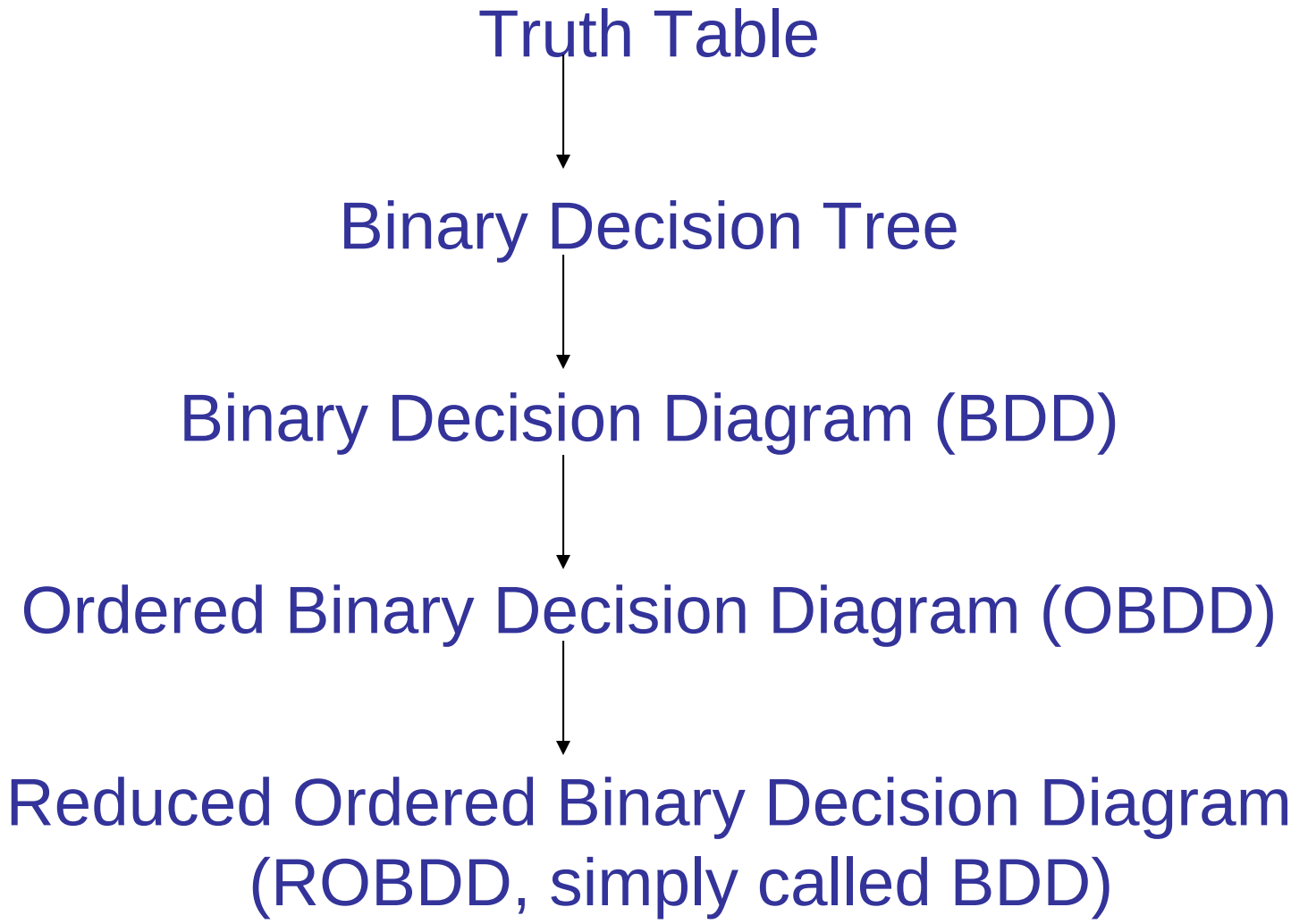
- What is the relation between cofactors of H and those of F and G?

Very Useful Property

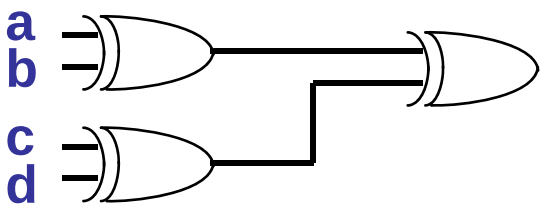
- Cofactor of NOT is NOT of cofactors
- Cofactor of AND is AND of cofactors
- …

- Works for any binary operator

BDDs from Truth Tables



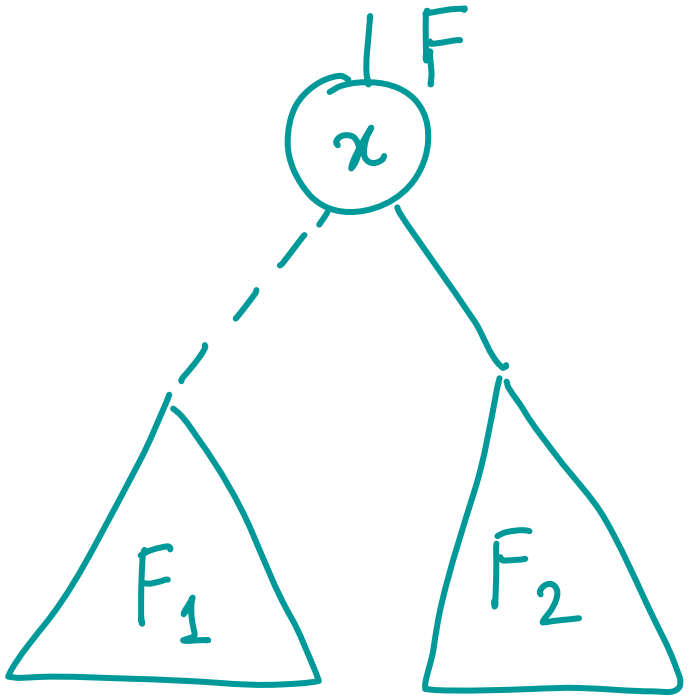
Example: Odd Parity Function



Edge labels along a root-leaf path form an assignment to a,b,c,d

Binary Decision Tree

Nodes & Edges

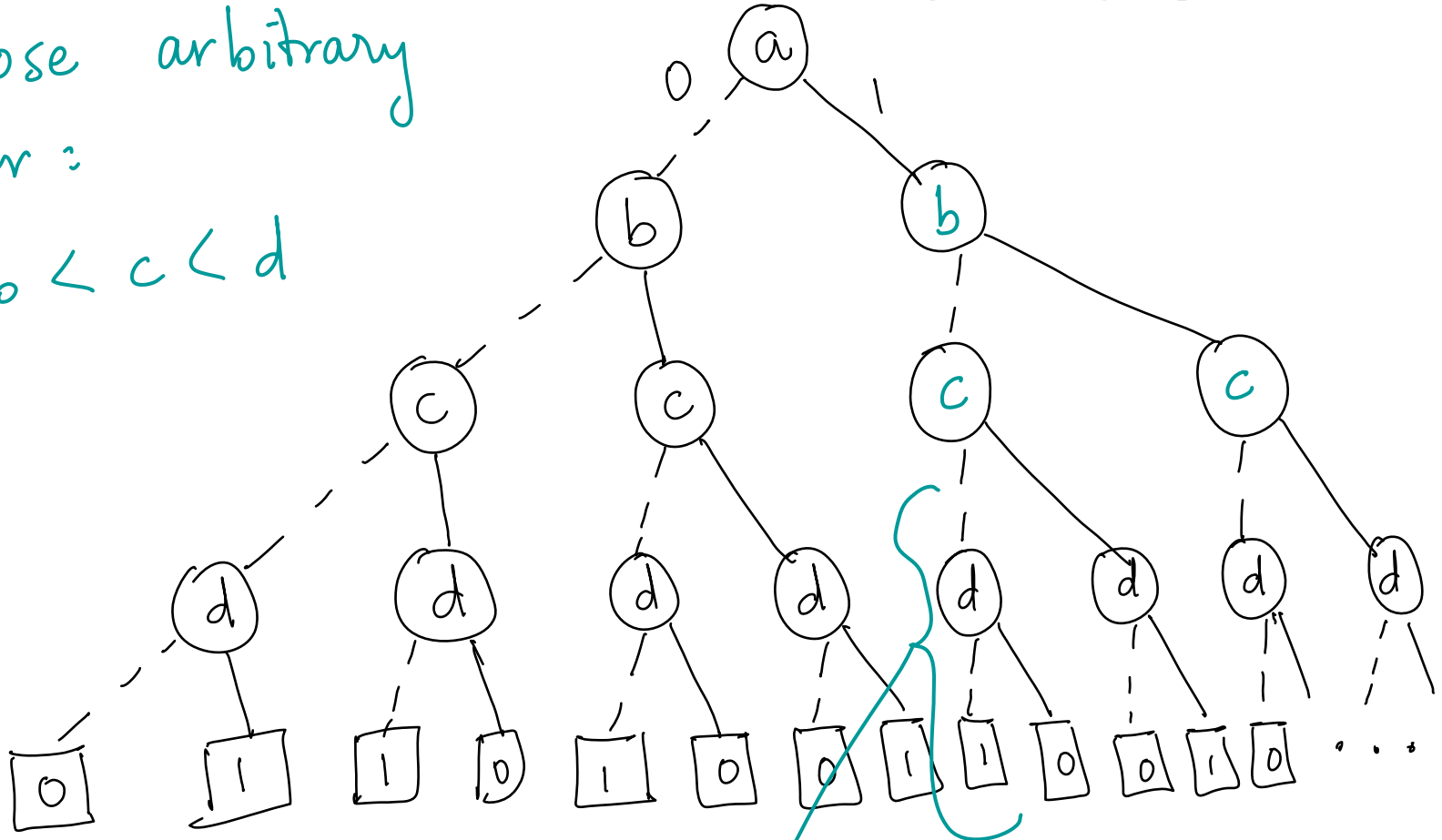


- How is F related to x, F_1, F_2 ?

Ordering: variables appear in same order from root to leaf along any path

Impose arbitrary order:

$a < b < c < d$



Each node is some cofactor of the function

Why didn't this part change?

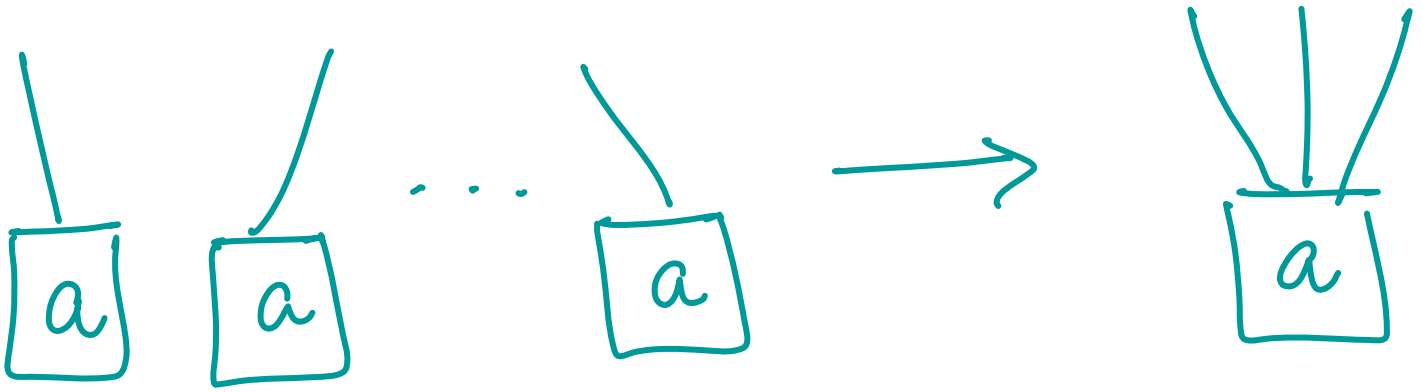
Reduction

- Identify Redundancies

- 3 Rules:

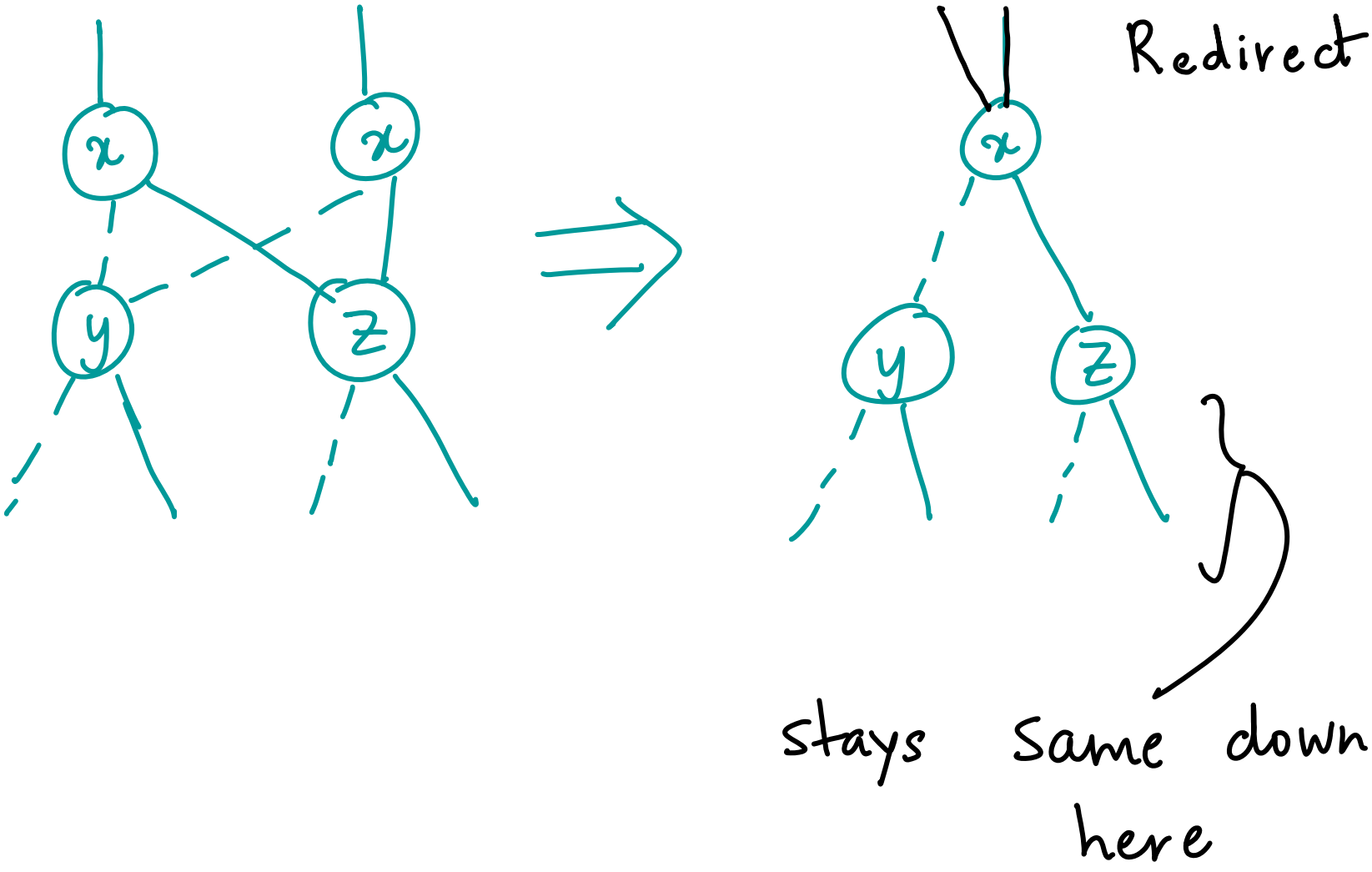
1. Merge equivalent leaves
2. Merge isomorphic nodes
3. Eliminate redundant tests

Merge Equivalent Leaves

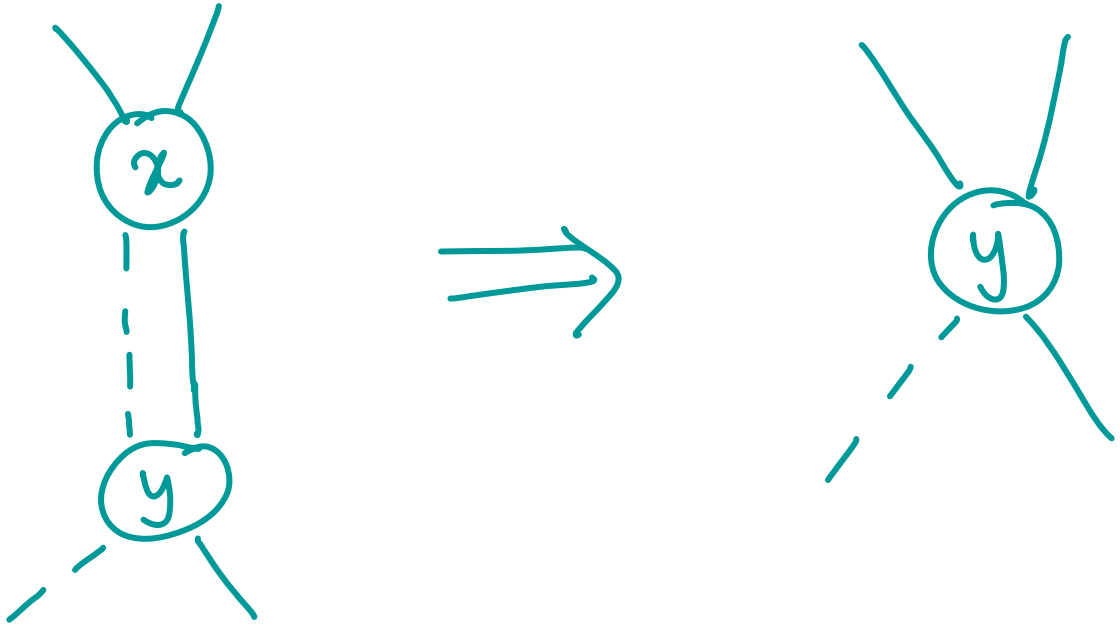


"a" is either 0 or 1

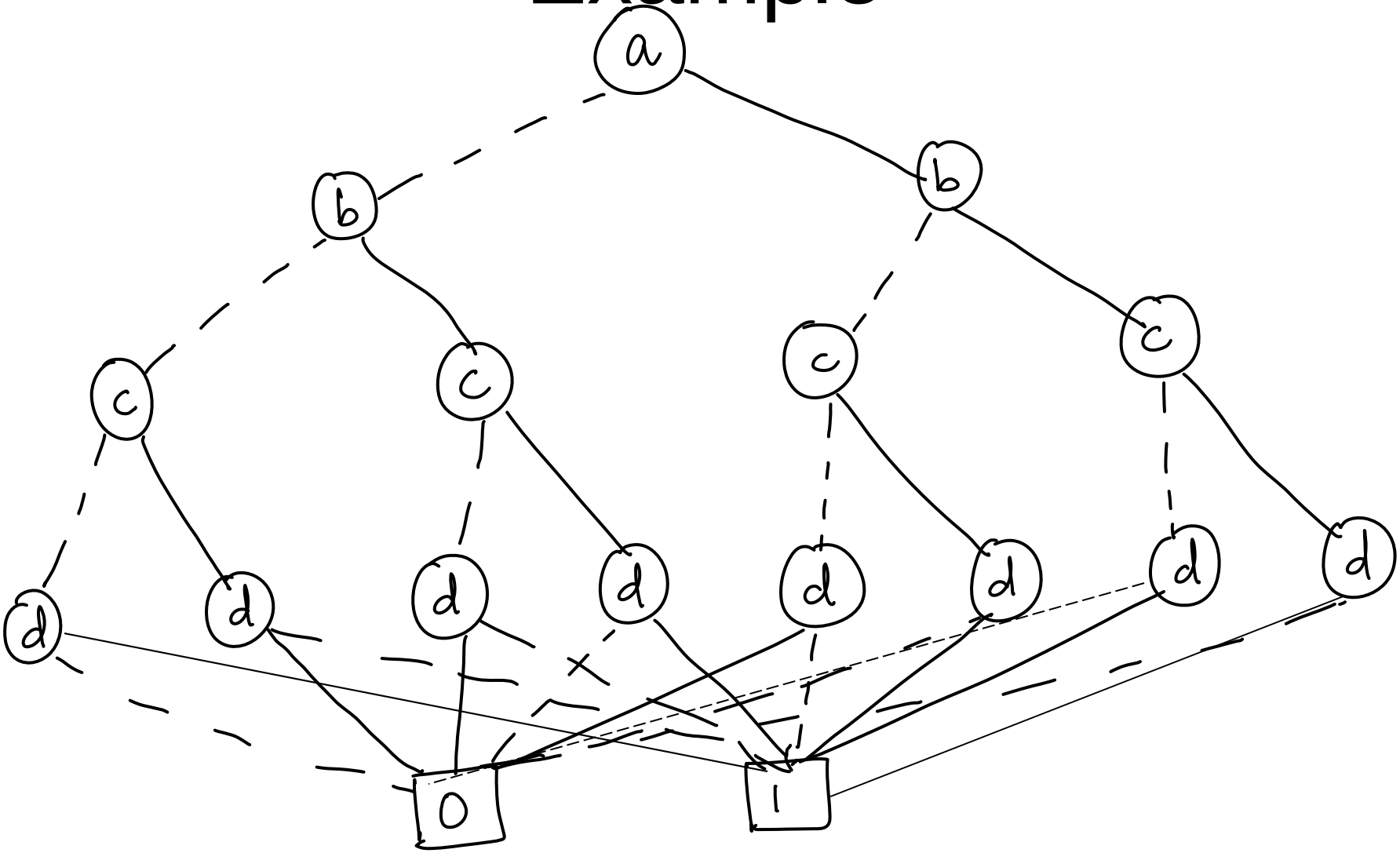
Merge Isomorphic Nodes



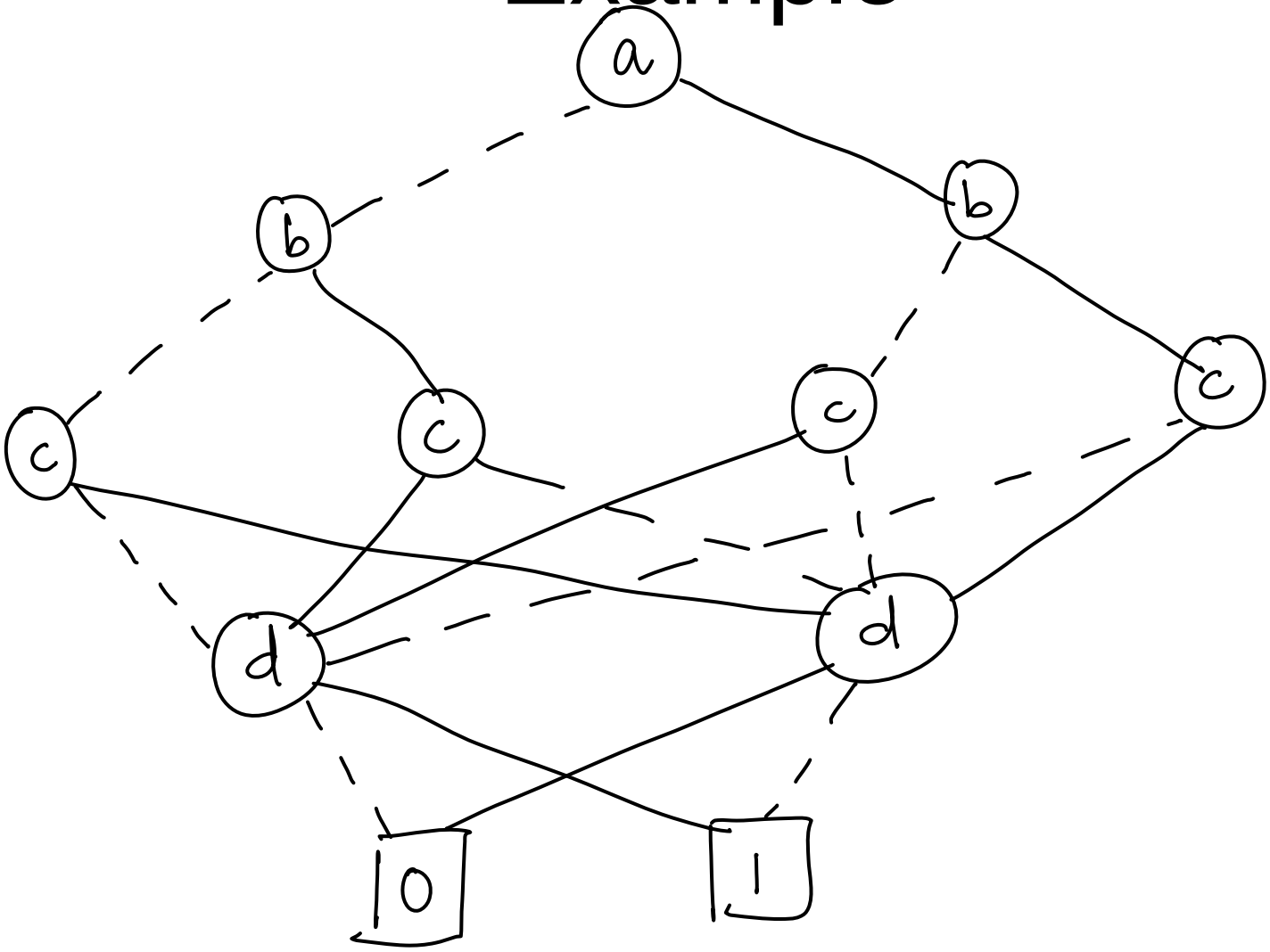
Eliminate Redundant Tests



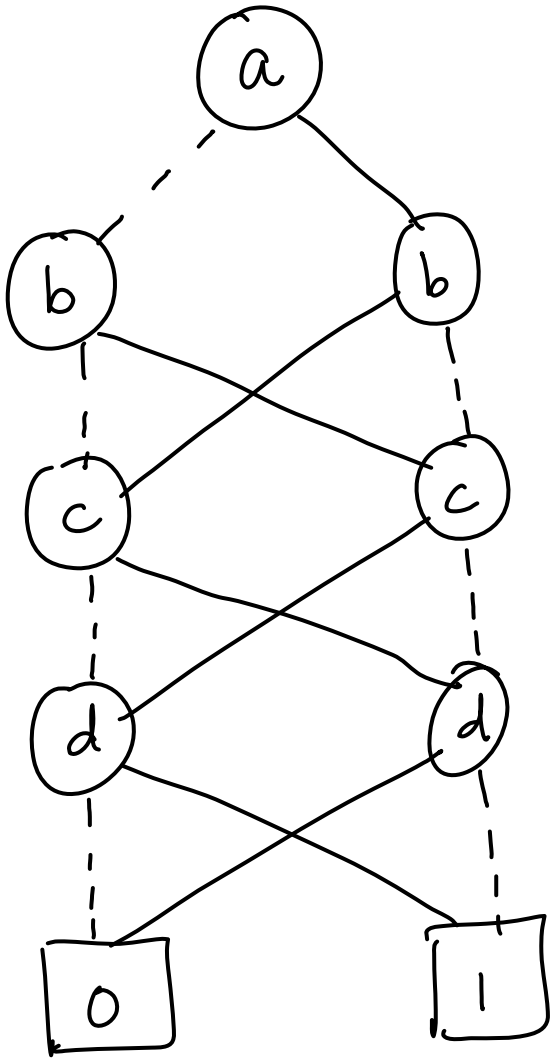
Example



Example

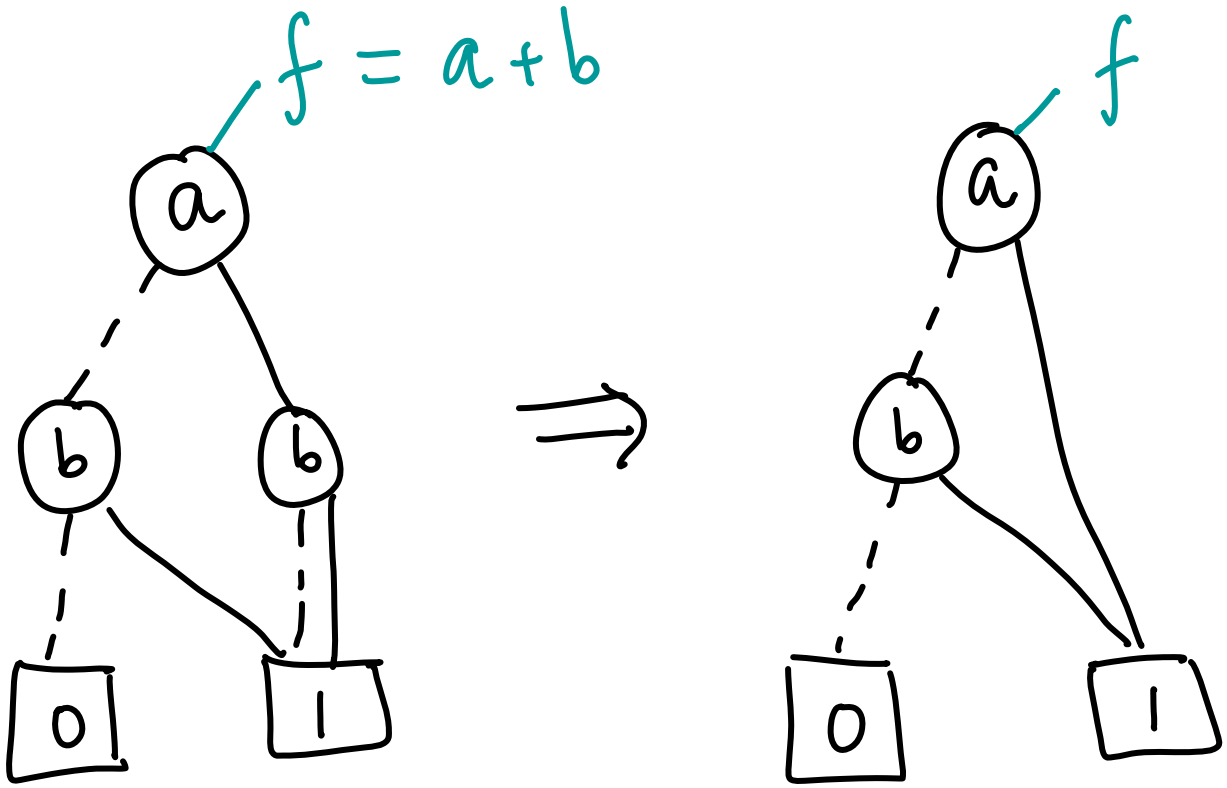


Final ROBDD for Odd Parity Function



2n-1 non-terminal nodes

Example of Rule 3



What can BDDs be used for?

- Uniquely representing a Boolean function
  - And a Boolean function can represent sets
- Symbolic simulation of a combinational (or sequential) circuit
- Equivalence checking and verification
  - Satisfiability (SAT) solving
- Finding / counting *all* solutions to a SAT (combinatorial) problem
- Operations on “quantified” Boolean formulas

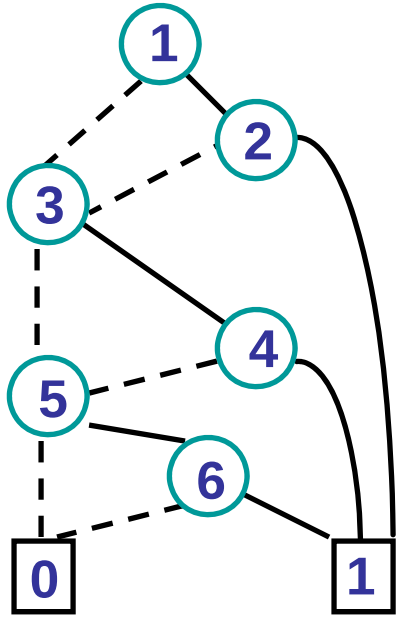
(RO)BDDs are canonical

- Theorem (R. Bryant): If G, G’ are ROBDD’s of a Boolean function f with k inputs, using same variable ordering, then G and G’ are identical.

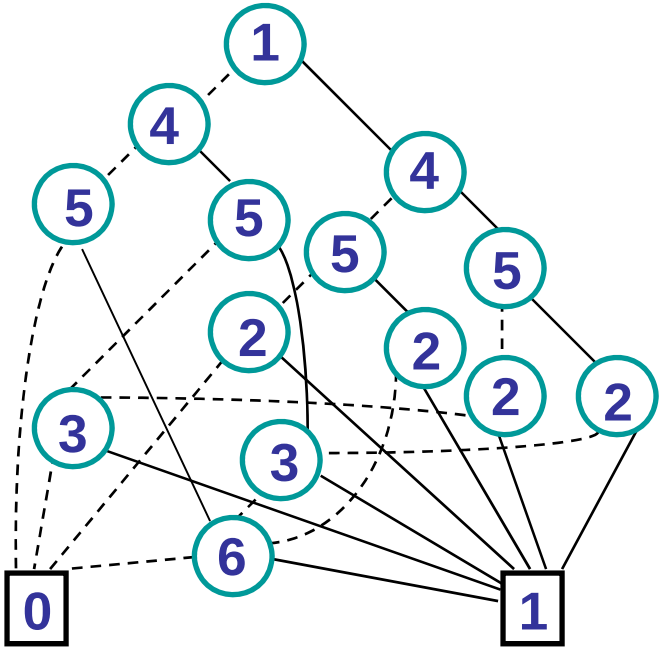
Sensitivity to Ordering

- Given a function with n inputs, one input ordering may require exponential # vertices in ROBDD, while other may be linear in size.
- Example: $f = x_1 x_2 + x_3 x_4 + x_5 x_6$

$x_1 < x_2 < x_3 < x_4 < x_5 < x_6$



$x_1 < x_4 < x_5 < x_2 < x_3 < x_6$



Constructing BDDs in Practice

- Strategy: Define how to perform basic Boolean operations
- Build a few core operators and define everything else in terms of those

Advantage:
- **Less programming work**
- **Easier to add new operators later by writing “wrappers”**

Core Operators

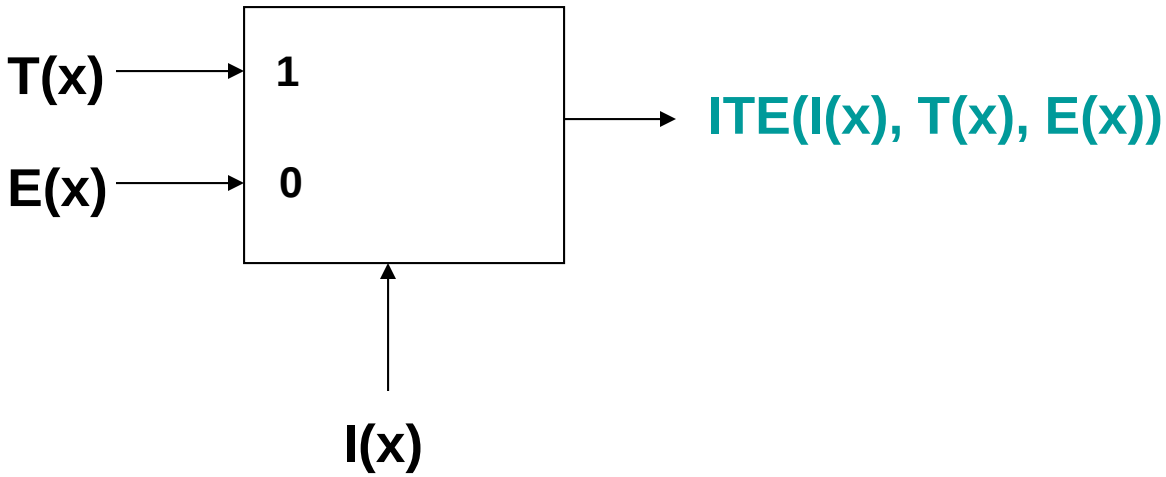
- Just two of them!

1. Restrict(Function F, variable v, constant k)
   - Shannon cofactor of F w.r.t. v=k

2. ITE(Function I, Function T, Function E)
   - “if-then-else” operator

ITE

- Just like:
  - "if then else" in a programming language
  - A mux in hardware
- ITE(I(x), T(x), E(x))
  - If I(x) then T(x) else E(x)



The ITE Function

- ITE(I(x), T(x), E(x))
- =
- I(x) . T(x) + I’(x). E(x)

What good is the ITE?

- How do we express
- NOT?

- OR?

- AND?

How do we implement ITE?

- Divide and conquer!

- Use Shannon cofactoring…
- Recall: Operator of cofactors is Cofactor of operators…

ITE Algorithm

```
ITE (bdd I, bdd T, bdd E) {
   if (terminal case) { return computed result; }
   else { // general case
      Let x be the topmost variable of I, T, E;
      PosFactor = ITE(I_x , T_x , E_x) ;
      NegFactor = ITE(I_x' , T_x' , E_x');
      R = new node labeled by x;
      R.low = NegFactor; // R.low is 0-child of R
      R.high = PosFactor; // R.high is 1-child of R
      Reduce(R);
      return R;
}
```

Terminal Cases (complete these)

- ITE(1, T, E) =
- ITE(0, T, E) =
- ITE(I, T, T) =
- ITE(I, 1, 0) =
- …

General Case

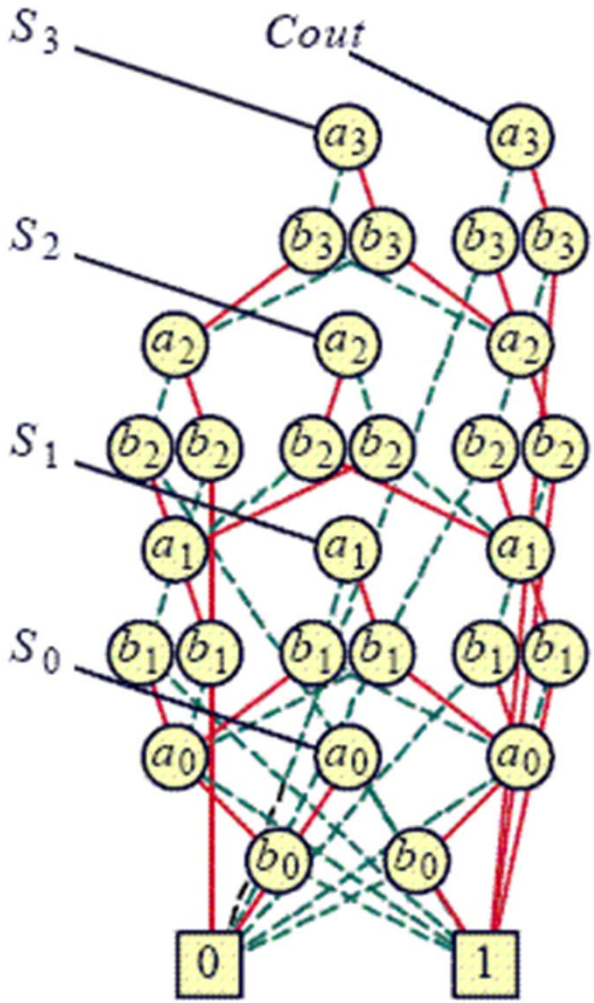
- Still need to do cofactor (Restrict)
- How hard is that?
  - Which variable are we cofactoring out? (2 cases)

Practical Issues

- Previous calls to ITE are cached
  - "memoization"
- Every BDD node created goes into a "unique table"
  - Before creating a new node R, look up this table
  - Avoids need for reduction

Sharing: Multi-Rooted DAG

- BDD for 4-bit adder: 5 output bits → 5 Boolean functions
- Each output bit (of the sum & carry) is a distinct rooted BDD
- But they share sub-DAGs



More on BDDs

- Circuit width and bounds on BDD size (reading exercise – slide summary posted)
- Dynamically changing variable ordering
- Some BDD variants

Bounds on BDD Size: Warm-up

- Suppose the number of nodes at any level in a BDD is bounded above by B
- Then, what is an upper bound on the total number of nodes in the BDD?

Cross-section of a BDD at level i

- Suppose a BDD represents Boolean function $F(x_1, x_2, \ldots, x_n)$ with variable order $x_1 < x_2 < \ldots < x_n$
- Size of cross section of the BDD at level i is the number of distinct Boolean functions F' that *depend on* x_i given by

  $F'(x_i, x_{i+1}, \ldots, x_n) = F(v_1, v_2, \ldots, v_{i-1}, x_i, \ldots, x_n)$

  for some Boolean constants v_i's (in {0,1})

Circuit Width

- Consider a circuit representation of a Boolean function F
- Impose a linear order on the gates of the circuit
  - Primary inputs and outputs are also considered as "gates" and primary output is at the end of the ordering
  - Forward cross section at a gate g: set of wires going from output of g_1 to input of g_2 where $g_1 \cdot g < g_2$
  - Similarly define reverse cross section: set of wires going from output of g_1 to input of g_2 where $g_2 \cdot g < g_1$
  - Forward width (w_f): maximum forward cross section size
  - Similarly, reverse width w_r

BDD Upper Bounds from Circuit Widths

- Theorem: Let a circuit representing F with n variables have forward width w_f and reverse width w_r for some linear order L on its gates. Then, there is a BDD representing F of size bounded above by

$$n.2^{w_f \cdot 2^{w_r}}$$

BDD Ordering in Practice

- If we can derive a small upper bound using circuit width, then that's fine
  - Use the corresponding linear order on the variables
- What if we can't?

- There are many BDD variable ordering heuristics around, but the most common way to deal with variable ordering is to start with something "reasonable" and then swap variables around to improve BDD size
  - DYNAMIC VARIABLE REORDERING → SIFTING

Sifting

- Dynamic variable re-ordering, proposed by R. Rudell
- Based on a primitive “swap” operation that interchanges x_i and x_{i+1} in the variable order
  - Key point: the swap is a local operation involving only levels i and i+1
- Overall idea: pick a variable x_i and move it up and down the order using swaps until the process no longer improves the size
  - A “hill climbing” strategy

Some BDD Variants

- Free BDDs (FBDDs)
  - Relax the restriction that variables have to appear in the same order along all paths
  - How can this help? → smaller BDD
  - Is it canonical? → NO

Some BDD Variants

- MTBDD (Multi-Terminal BDD)
  - Terminal (leaf) values are not just 0 or 1, but some finite set of numerical values
  - Represents function of Boolean variables with non-Boolean value (integer, rational)
    - E.g., input-dependent delay in a circuit, transition probabilities in a Markov chain
  - Similar reduction / construction rules to BDDs

Some BDD packages

- CUDD – from Colorado University, Fabio Somenzi's group
  - PerlDD front-end to CUDD
- BuDDy – from IT Univ. of Copenhagen

Reading

- Bryant's 1992 survey paper is required reading (posted on bCourses)
- Optional reading: Don Knuth's chapter on BDDs (posted on bCourses)