



Decision Diagrams & Equivalence Checking

--- Introduction to Formal V

Instructor : Debdeep Mukhopadhyay, CSE IIT Madras



Introduction

- Fundamental Problem is to find out whether two Boolean Functions are functionally equivalent
- Are the following expressions equivalent?
 - $y = (\sim a)c + b(\sim c) + a(\sim b)$
 - $y = a(\sim c) + (\sim b)c + (\sim a)b$
- They are equivalent...
- The forms are not therefore canonical



Checking Equivalence is NP-complete

- Expand the combinational function in minterm form and compare them term by term.
- Property: Two equivalent functions have identical minterm expressions. This property is known as *canonicity of representation*.
- But runs into exponential size.



So, we require *Compactness*

- But also canonicity is necessary.
- As logical equivalence is a NP-complete problem, all canonical representations are exponential in the *worst case*.
- However if for all practical purposes, if the size is reasonable (manageable with our present day computation power) we are happy!
- So, various techniques have been proposed for various kinds of functions: like BDDs for Boolean functions; An alternative technique is called SAT.

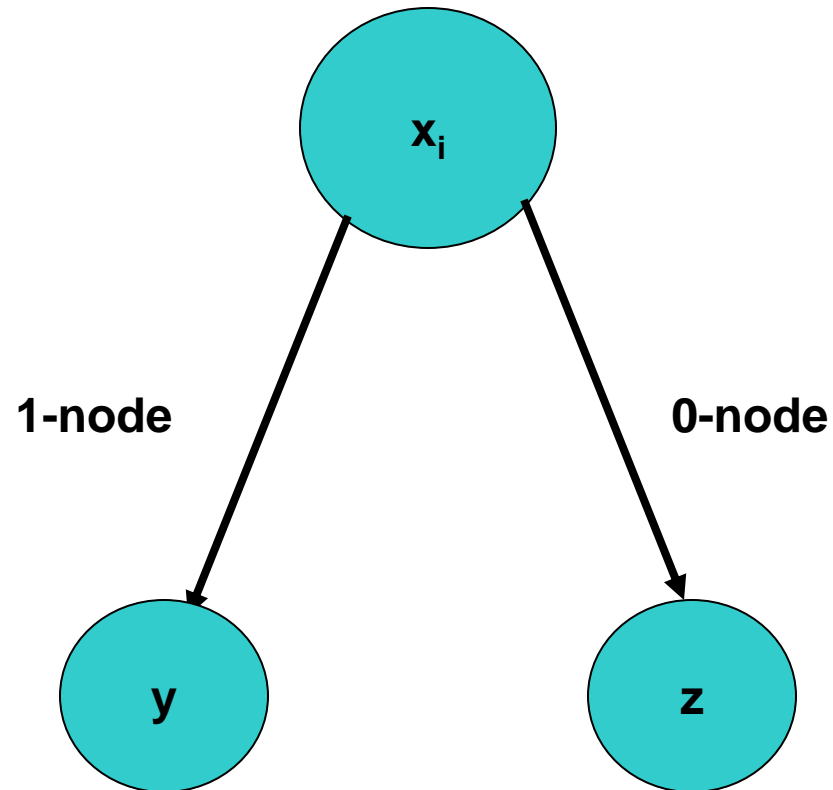


Binary Decision Diagrams

- *Defn:* A BDD is a *directed acyclic graph* (DAG) that represents a Boolean function.

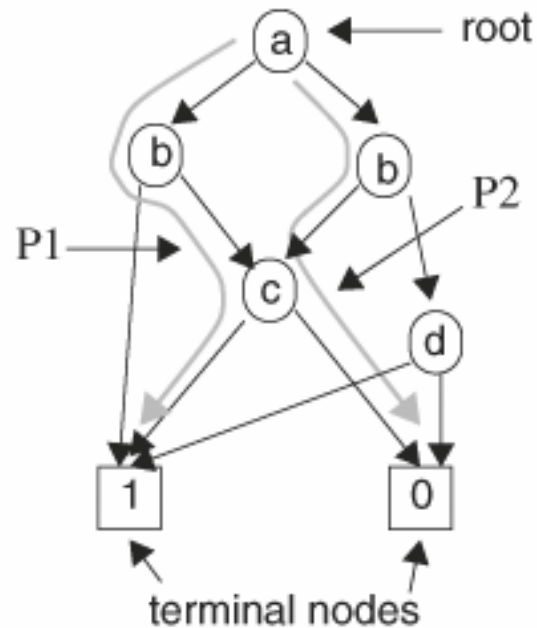
A node in the BDD is associated with a Boolean variable and has 2 outgoing edges

A Pictorial Description



$$f = x_i y + (\sim x_i) z$$

Functions from BDDs



- $f = ab + a(\sim b)c + (\sim a)(\sim b)d + a(\sim b)c$

// From the on-set: paths which lead to 1

- $\sim f = (\sim a)(\sim b)(\sim d) + a(\sim b)(\sim c) + (\sim a)b(\sim c)$

// From the off-set: path which leads to 0

- Using DeMorgan's laws,

$$f = (a + b + d)(\sim a + b + c)(a + (\sim b) + c)$$

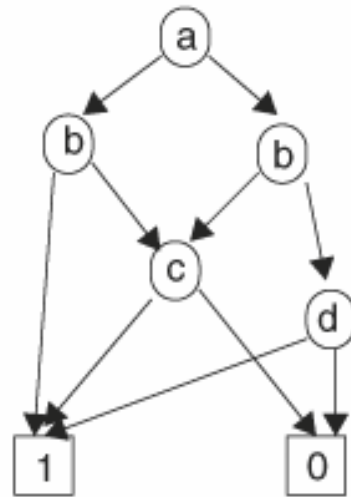


A top down recursive algorithm

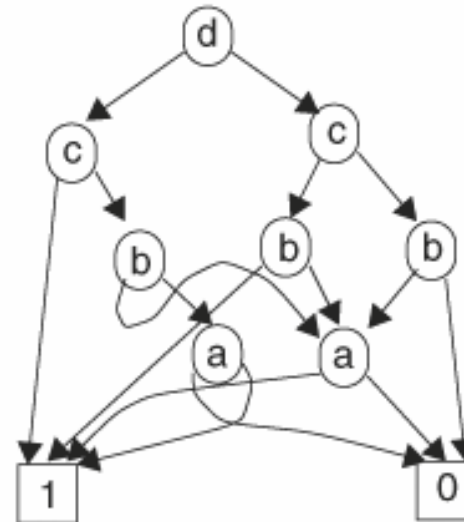
- Compute $\text{BDDFunction}(x)$
- Input: a BDD
Output: the Boolean function represented by the BDD,
 1. $x = \text{root}$ of BDD
 2. If x is a constant, return the constant
 3. Let y and z be the 1-node and the 0-node respectively of x
 4. Return $x\text{BDDFunction}(y) + (\sim x)\text{BDDFunction}(z)$

Ordered BDDs (OBDDs)

- Compute the Boolean Functions of the following BDDs:



A



B

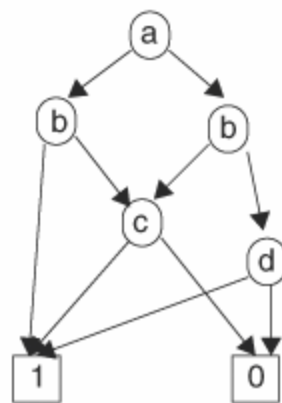
They are identical=> Ordering is important for canonicity

OBDD is a BDD with variables, that conform to an order.

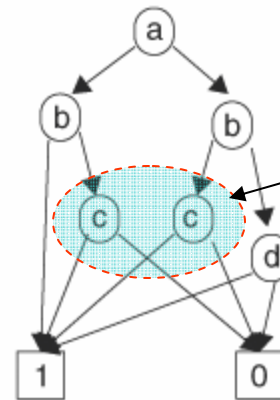
Ordering $x < y$ means that in any path in the OBDD, y is child of x

Still not unique

- Example: Same ordering but different DDs for the same function:
 - $y = ab + a(\sim b)c + (\sim a)bc + (\sim a)(\sim b)d$
 - $y = ab + a(\sim b)c + (\sim a)bc + (\sim a)(\sim b)d$



A

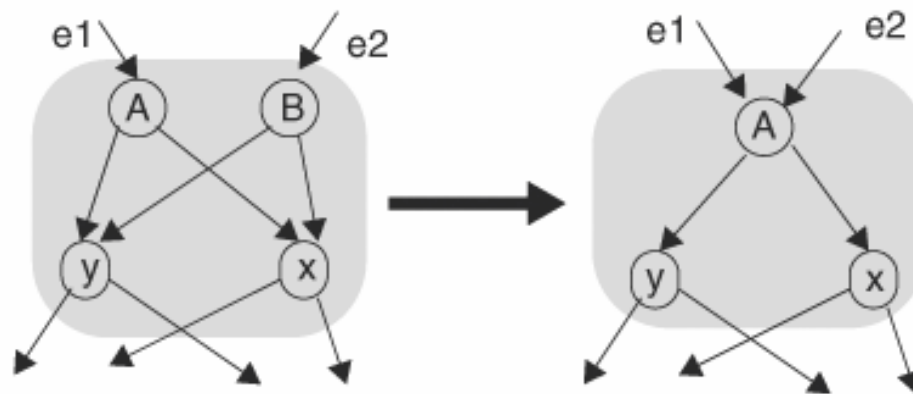


Why two c nodes?

B

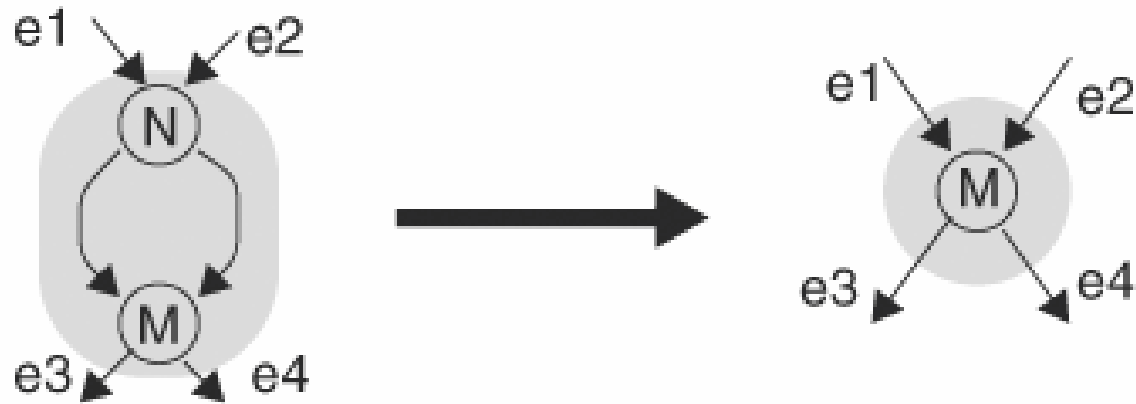
Merge

- **Merge**: Merge nodes A and B, if they have the same 0-node and 1-node



Eliminate

- *Eliminate* a node with two edges pointing to the same node





Canonicity of ROBDD

- **Theorem:** *Two Boolean Functions are equivalent iff their reduced OBDDs (ROBDD) are identical with respect to any variable ordering*



Operations on BDDs

○ Construction of BDDs:

- Shannon's expansion: any function can be expressed in the form:

$$f = xf_x + (\sim x)f_{(\sim x)}$$

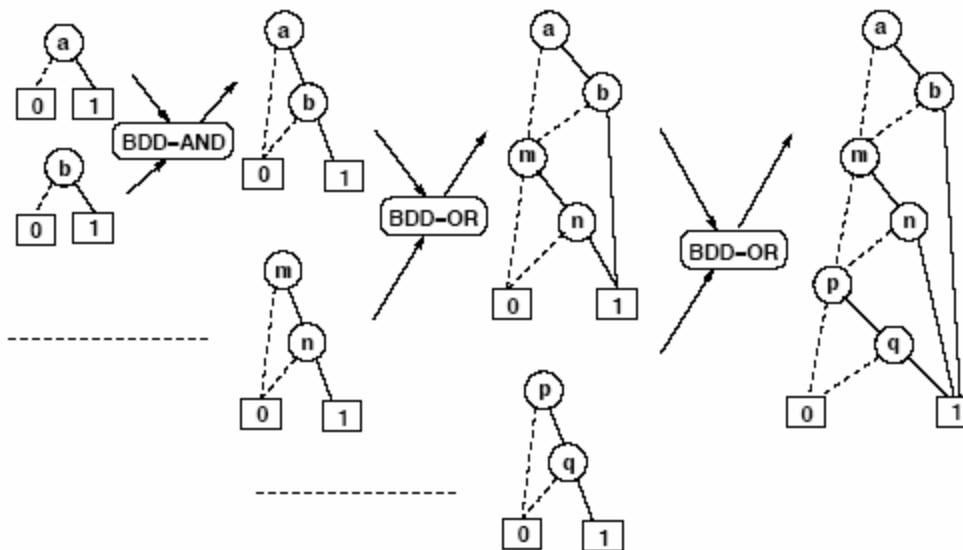
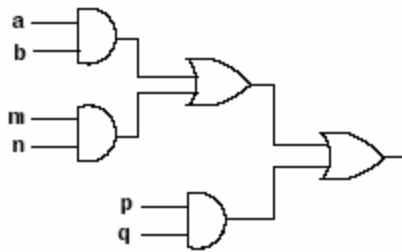
- Corresponds to a BDD with node x , with 1-edge pointing to f_x and 0-edge pointing to $f_{(\sim x)}$
- ## ○ Example: Construct the BDD for
- $f = ab + (\sim b)c$. Choose variable ordering $a < b < c$
 - Requires 2^n cofactor function for n variable function \Rightarrow Here 2^3 steps



Bottom Up approach

- BDDs are built for a , b and c (3 operations)
- ApplyAnd/Not to build ab , $\sim b$, $\sim bc$ (3 operations)
- ApplyOr to build $f = ab + (\sim b)c$ (1 operation)
- Thus in total we have 7 operations in place of 16 operations if we apply Shannons cofactor technique
- All the Apply Operations are polynomial in the number of nodes.

Incremental BDD construction



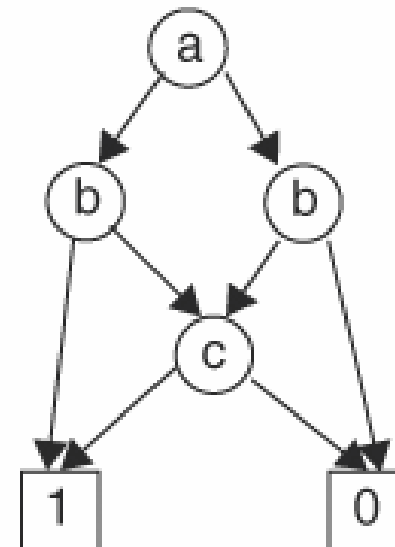
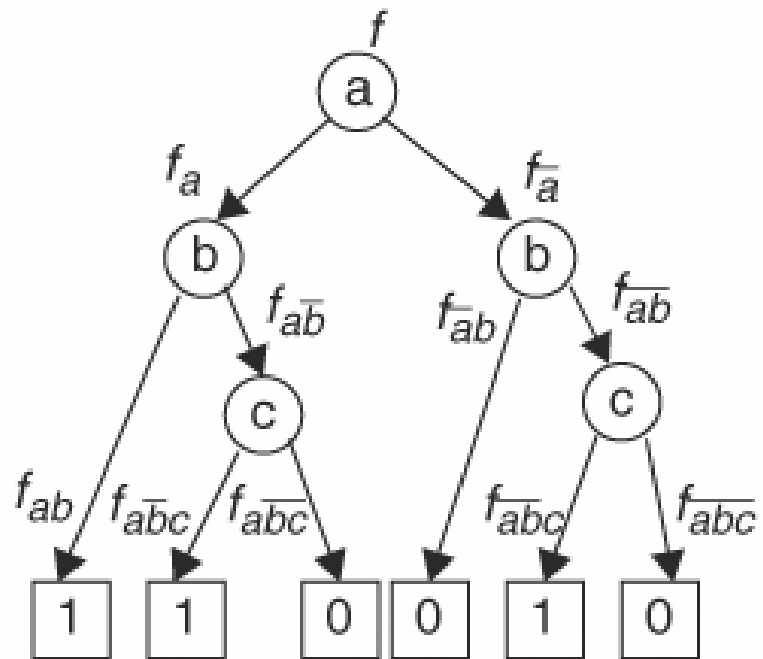
--- (dotted lines represent 0-nodes); solid lines represent 1-nodes.



Reduction

- Transforms a BDD into a ROBDD by applying recursively merge and eliminate
- Complexity is $O(n)$, where n is size of BDD
- Merge and eliminate are applied in the reverse ordering of the pre-decided order.

Reduction



Direction of Reduction

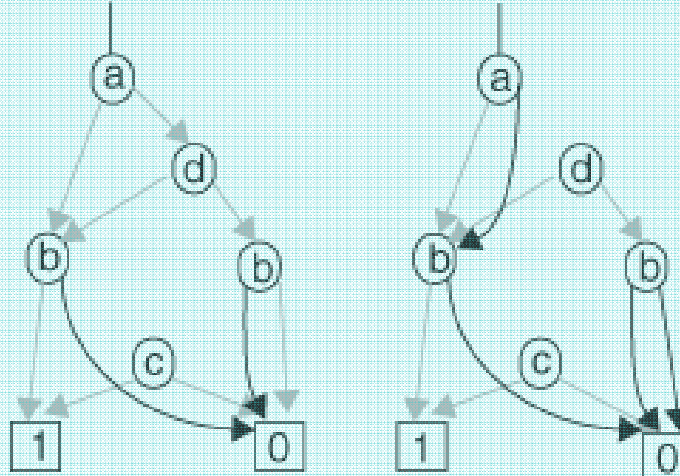
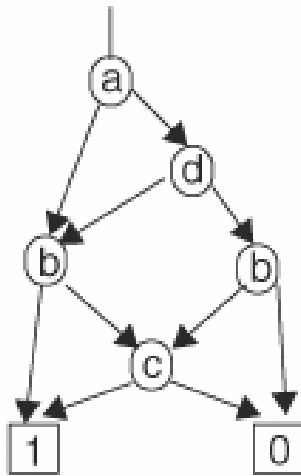
operation



Restriction

- Set certain variables to set values
- To restrict v to 1, simply direct all incoming edges to v to their 1-nodes
- To restrict v to 0, direct all incoming edges to v to their 0-nodes
- Reduce the resultant BDDs
- Clean up the BDD, by removing all nodes (except root) which does not have incoming edges.

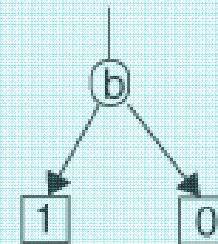
Example



c=0

d=1

Reduce





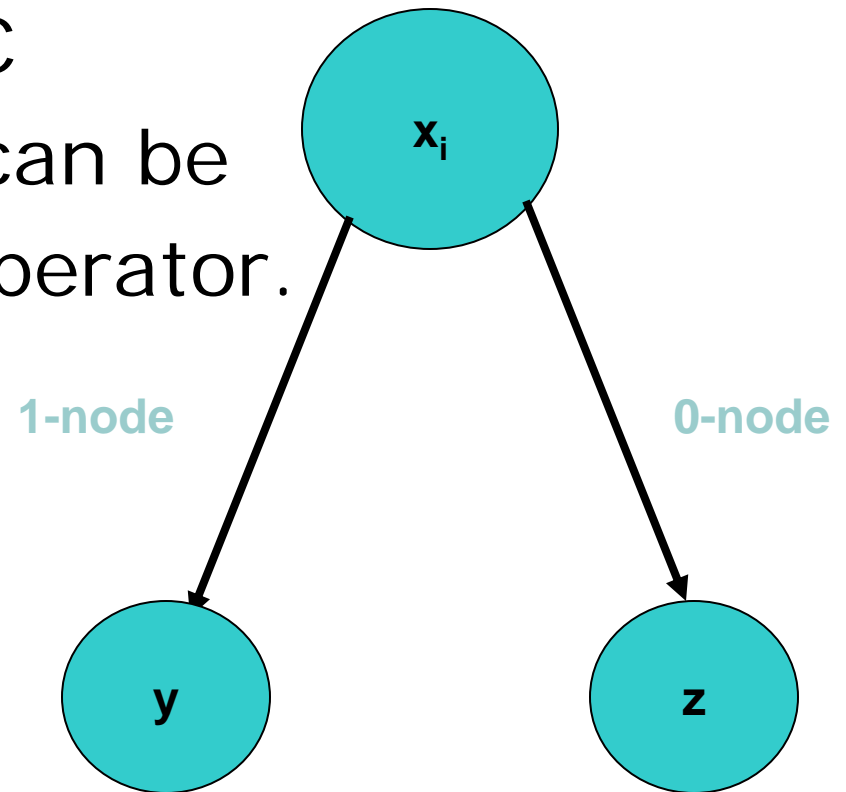
Boolean Operations

- As discussed we require Apply operations on BDDs, like AND, OR, etc
- Construct the BDD X for $y=ab+c$
- Construct the BDD for Applynot(X)
- Just need to flip the leaf nodes.

The ITE operator

- $\text{ITE}(A, B, C) = AB + (\sim A)C$
- Any node in the BDD can be
- expressed using the operator.

$\text{ITE}(x_i, y, z)$



$f = x_i y + (\sim x_i) z$

ITE operator encompasses all binary and unary operators

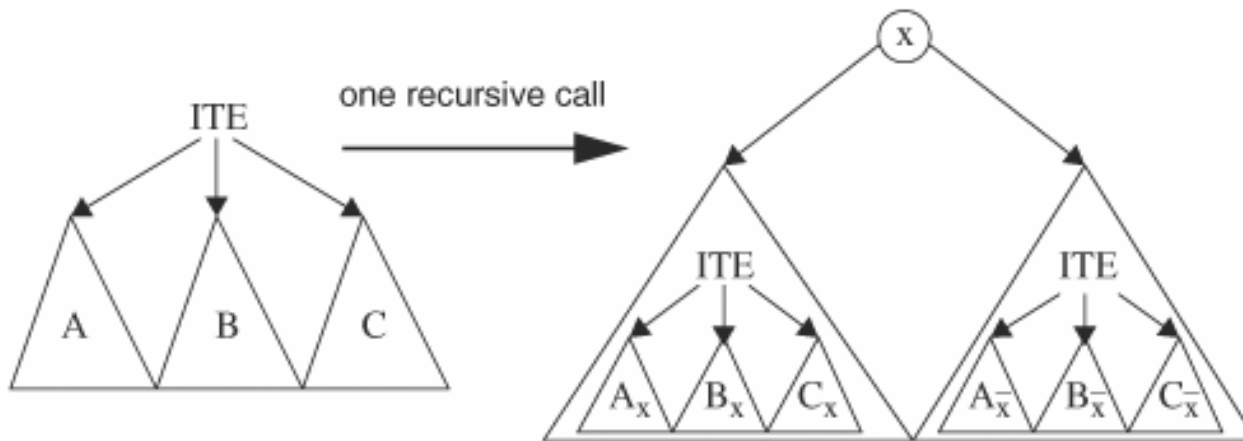
Operator	ITE form
\bar{X}	$ITE(X,0,1)$
XY	$ITE(X,Y,0)$
$X+Y$	$ITE(X,1,Y)$
$X \oplus Y$	$ITE(X,\bar{Y},Y)$
$MUX(X,Y,Z)$	$ITE(X,Y,Z)$
composition, $f(x,g(x))$	$ITE(g(x),f(x,1),f(x,0))$
$\exists_x f(x)$	$ITE(f(1),1,f(0))$
$\forall_x f(x)$	$ITE(f(1),f(0),0)$



Compute ITE

- $ITE(A, B, C) = AB + (\sim A)C$
 $= x(AB + (\sim A)C)_x + (\sim x)(AB + (\sim A)C)_{(\sim x)}$
 $= x(A_x B_x + (\sim A)_x C_x) + (\sim x)(A_{(\sim x)} B_{(\sim x)} + (\sim A)_{(\sim x)} C_{(\sim x)})$
 $= ITE(x, ITE(A_x, B_x, C_x), ITE(A_{(\sim x)}, B_{(\sim x)}, C_{(\sim x)}))$

Recursive Approach



1. Generates the BDD recursively
2. Recursion stops for trivial cases
3. $X = \text{ITE}(1, X, Y) = \text{ITE}(0, Y, X) = \text{ITE}(X, 1, 0) = \text{ITE}(Y, X, X)$
4. $1 = \text{ITE}(1, 1, Y) = \text{ITE}(0, Y, 1) = \text{ITE}(X, 1, 1) = \text{ITE}(Y, X, X)$



Reduction while construction

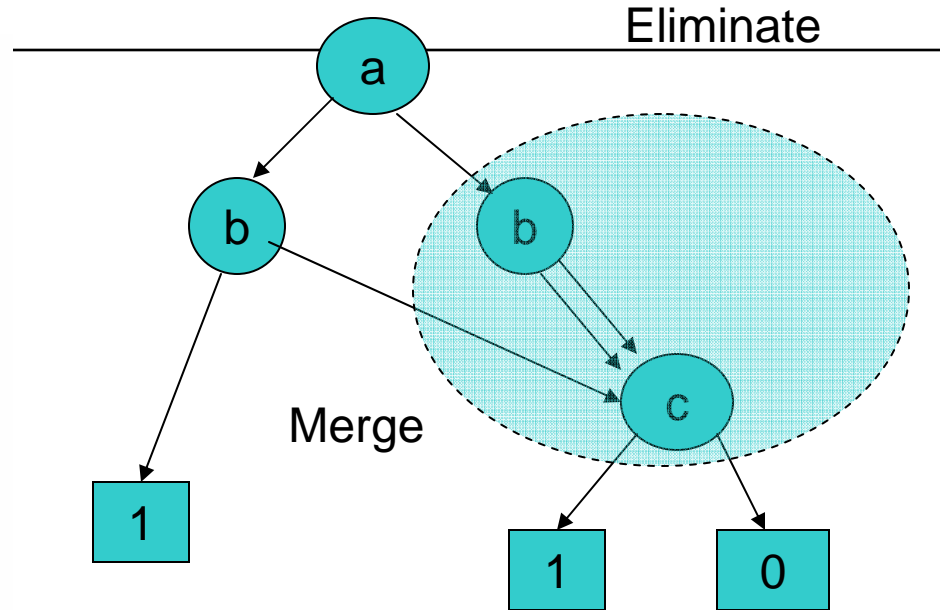
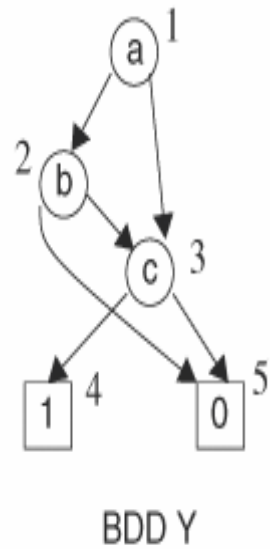
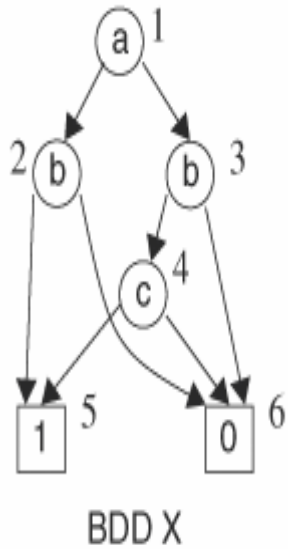
- **Merge** : Maintain a unique table, which remembers all the unique BDDs which have been generated. Indexed by the node and the 1-node and 0-node
- When calls of $\text{ITE}(A_x, B_x, C_x)$ and $\text{ITE}(A_{(\sim x)}, B_{(\sim x)}, C_{(\sim x)})$ return, before $\text{ITE}(A, B, C) = (x, \text{ITE}(A_x, B_x, C_x), \text{ITE}(A_{(\sim x)}, B_{(\sim x)}, C_{(\sim x)}))$ is created, we first check for the table for an entry with x node and the same 1 and 0 node.
- If such a node exists, use it.



Eliminate

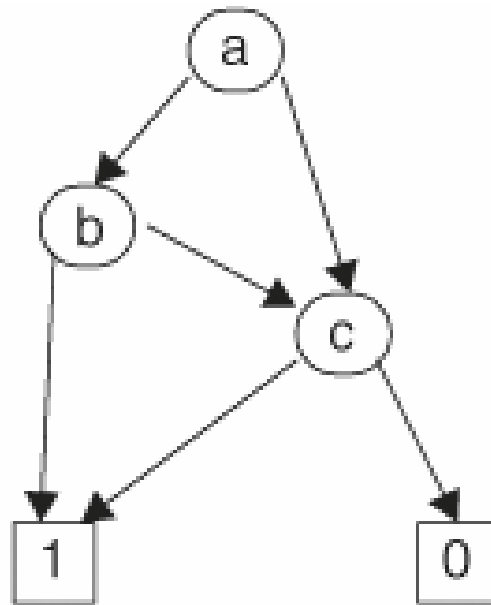
- If *then* node is identical to *else* node, no extra node is created.
- We shall use ITE to compute Apply operations.
- Dynamic programming can be handy.

ApplyOr(X,Y)



- $X+Y = \text{ITE}(X, 1, Y) = (a, \text{ITE}(X.2, 1, Y.2), \text{ITE}(X.3, 1, Y.3))$
- $\text{ITE}(X.3, 1, Y.3) = (b, \text{ITE}(X.4, 1, Y.3), \text{ITE}(X.6, 1, Y.3))$
- $\text{ITE}(X.6, 1, Y.3) = \text{ITE}(0, 1, Y.3) = Y.3 // \text{add } Y.3 = c \text{ to table}$
- $\text{ITE}(X.4, 1, Y.3) = (c, \text{ITE}(1, 1, 1), \text{ITE}(0, 1, 0)) = (c, 1, 0) = Y.3$ (already exists so return but donot create extra node => merge with previous instance)
- $\text{ITE}(X.2, 1, Y.2) = (b, \text{ITE}(X.5, 1, Y.5), \text{ITE}(X.6, 1, Y.3))$
- $\text{ITE}(X.6, 1, Y.3) = Y.3$ (already computed once)
- $\text{ITE}(X.5, 1, Y.5) = 1$

Final BDD





Complexity

- Maintain table entries with arguments
- If previous query is made, do not recompute
- Possible size of table to compute $\text{ITE}(A, B, C)$ is $O(|A| |B| |C|)$.

Final Algorithm

Boolean Operations on BDDs: ITE(A, B, C)

Input: Three BDDs, A, B, and C.

Output: ROBDD representing $ITE(A, B, C)$.

1. If $ITE(A, B, C)$ is a terminal case, return with result.
 2. If $ITE(A, B, C)$ is in the computed table, return with result.
 3. Select the root variable x that is ordered earliest.
 4. Compute $BDD_0 = ITE(A_{\bar{x}}, B_{\bar{x}}, C_{\bar{x}})$ and $BDD_1 = ITE(A_x, B_x, C_x)$.
 5. If $(BDD_0 = BDD_1)$, return BDD_0 .
 6. If $(x, \text{root of } BDD_1, \text{root of } BDD_0)$ is in the unique table, return the existing node.
 7. Create BDD node x with 0-edge and 1-edge pointing to BDD_0 and BDD_1 respectively.
-



Variable Ordering

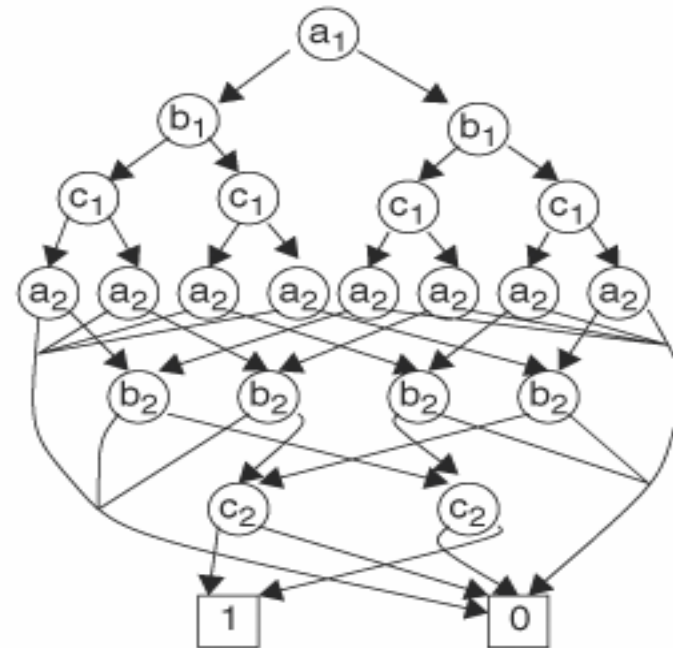
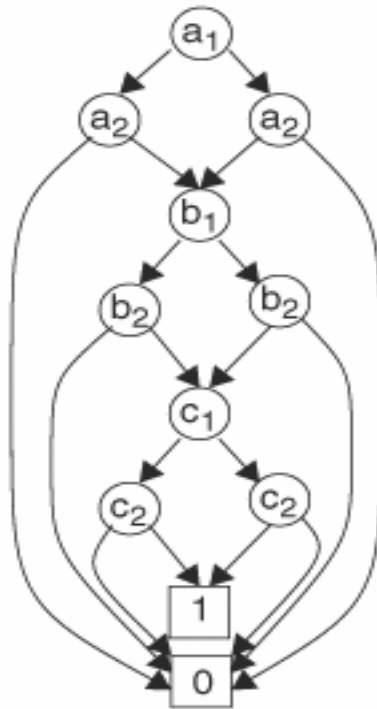
- Different variable ordering can cause drastic differences in BDD size.
- Task of finding an optimal variable ordering for a function is hard (NP-hard).
- Let us observe certain facts to help us in choosing the order. However these are heuristics, and so they do not guarantee an optimal solution. However they produce near optimal solutions or good results for most practical purposes.



Example

- Build a BDD for
 - $(a_1 \wedge a_2)(b_1 \wedge b_2)(c_1 \wedge c_2)$
- Consider order:
 - Order I: $a_1 < a_2 < b_1 < b_2 < c_1 < c_2$
 - Order II: $a_1 < b_1 < c_1 < a_2 < b_2 < c_2$

BDDs



Number of nodes in I is 11, while that in II is 23. Improper ordering can lead to explosion.



Observation 1

- Observe that in I , the values of a_1 and a_2 determine the result much quickly
- *When the variables are ordered together early that completely determine the value of the function, fewer nodes appear on the paths from BDD root to constant roots, and hence simpler BDDs result.*



Size of BDD depends on height.

- Intuitive Informal Proof:

- Size of BDD depends on height and width
- Width of BDD may be defined as the number of paths from root to constant nodes.
- Height of BDD is the average number of nodes from root to constant nodes.
- But each nodes lead to 2 paths.
- So, with the number of nodes, the number of paths also increase \Rightarrow width depends on height
- So, size is determined by the height of BDD.

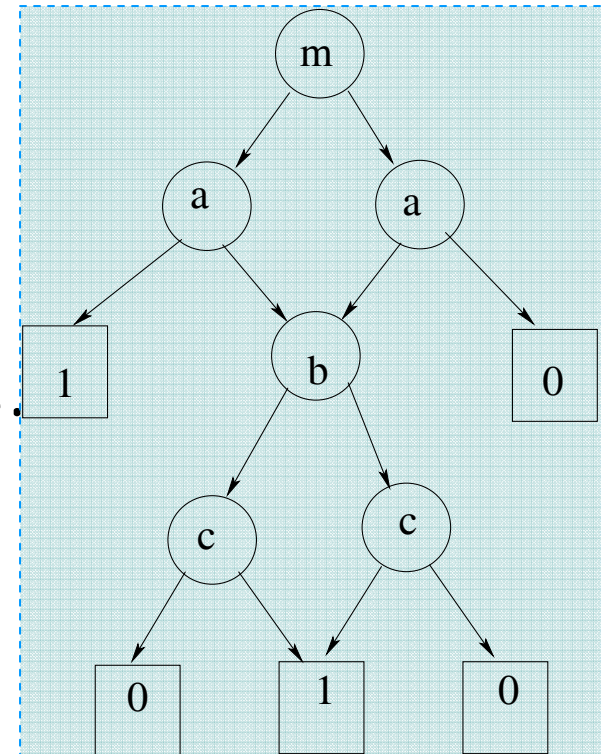


First Point

- A good variable ordering should have the property that as variables are evaluated one by one in the order, the function value is decided with fewer number of variables.
- In such a case the number of nodes will be less in a path of the BDD, thus the height will be less and so the size of the BDD.

Observation 2

- Node sharing reduces the size of the BDD.
- $f = ma + [m(\sim a) + (\sim m)a][b(\sim c) + (\sim b)c]$
- Note that the values of successor of b are independent of the values of its predecessors. So, b can be shared.





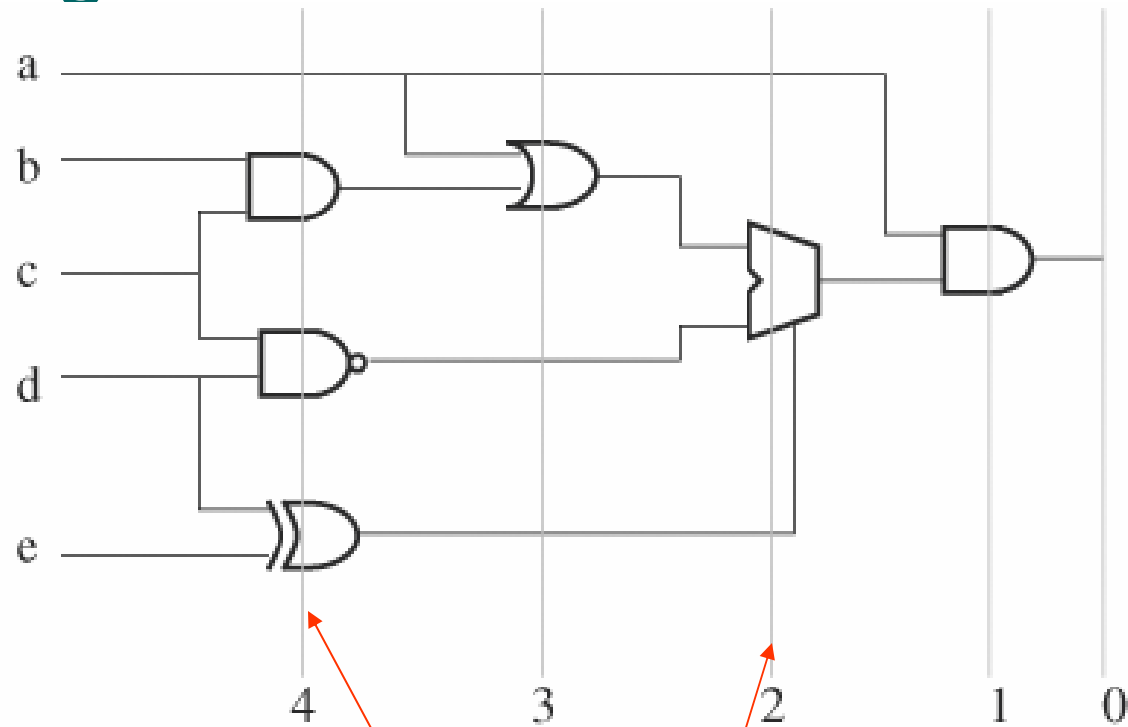
Point 2

- The more independent the successors and the predecessors are in a variable ordering, more the chance for sharing. This results in smaller BDDs.
- *A good variable ordering groups interdependent variables closer together.*

Ordering from a circuit

a	1
b	4
c	3
d	3
e	3

Suggested Order:
a<c<d<e<b





Heuristics

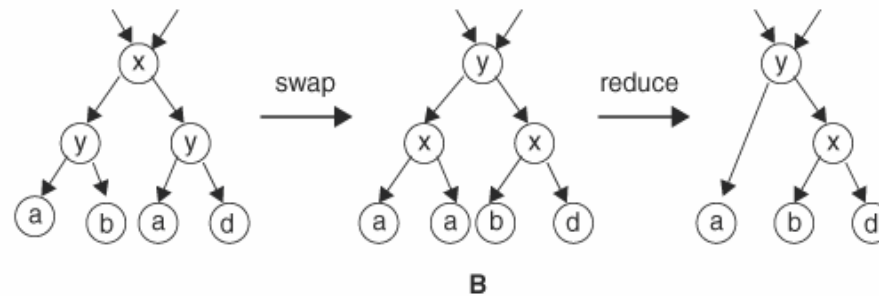
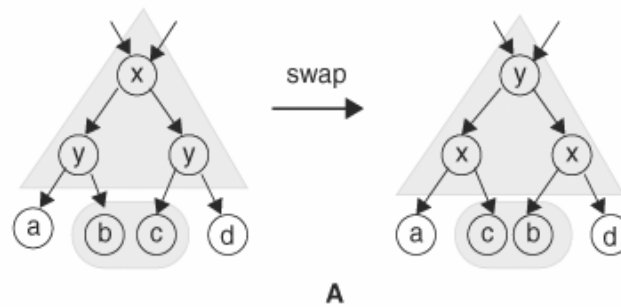
- Order first variables whose distance is less from the output (that is those that are closer to the output are placed early in the order). Why?
- Order the primary input variables such that the intermediate node values are determined as soon as possible.



Dynamic Variable Ordering

- Relevant when choosing ordering while compositing BDDs.
- Compose f , g and h
- Static algorithm: Decide the ordering of the variables in f , g and h before-hand
- Dynamic algorithm:
 - Choose the ordering for f , g and h .
 - While composing say f and g , as the BDD size crosses a thresh-hold, perform a change in the variable ordering and check.
 - Greedy Algorithms may be effective.

Swapping of two adjacent BDD variables



Reduction is necessary to maintain local canonicity.



A Heuristic Algorithm: Using Swap

- Any variable ordering can be obtained by swapping.
- However no known guidance exists.
- A possible heuristics:
 - Develop an ordering
 - Choose a variable
 - A shifting algorithm moves a selected variable to all possible positions and chooses the one with the smallest BDD size.



Functions and BDD sizes

- There are some functions whose size is always exponential in the no of input variables: eg multipliers.
- There are some functions whose size is always polynomial in the no of input variables: eg symmetric functions
 - $f(a,b,d) = a(\sim b)(\sim d) + (\sim a)b(\sim d) + (\sim a)(\sim b)d$
- But most functions are dramatically sensitive to variable ordering.



Binary Moment Diagrams (BMD)

- $B^n \rightarrow R$
- Treats boolean variables as integer variables restricted to 0 or 1.
- Non-terminal nodes have the interpretation that if the 1-edge is taken, the variable of the node is included, else excluded.
- A path from the root to a leaf node represents a term in the polynomial by multiplying the value of the leaf node and all included variables along the path.



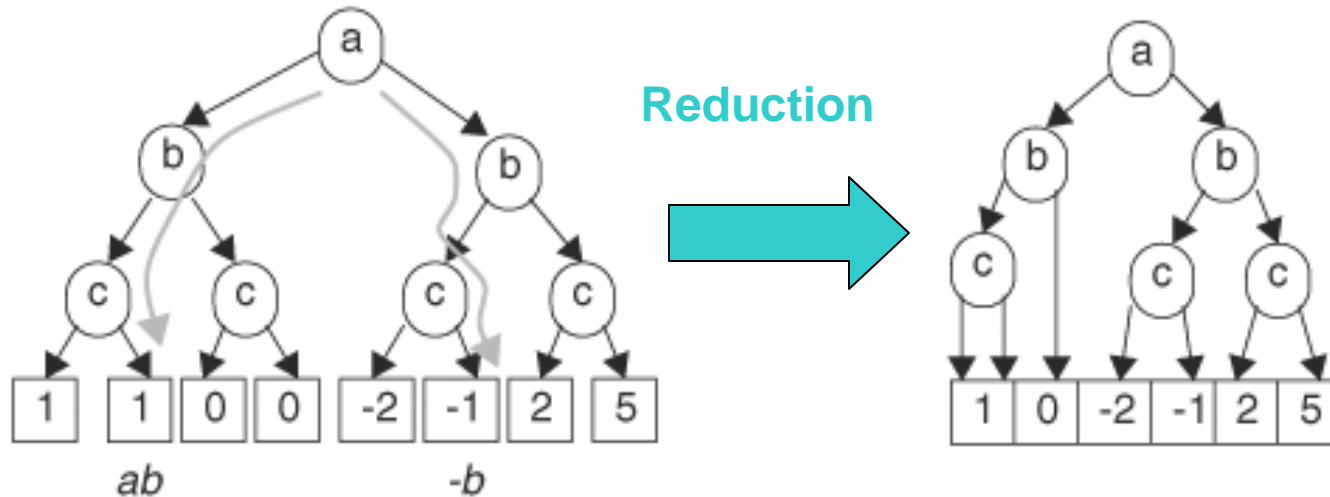
An example

Input (abc)	Power	Input (abc)	Power
000	5	001	7
010	4	011	4
100	5	101	7
110	5	111	6

- $f(a,b,c) = 5(1-a)(1-b)(1-c) + 4(1-a)b(1-c) + 5a(1-b)(1-c) + 5ab(1-c) + 7(1-a)(1-b)c + 4(1-a)bc + 7a(1-b)c + 6abc = -b + ab + 2c - 2bc + abc + 5$

BMD

○ $f(a,b,c) = abc + ab - 2bc - b + 2c + 5$



Reduction rules:

1. If the 1-edge points to 0, remove the node and redirect the incoming edges to its 0-node
2. All isomorphic subgraphs are merged.



Boolean Satisfiability

- Alternative to BDD in checking equivalence
- The problem of Boolean Satisfiability decides whether a Boolean formula has an assignment of variables such that the expression evaluates to 1.



Relation with Equivalence Checking

- Can be translated into SAT problem by xoring 2 functions, f and g
 - $d = f \oplus g$
- *If expression d is satisfiable, then f and g are not equivalent.*
 - *Otherwise they are equivalent.*



The SAT problem

- Expression in boolean satisfiability is in CNF form (POS)
- A sum is called clause.
- If each clause of the CNF form has at most 2 variables, then the problem is called 2-SAT (*polynomial time solution exists*).
- If it has more than 3 variables, 3-SAT (NP-complete)
- Satisfiability of any Boolean function can be reduced to a 3-SAT problem in polynomial time.



Example

- $f(a,b,c) = (a+b+c)(\sim a+b+(\sim c))(a+(\sim b)+(\sim c))$
 $((\sim a)+(\sim b)+(\sim c))((\sim a)+(\sim b)+c)$
- $a=1, b=1, c=0 \Rightarrow$ fails
- $a=1, c=0, b=0 \Rightarrow$ satisfied
- Worst case all $2^3=8$ assignments need to be checked
- We present 2 algorithms to solve SAT problems.




Resolvent Algorithm

- Prove that:
 - $f = (x + A)(\sim x + B) = (x + A)(\sim x + B)(A + B)$
 - $f = xC + (\sim x)D = xC + (\sim x)C + CD$
- A and B are the sum of literals. C and D are the product of literals.
- $A + B$ is called the resolvent of $(x + A)$ & $(\sim x + D)$ [*useful for Conjunctions*]
- CD is called the concensus of xC and $\sim xD$
[*useful for Disjunctions*]



Result

- $A+B$ is satisfiable *iff* $(x+A)(\sim x+B)$ is satisfiable.
- Proof: $(A+B)$ is satisfiable $\Rightarrow A, B$ or both are satisfiable. Let A be so.
- From identity, if $x=0$, $LHS=A$, and hence is satisfiable.
- If B is satisfiable let $x=1 \Rightarrow LHS=B$ and hence is also satisfiable.

- 
-
- Let $(x+A)(\sim x+B)$ be satisfiable \Rightarrow x is either 0 or 1.
 - Let $x=0$, $LHS=A$. So, A is satisfiable and thus $A+B$.
 - If $x=1$, $LHS=B$. So, B is satisfiable and thus is $A+B$.
 - QED.
 - ***Caution:*** *Solution for $A+B$ does not necessarily satisfy $(x+A)(\sim x+B)$. It only reduces the problem complexity by 1 variable.*



Extension to more than 2 clauses

- $(x+A)(\sim x+B)(x+C)(\sim x+D)$ is satisfiable iff $(A+B)(A+D)(B+C)(C+D)$ is so.
- $(a+b)(\sim a+b)$, b is unate and a is binate
- Straight Forward cases appearing while resolving :
 - Unate variable or pure literal rule:
 - $b=1$ (assign values so that the clauses which has the unate variable becomes 1).
 - Unit clause rule:
 $(a+b+c)(\sim b)(b+(\sim c)+d) \Rightarrow b=0$.

Example

Determine the satisfiability of

- $f(a,b,c) = (a+b+c)(\sim a+b+(\sim c))(a+(\sim b)+(\sim c))$

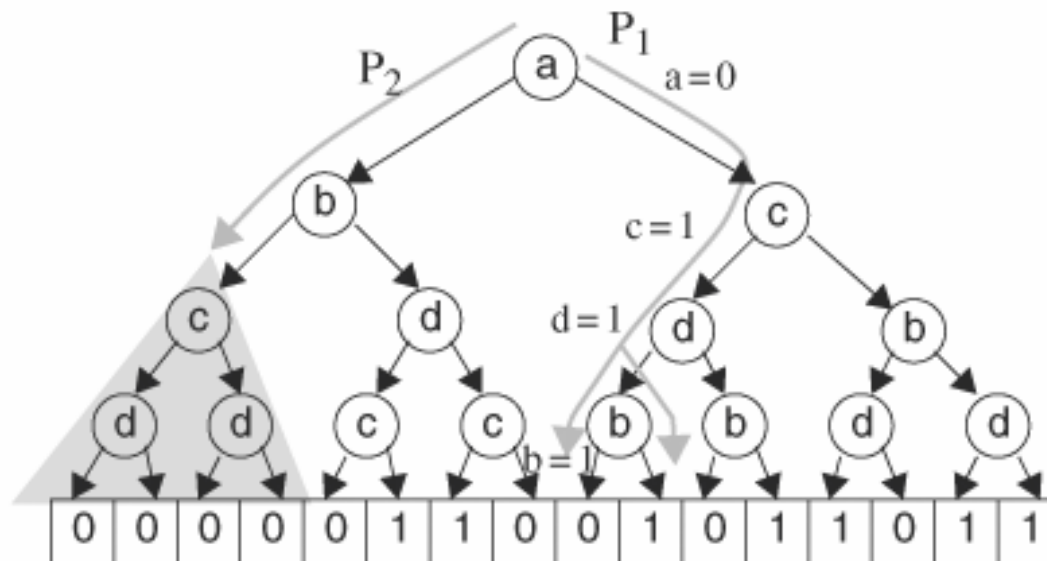
$$(\sim a+(\sim b)+(\sim c))(\sim a+(\sim b)+c)$$

- Resolving over $a \Rightarrow$ Expression is satisfiable iff $(\sim b+(\sim c))$ is satisfiable. Since it is so, so is the expression.
- $a=0, b=0, c=0$ satisfies $(\sim b+(\sim c))$ but does not satisfy the expression.

Binate Clauses	Resolvent	Simplifies To
$a+b+c, \bar{a}+b+\bar{c}$	$b+c+b+\bar{c}$	1
$a+b+c, \bar{a}+\bar{b}+\bar{c}$	$b+c+\bar{b}+\bar{c}$	1
$a+b+c, \bar{a}+\bar{b}+c$	$b+c+\bar{b}+c$	1
$\bar{a}+b+\bar{c}, a+\bar{b}+\bar{c}$	$b+\bar{c}+\bar{b}+\bar{c}$	1
$\bar{a}+\bar{b}+\bar{c}, a+\bar{b}+c$	$\bar{b}+\bar{c}+\bar{b}+\bar{c}$	$\bar{b}+\bar{c}$
$\bar{a}+\bar{b}+c, a+\bar{b}+\bar{c}$	$\bar{b}+c+\bar{b}+\bar{c}$	1

Technique though elegant cannot be applied for larger examples as it has to handle exponential number of terms.

Search Based Algorithm





Algorithm

Search-Based SAT Algorithm

SolveSAT()

input: a formula

output: SAT or UNSAT

```
    forever {  
        state = select_branch(); // choose and assign a variable  
        if (state == EXHAUSTED) return UNSAT;  
  
        result = infer(); // infer variable values  
        if ( result == SAT)  
            return SAT;  
        else if (result == UNSAT)  
            backtrack(); // backtrack to a prior decision  
        else // result == INDETERMINATE  
            continue; // need further assignment  
    }
```



- Select_branch : Choose the branch such that the chosen assignment has a high probability to satisfy the expression.
- **Example:** choose the variable which has larger number of literals in the remaining unsatisfied clause.
- Infer: After substituting the assigned values, we apply pure literal rule and unit clause rules (already discussed).



○ Back_track:

- Chronological backtracking : Go to the last variable
- Nonchronological Backtracking:
Reverse a previous variable



Example

- $f = (a+b)(\sim a+b)(a+(\sim b)+c)(\sim a+(\sim c)+d)(\sim a+(\sim b)+c)(\sim b+c+(\sim d))(a+(\sim d))(\sim d+e)(\sim d+e+f)(a+(\sim e)+f)$
- $\sim d$ and a each appear max times (4)
- Assign $d=0$
- After BCP:
 $f = (a+b)(\sim a+b)(a+(\sim b)+c)(\sim a+(\sim c))(\sim a+(\sim b)+c)(a+(\sim e)+f)$
- Now a and $\sim a$, each highest count of 3. Choose $a=1 \Rightarrow f = b(\sim c)(\sim b+c)$
- Pure literal rule: $b=1, c=0$...fails
- Backtrack to last variable, a and make it 0. So,
 $f = b(\sim b+c)(\sim e+f)$
- Choose $b=1, c=1$: obtain $f = (\sim e+f) \Rightarrow e=0$.
- Solution: $d=0, a=0, b=1, c=1, e=0$.

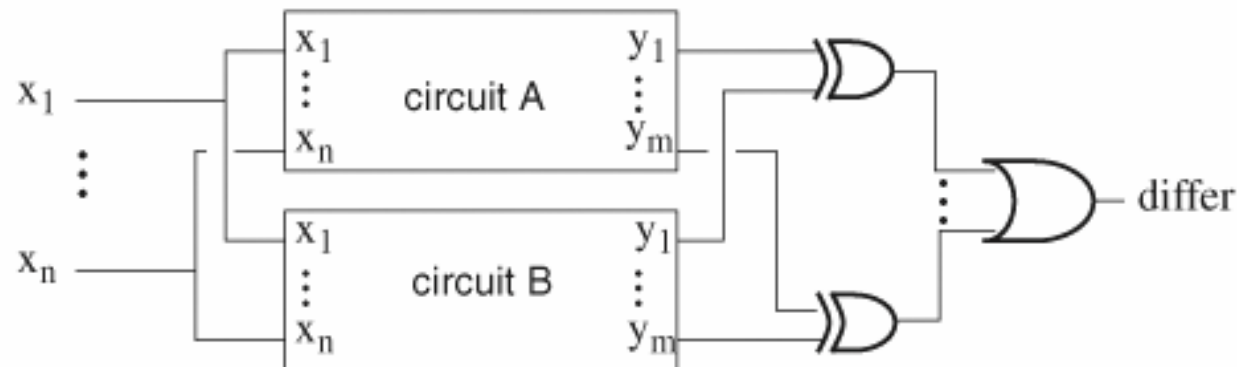


DD based Equivalence Checking

- We have seen what is meant by equivalence checking.
- Determine whether an RTL description is equivalent to its gate-level description or timing optimized version.
- Derive the ROBDD from both the circuits
- Because of canonicity of ROBDDs, the two circuits are equivalent iff their ROBDDs are isomorphic.

Eq checking for seq. circuits

- Correspondence of state bits
- Equivalence of the 2 circuits reduce to checking equivalence of the next state function





Checking equivalence

- Perform a xor operation between the two DDs.