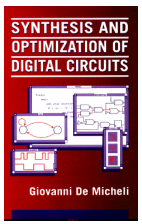
Binary Decision Diagrams

Giovanni De Micheli

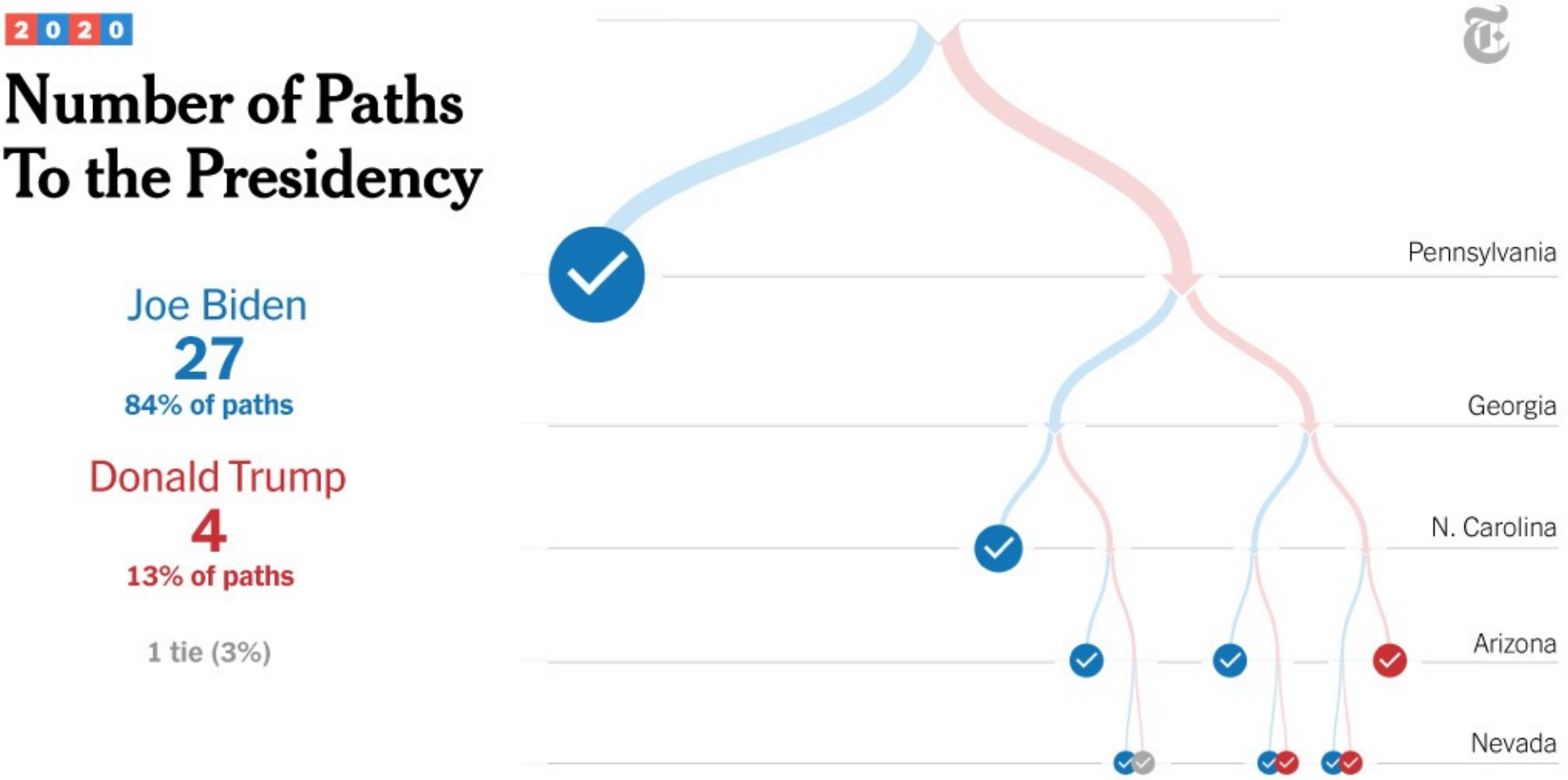
Integrated Systems Laboratory



Module 1

- **Objectives:**
  - Definitions of BDDs, OBDDs and ROBDDs
  - Logic operations on BDDs
  - The ITE operator

2020 US election – while counting one day after



Motivation

- Efficient way to represent logic functions
- History
  - Original idea for BDD due to Lee (1959) and Akers (1978)
  - Refined, formalized and popularized by Bryant (1986)
    - Smaller memory footprint
    - Canonical form – each distinct function correspond to a unique distinct diagram

Canonical forms - review

- Each logic function has a unique representation
- Truth table

a	b	c	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

- Sum of minterms

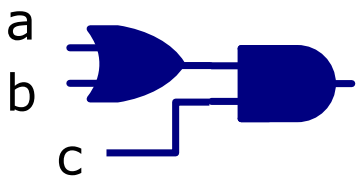
a’bc+ab’c+abc

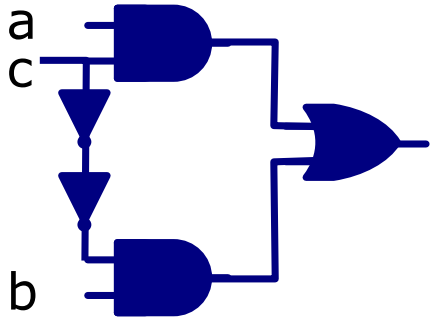
Non canonical forms - review

- Each function has also multiple representations
- Factored form

(a+b)c ac+bc

- Logic network representation





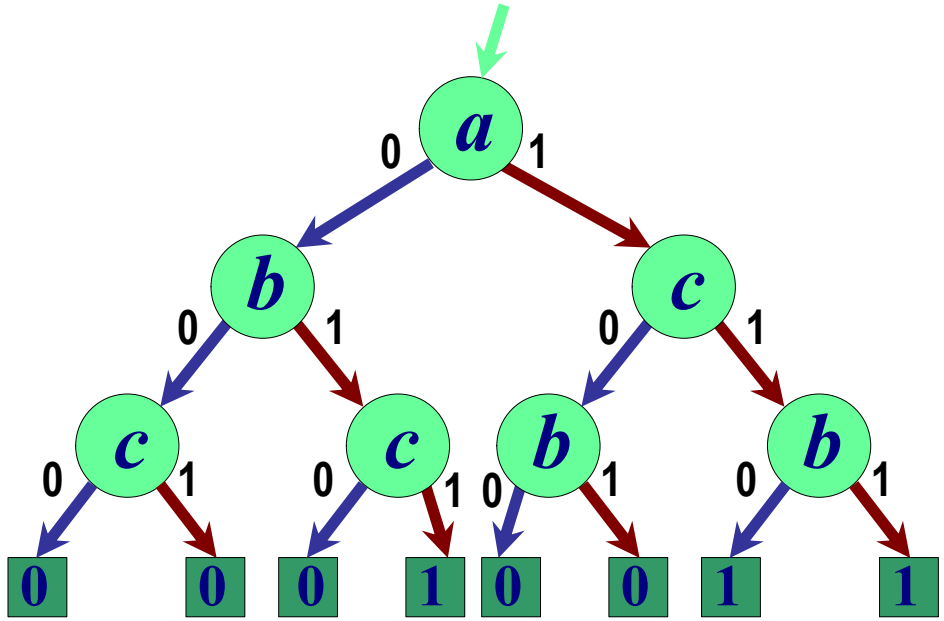
Terminology

- A Binary Decision Diagram (BDD) is a *directed acyclic graph*
  - **Graph**: set of vertices connected by edges
  - **Directed**: edges have direction
  - **Acyclic**: no path in the graph can lead to a cycle
- Often abbreviated as DAG
  - Simplest model:
    - Two leaves (Boolean constants 0 and 1)
    - One root
    - Can degenerate to a tree

BDD - Example

- $F = (a + b)\, c$

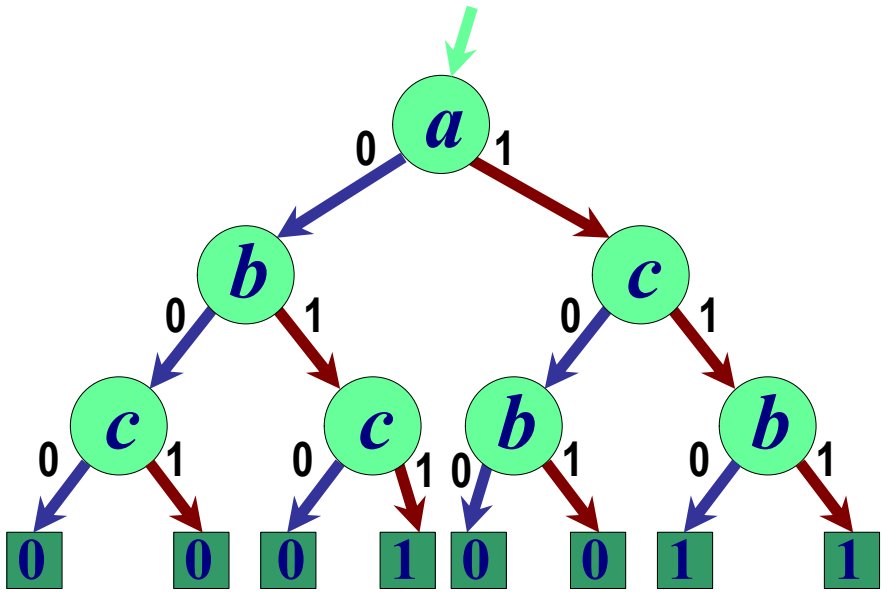
a	b	c	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1



1. Each vertex represents a decision on a variable
2. The value of the function is found at the leaves
3. Each path from root to leaf corresponds to a row in the truth table

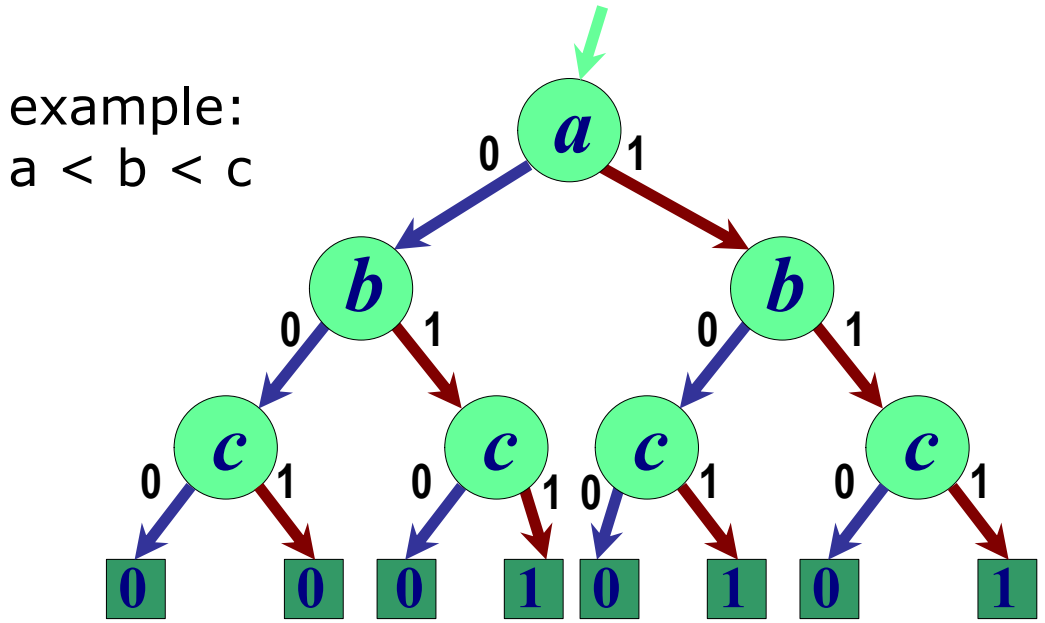
BDD - observations

- The size of a BDD is as big as a truth table:
  - 1 leaf per row
  - Exponential size
- Each path *from root to leaf* evaluates variables in some order
  - But the order is not fixed:
  - (a,b,c) and (a,c,b)
  - Free BDD



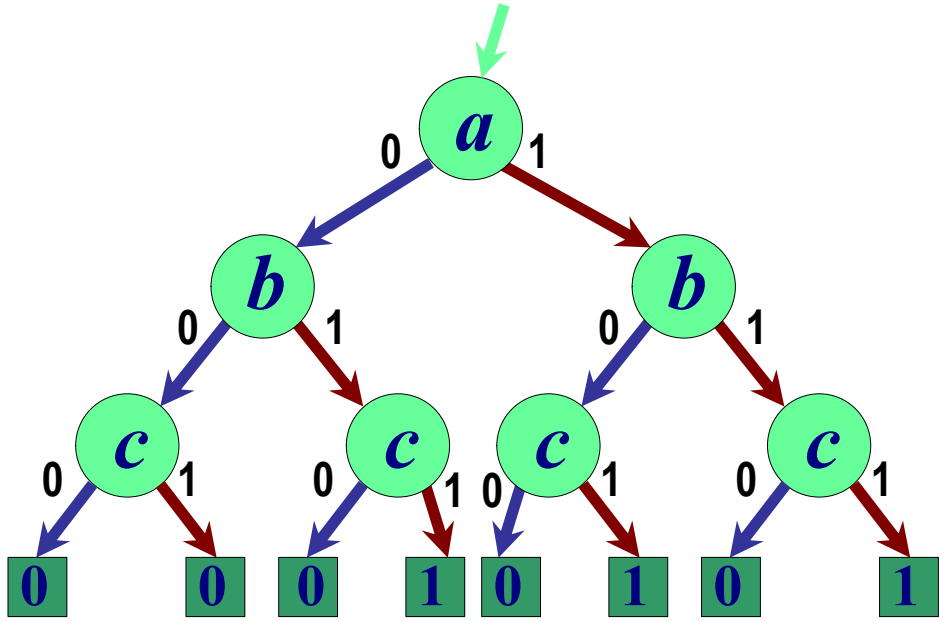
1st idea: Ordered BDD (OBDD)

- Choose arbitrary total ordering on the variables
  - Variables must appear in the same order along each path from root to leaves
  - Each variable can appear at most once on a path

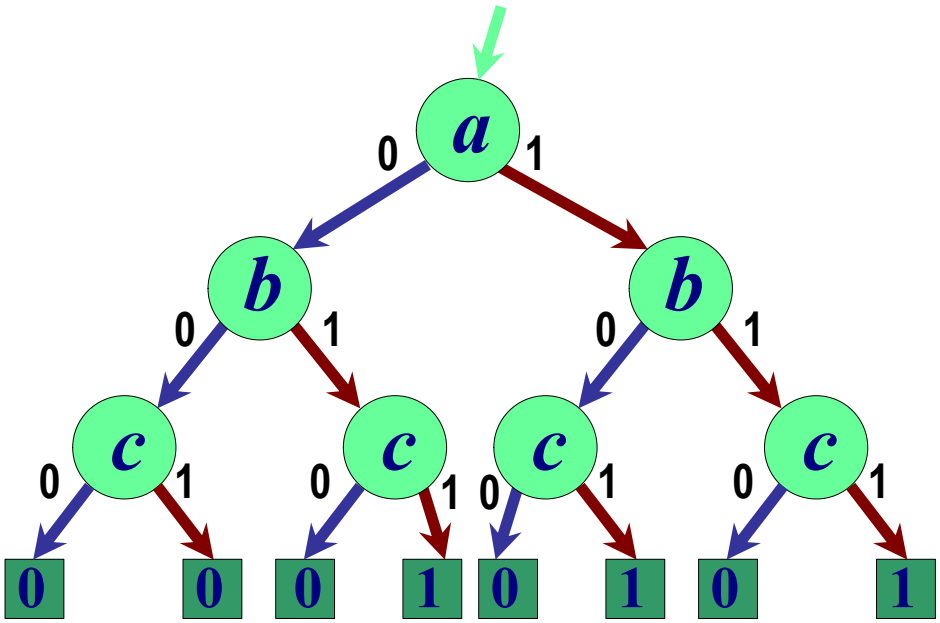


2nd idea: Reduced OBDD (ROBDD)

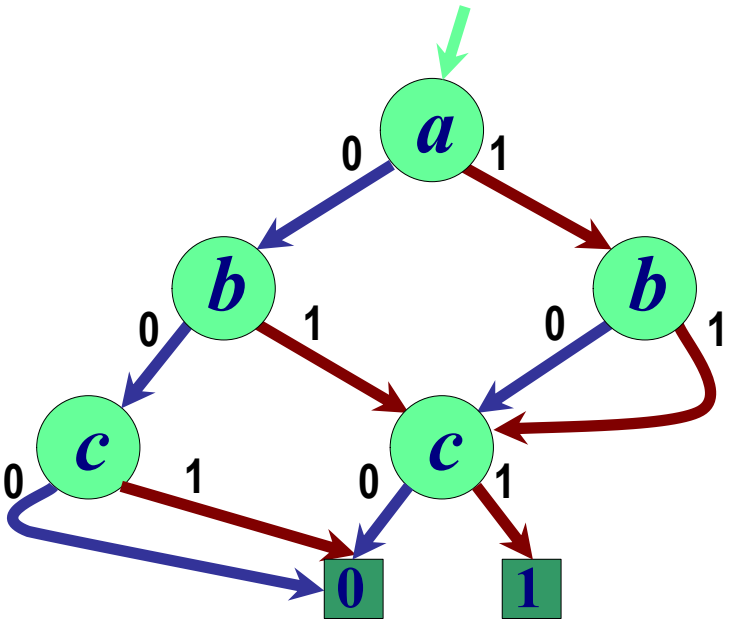
- **Two reduction rules:**
  1. **Merge equivalent sub-trees**
  2. **Remove nodes with identical children**



1. Merge equivalent sub-trees

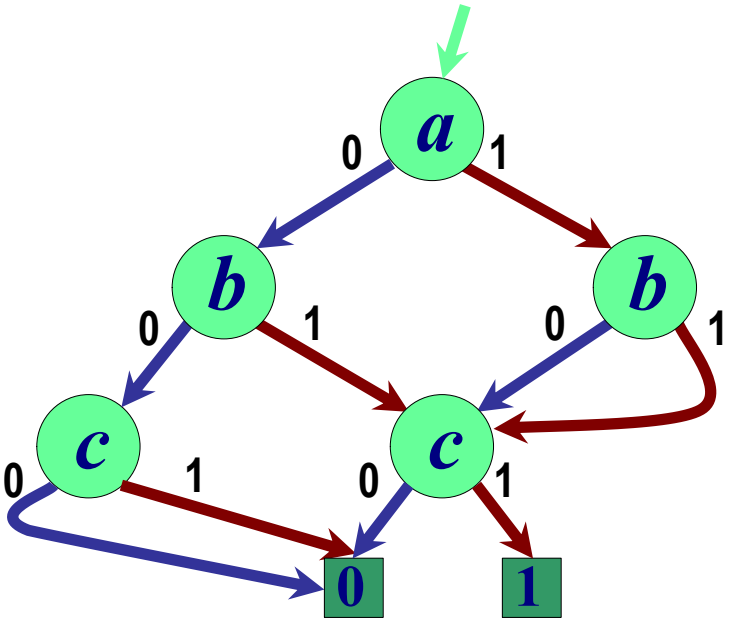


before

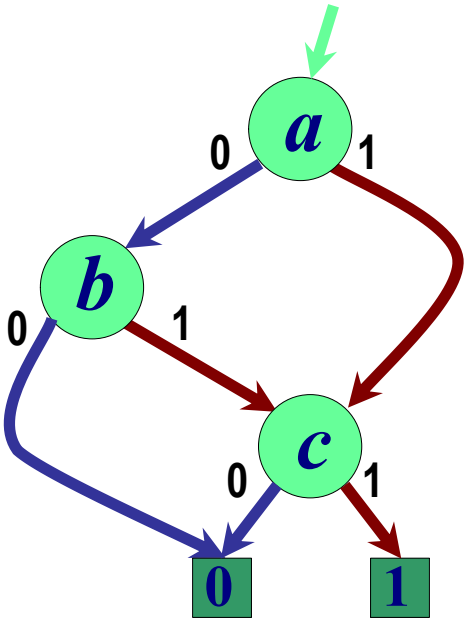


after

2. Remove node with identical children



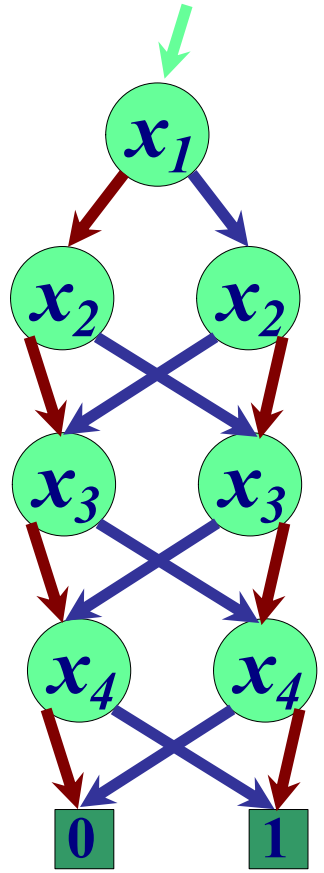
before



after

ROBDDs

- ROBDDs are canonical
  - For a given variable order
- ROBDD are more compact than other canonical forms
  - Efficient representation
- ROBDD size depends on the variable order
  - Many useful functions have linear-space (or slightly above) representation

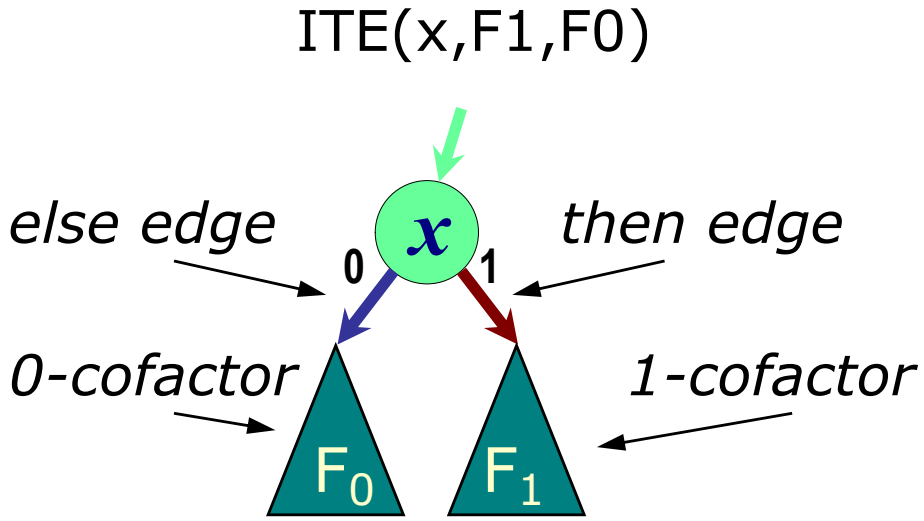


$$F = x_1 \oplus x_2 \oplus x_3 \oplus x_4$$

BDD semantics

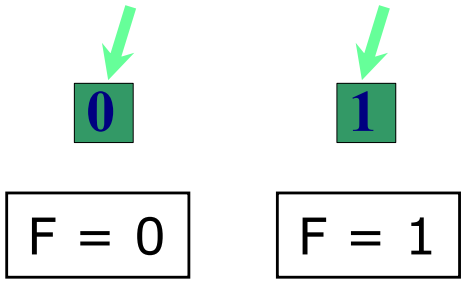
Constant nodes

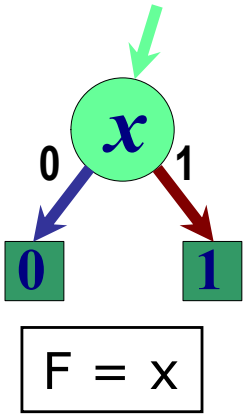
0 **1**

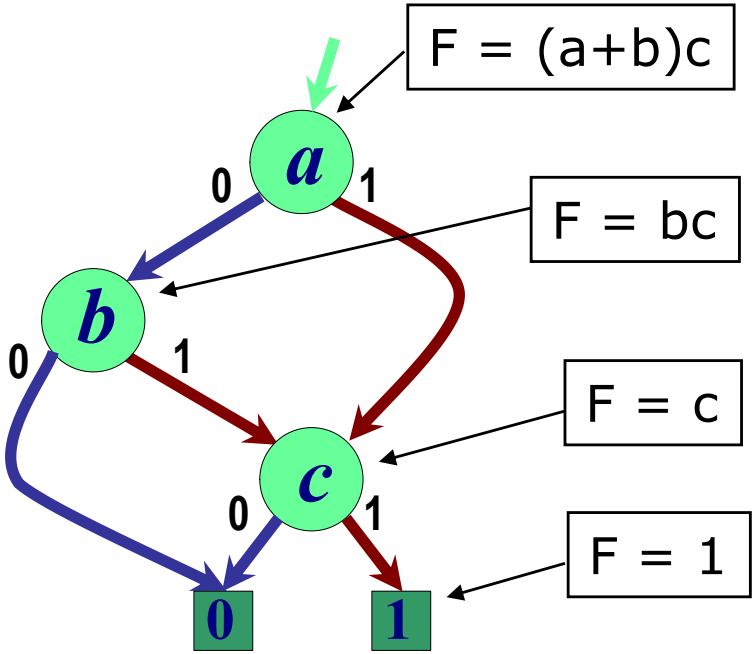


Cofactor(F,x): the function you obtain when you substitute 1 for x in F

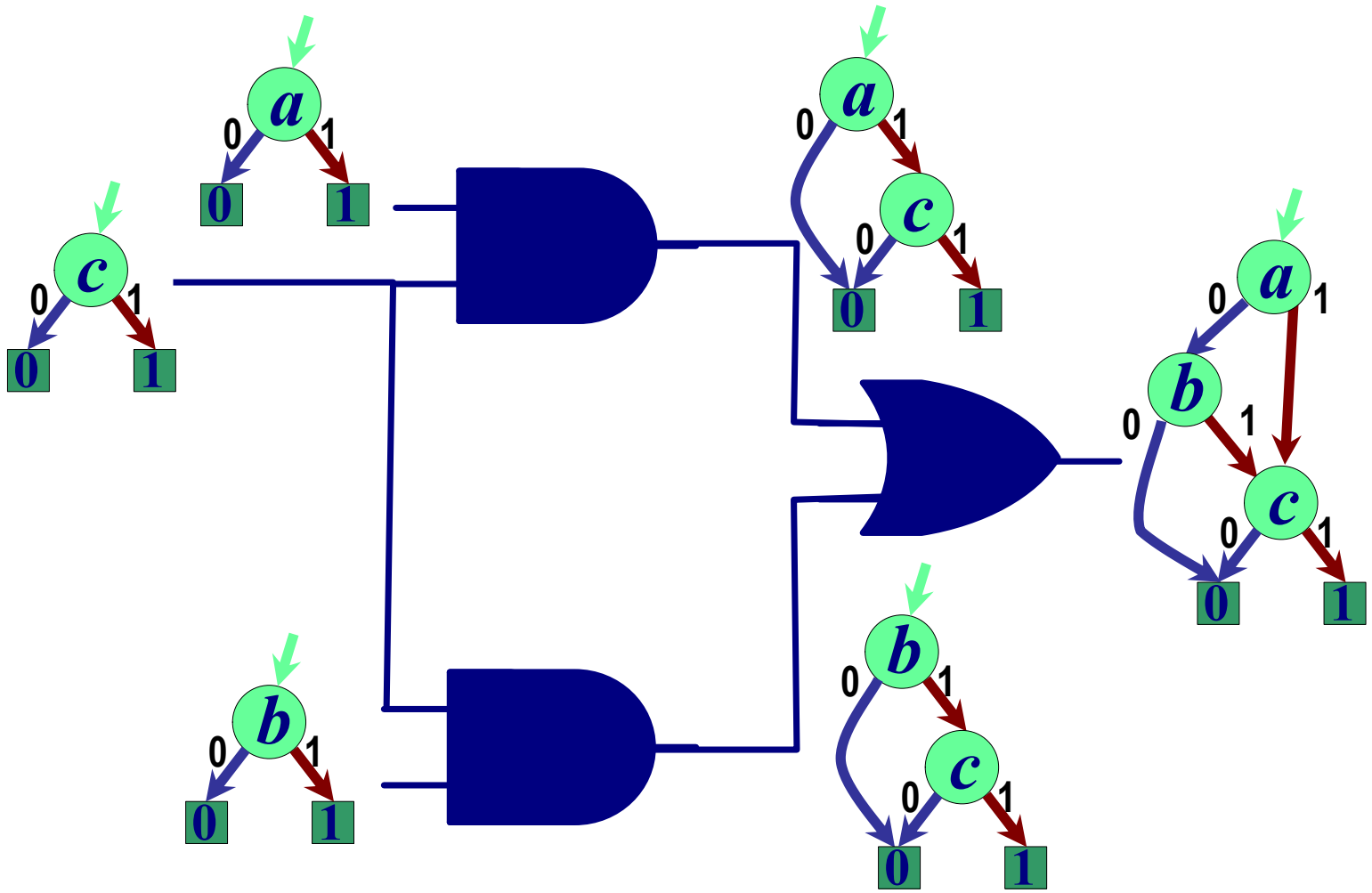
A few simple functions







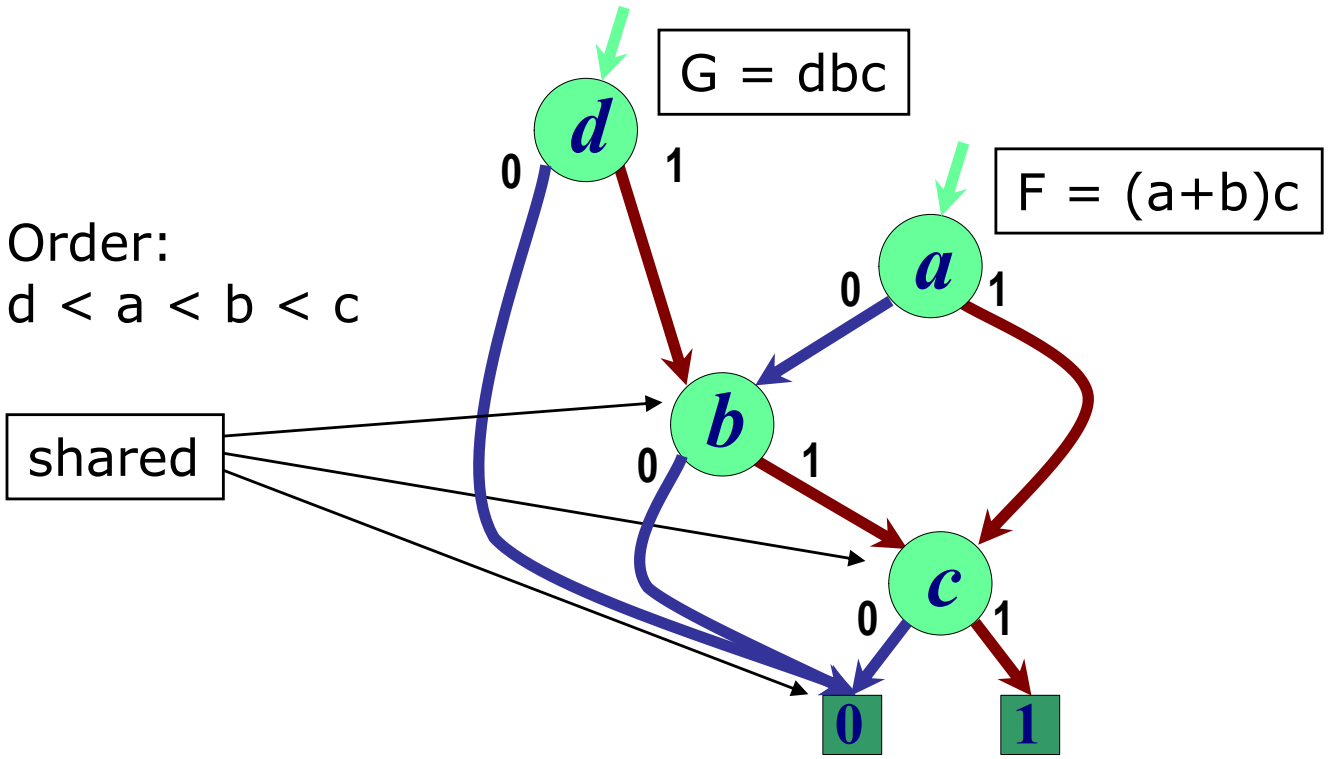
A network example



ROBDD- sharing

We already share subtrees within a ROBDD

…but we can share also among multiple ROBDDS



ROBDDs- why do we care ?

- **Easy to solve some important problems:**
  1. **Tautology checking**

     Just check if BDD is identical to function

  2. **Identity checking: F = G**

  3. **Satisfiability**

     Look for a path from root to leaf 1

- **All while having a compact representation**
  - **Use small memory footprint**

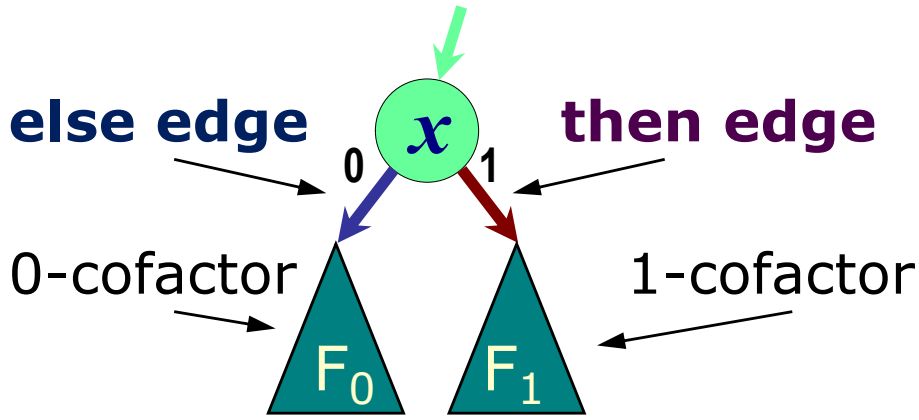
Logic operations with ROBDDs

1. Cofactor

- Given: ROBDD for G
- Positive co-factor G_x wrt. x: restrict G to $x = 1$

  Remove every node with label x, redirect incoming edges to node with then edge
- Negative co-factor $G_{x'}$ wrt. x: restrict G to $x = 0$

  Remove every node with label x, redirect incoming edges to node with else edge



Logic operations with ROBDDs

2. Boolean operators $\star$ ($\cdot$, $+$, $\oplus$, ...)

- Given: two ROBDD for **G**, **H**
- Find: the ROBDD of **G** $\star$ **H**

- ite operator:
  - ite(f,g,h) = fg + f'h
  - If (f) then (g) else (h)

- Recursive paradigm
  - Exploit the generalized expansion of G and H

ite (f,g,h) = ite(x,ite(f_x,g_x,h_x),ite($f_{x'}$, $g_{x'}$,$h_{x'}$))

Example

- Apply AND to two ROBDDs: f,g
  - fg = ite (f,g,0)
- Apply OR to two ROBDDs: f,g
  - f+g = ite (f,1,g)
- Similar for other Boolean operators

Boolean operators

Operator	*Equivalent ite form*
0	0
$f \cdot g$	$ite(f, g, 0)$
$f \cdot g'$	$ite(f, g', 0)$
f	f
$f'g$	$ite(f, 0, g)$
g	g
$f \oplus g$	$ite(f, g', g)$
$f + g$	$ite(f, 1, g)$
$(f + g)'$	$ite(f, 0, g')$
$f \overline{\oplus} g$	$ite(f, g, g')$
g'	$ite(g, 0, 1)$
$f + g'$	$ite(f, 1, g')$
f'	$ite(f, 0, 1)$
$f' + g$	$ite(f, g, 1)$
$(f \cdot g)'$	$ite(f, g', 1)$
1	1

ROBDD construction – terminal cases (AND)

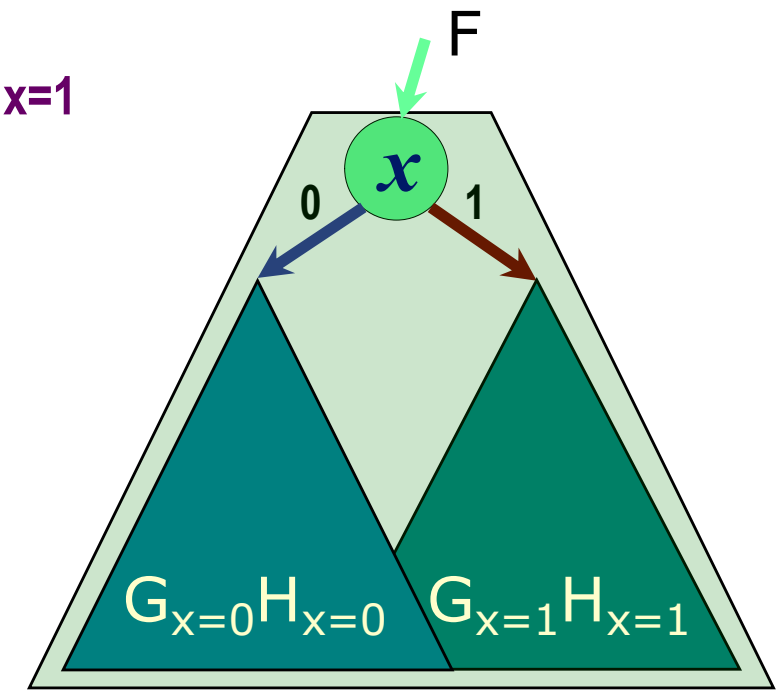
- Consider a simple example: compute AND of two ROBDDs
- Terminal cases:
  - AND (0,H) = 0
  - AND (1,H) = H
  - AND (G,0) = 0
  - AND (G,1) = G

ROBDD construction – recursive step (AND)

- $G(x,\ldots) = x' \; G_{x=0} + x \; G_{x=1}$
- $H(x,\ldots) = x' \; H_{x=0} + x \; H_{x=1}$

- $F = GH = x' \; G_{x=0} H_{x=0} + x \; G_{x=1} H_{x=1}$

Now we have reduced the problem to computing 2 ANDs of smaller functions



One last problem

- Suppose we have computed

  $G_{x=0}\, H_{x=0}$ and $G_{x=1} H_{x=1}$

- We need to construct a new node,
  - label: x
  - 0-cofactor($F_{x=0}$): ROBDD of $G_{x=0}\, H_{x=0}$
  - 1-cofactor($F_{x=1}$): ROBDD of $G_{x=1}\, H_{x=1}$

- BUT, we need first to make sure that we don't violate the reduction rules!

The unique table

To obey reduction rule #1:

- If $F_{x=0} == F_{x=1}$, the result is just $F_{x=0}$

To obey reduction rule #2:

- We keep a *unique table* of all the BDD nodes and check first if there is already a node

  $(x, F_{x=0}, F_{x=1})$

Otherwise, we build the new node

- And add it to the unique table

Putting all together

```
AND(G,H) {
        if (G==0) || (H==0) return 0;
        if (G==1) return H;
        if (H==1) return G;

        x = top_variable(G,H);
        G1 = G.then; H1 = H.then;
        G0 = G.else; H0 = H.else;
        F0 = AND(G0,H0);
        F1 = AND(G1,H1);
        if (F0 == F1) return F0;
        F = find_or_add_unique_table(x,F0,F1);

        return F;
}
```

Generalizing to Boolean operators

```
ITE(F,G,H) {

        if (terminal case) return (r = trivial result);
        cmp = computed_table_lookup(G,H);
        if (cmp != NULL) return (r = cmp);

        x = top_variable(F,G,H);
        t  = ITE (Fx , Gx , Hx )
        e  = ITE (Fx' , Gx' , Hx' )

        if (t == e) return (r = t);
        r = find_or_add_unique_table(x,t,e);
        computed_table_insert{(F,G,H),r};
        return ( r );

}
```

Logic operations - summary

- Recursive routines – traverse the DAGs depth first
- Two tables:
  - Unique table – hash table with an entry for each BDD node
  - Computed table – store previously-computed partial results
- Time complexity is quadratic in the BDD sizes

Some algorithmic complexities

▲Checking tautology	K time
▲Checking identity	K time
▲Satisfiability	linear (#vars)
▲Binary operators: AND, OR	quadratic
▲Smoothing, Consensus	quadratic

Motivation - again

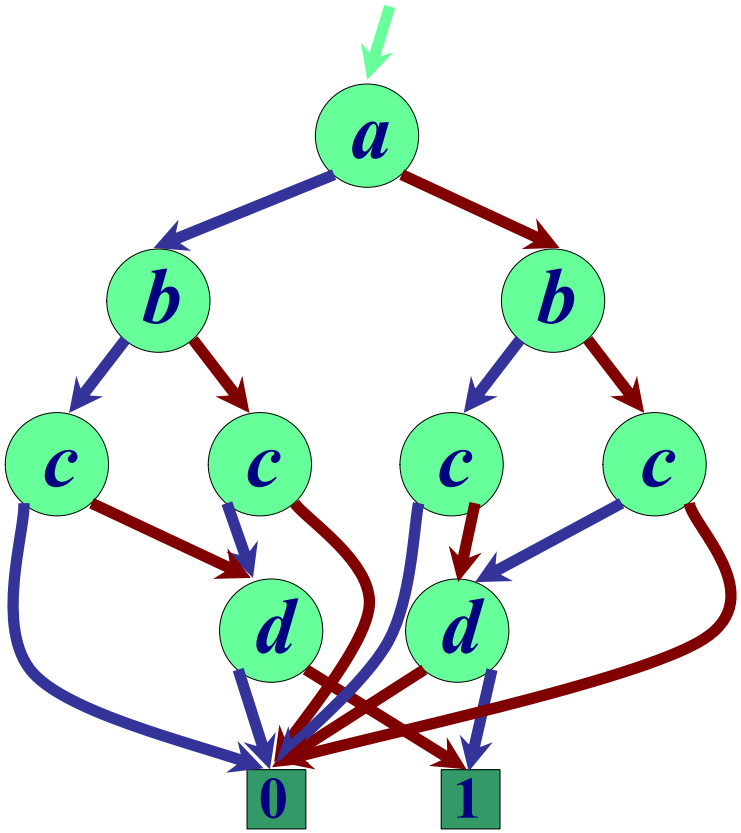
- Why are ROBDD popular?
  - Several intractable problems can be solved in polynomial time
    - Of the BDD size
  - In several cases, the BDD sizes grow mildly with the problem size
    - Variables
- This does not mean that BDD solve intractable problems in polynomial time
  - Few counterexamples exists

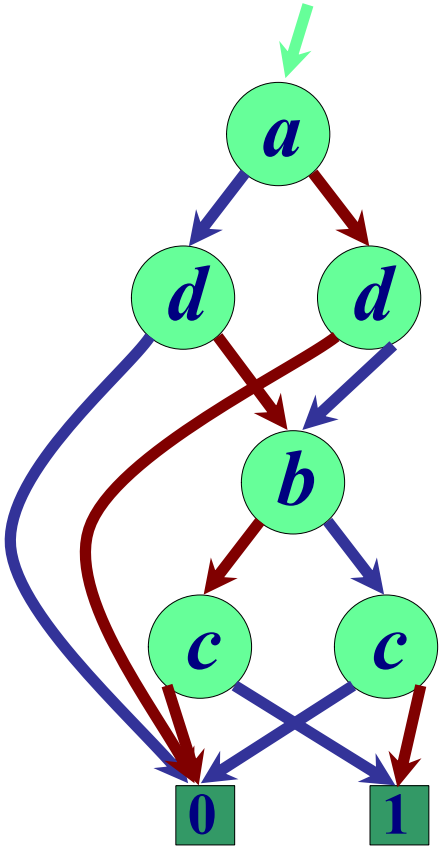
Module 2

- **Objectives:**
  - Variable ordering (static and dynamic)
  - Other diagrams and applications

The importance of variable order

$$F = (a \oplus d)(b \oplus c)$$





Ordering results

Function type	*Best order*	*Worst order*
addition	**linear**	**exponential**
symmetric	**linear**	**quadratic**
multiplication	**exponential**	**exponential**

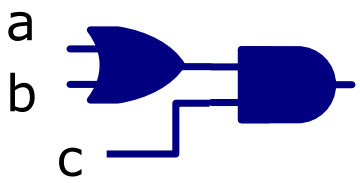
- **In practice:**
  - **Many common functions have reasonable size**
  - **Can build ROBDDs with millions of nodes**
  - **Algorithms to find good variables ordering**

Variable ordering algorithms

- *Problem*: given a function F, find the variable order that minimizes the size of its ROBBDs
- *Answer*: problem is intractable
- Two heuristics
  - Static variable ordering (1988)
  - Dynamic variable ordering (1993)

Static variable ordering

- Variables are ordered based on the network topology
  - *How*: put at the bottom the variables that are closer to circuit's outputs
  - *Why*: because those variables only affect a small part of the circuit



good order: a < b < c

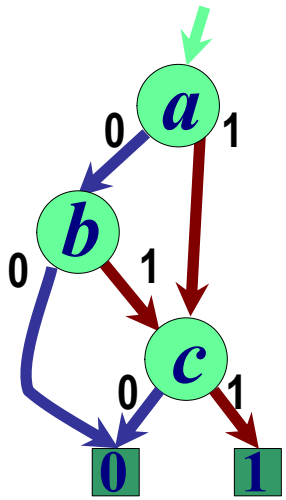
  - *Disclaimer*: it is a heuristic, results are not guaranteed

Dynamic variable ordering

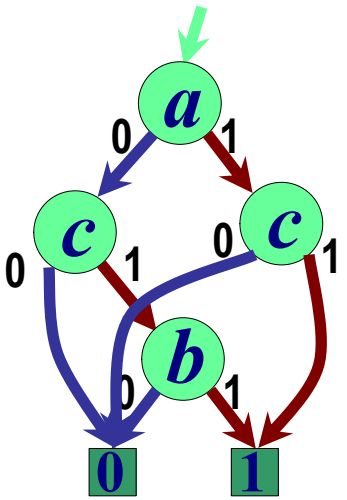
- Changes the variable order on the fly whenever ROBDDs become too big
- *How*: trial and error – sifting algorithm
  1. Choose a variable
  2. Move it in all possible positions of the variable order
  3. Pick the position that leaves you with the smallest ROBDDs
  4. Choose another variable …

Dynamic variable ordering

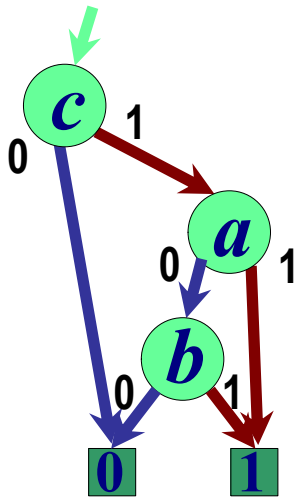
- **Tiny example: F=(a+b)c**
  - **We want to find the optimal position for variable c**



initial order:
a < b < c



Swap (b, c):
a < c < b

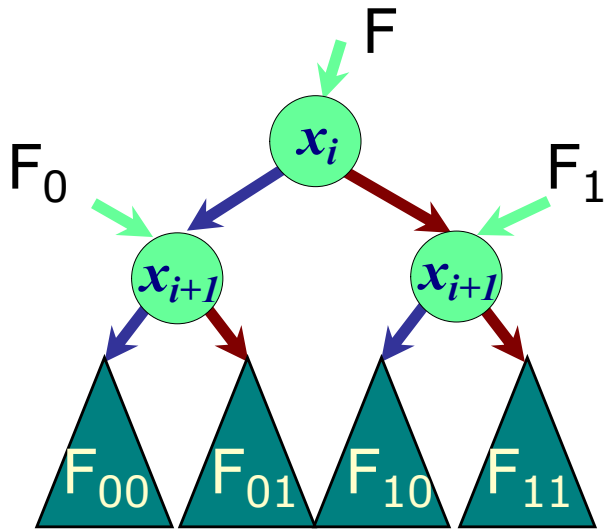


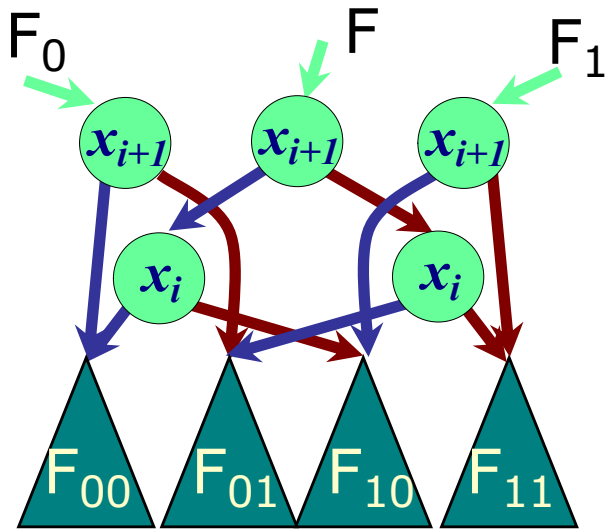
Swap (a, c):
c < a < b

Final order:
c<a<b

Variable swapping

$$ITE(x_i, F_1, F_0) =$$
$$= ITE(x_i, ITE(x_{i+1}, F_{11}, F_{10}), ITE(x_{i+1}, F_{01}, F_{00}))$$
$$= ITE(x_{i+1}, ITE(x_i, F_{11}, F_{01}), ITE(x_i, F_{10}, F_{00}))$$



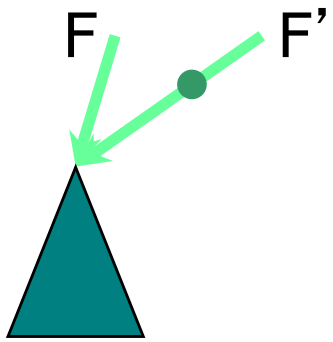


Dynamic variable ordering

- ***Key idea*: swapping two variables can be done locally**
  - ***Efficient:***
    - **It can be done just by sweeping the unique table**
  - ***Robust*:**
    - **It works well on many more circuits**
  - ***Warning*:**
    - **It is still non optimal**
    - **At convergence, you most probably have found only a local minimum**

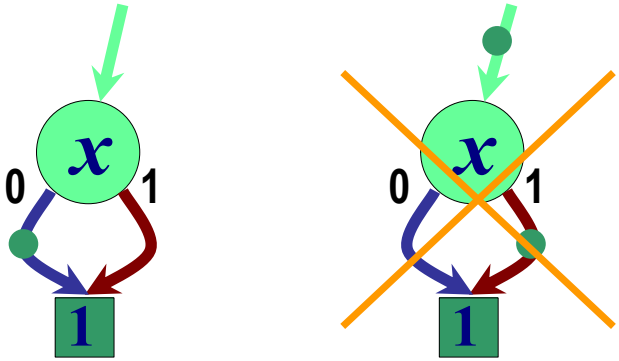
Improvements on BDDs

- **Complement edges** (1990)
  - Creates more opportunities for sharing
    - Fewer nodes
  - For every pair (F,F’), we
    - Only construct the ROBDD for F
    - F’ is given by using a complement edge to F
  - Which do you pick ?
    - THEN edge can never be complemented
    - Only constant value 1



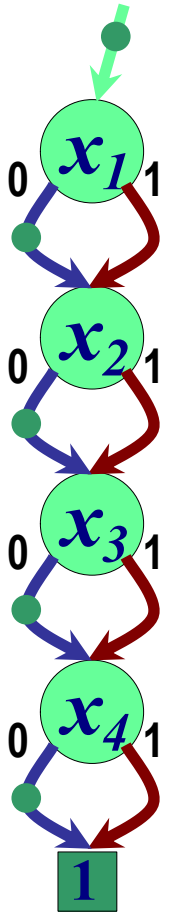
Complement edges

- **F = x**



$$F = x_1 \oplus x_2 \oplus x_3 \oplus x_4$$

- **Still canonical**

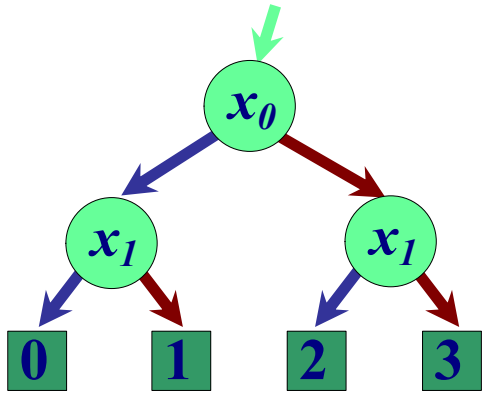


Other types of Decision Diagram

- Based on different expansion
  - OFDD
  - Ordered functional decision diagrams

$$F = F_{x=0} \oplus x(F_{x=0} \oplus F_{x=1})$$

- For discrete functions:
  - ADD
  - Algebraic decision diagrams



Boolean functions and sets of combinations

a	b	c	F	
0	0	0	0	
1	0	0	0	
0	1	0	0	
1	1	0	1	→ ab
0	0	1	1	→ c
1	0	1	1	→ ac
0	1	1	0	
1	1	1	0	

Boolean function:

$$F = (a\ b\ {\sim}c)\ \mathrm{V}\ ({\sim}b\ c)$$

Set of combinations:

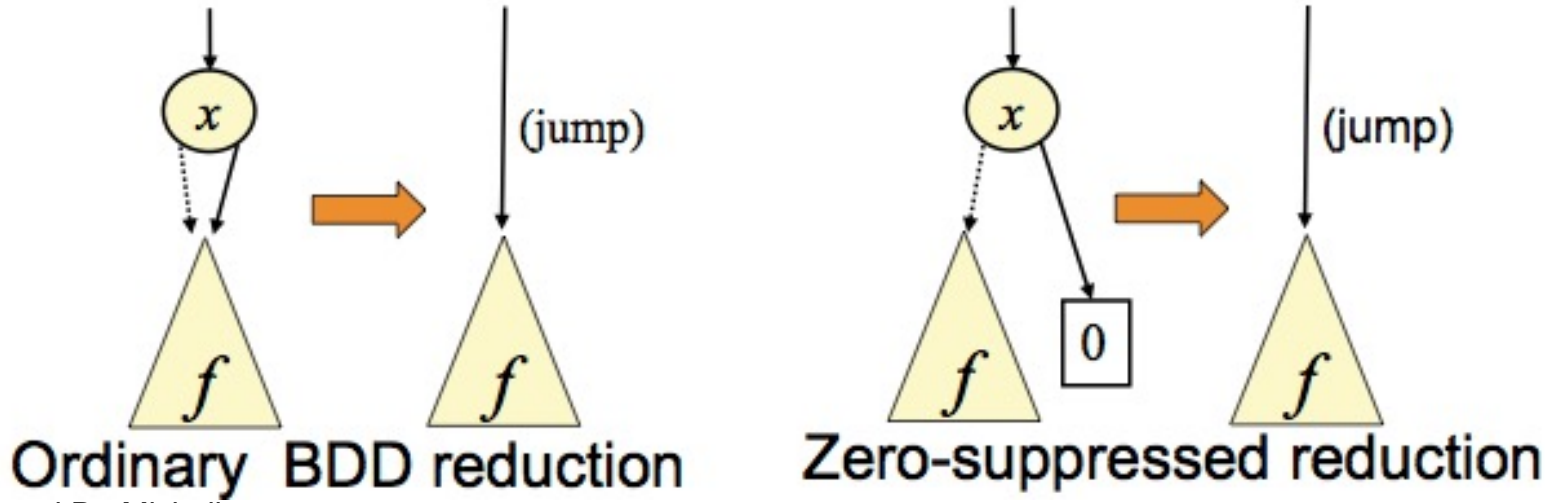
$$F = \{ab, ac, c\}$$

(customer's choice)

- Operations of combinatorial itemsets can be done by BDD-based logic operations.
  - Union of sets → logical OR
  - Intersection of sets → logical AND
  - Complement set → logical NOT

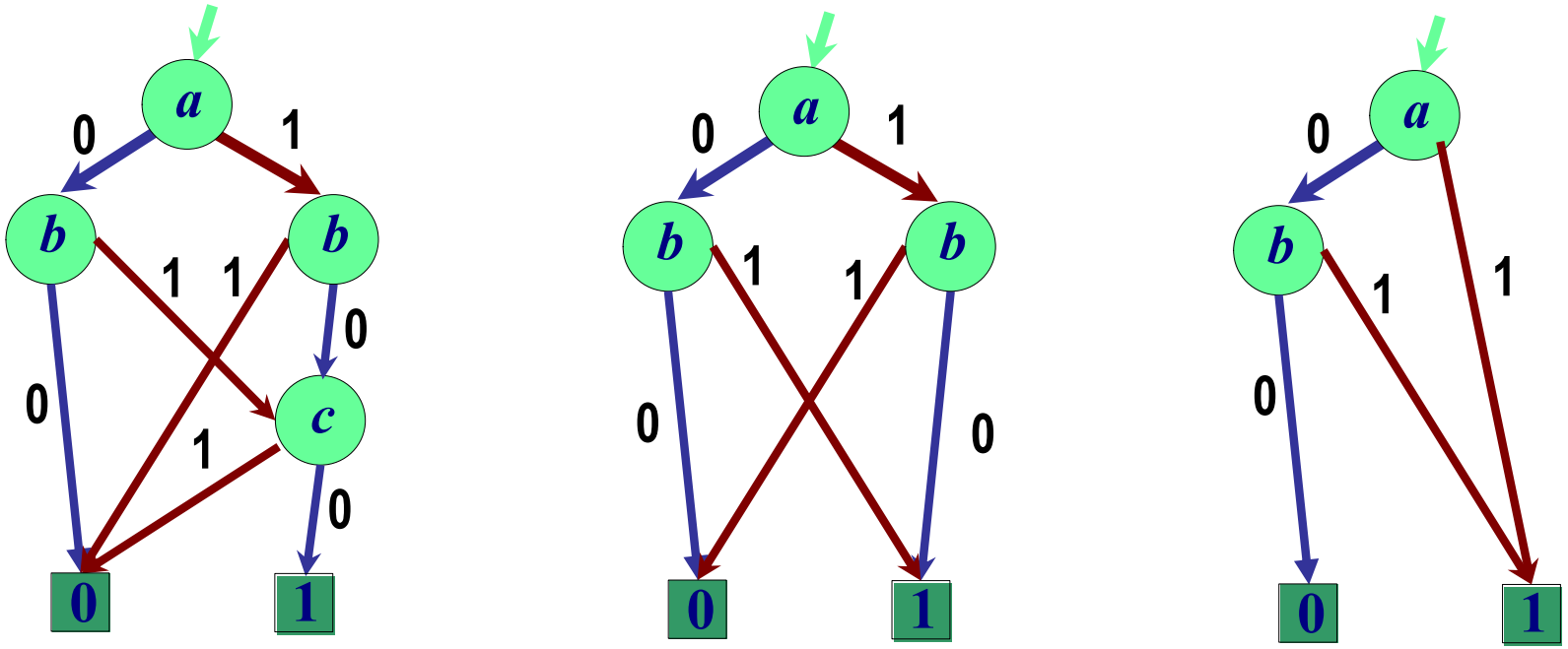
ZDDs -- Zero-suppressed BDDs

- BDDs with different reduction rules
  - Eliminate all nodes whose 1-edge points to the 0-leaf and redirect incoming edges to the 0-subgraph
  - Share all equivalent subgraphs
- If item x does not appear in any itemset, the ZDD node is eliminated
  - When average occurrence ratio of each item is 1%, than ZDD are more efficient than BDDs (up to 100 times)



ZDD - example

- Itemset {a,b}; characteristic function F = ab'c' + a'bc'



Eliminate all nodes whose 1-edge points to the 0-leaf and redirect incoming edges to the 0-subgraph

Summary

- BDDs
  - + Very efficient data structure
  - + Efficient manipulation routines
  - – A few important functions don't come out well
  - – Variable order can have a high impact on size
- Application in many areas of CAD
  - ▲ Hardware verification
  - ▲ Logic synthesis