Binary Decision Diagram (BDD)

資料結構與程式設計

Data Structure and Programming

12/09/2015

Types of Data Structures

◆ So far, we have learned:

1. Linear ADT: linked list, (dynamic) array
   - Good at insert() and erase() (at ends), but not good at find() and random erase()
2. Tree: BST, BBST, heap, set/map in STL
   - Good at insert/erase/find; can be as good as O(log n)
   - Heap is especially good for find min/max
3. Graph
   - Various applications; many established graphic algorithms
4. Hash/Cache
   - Constant time in insert/erase/find, but the data is not sorted

➔ All the above data structures are used as "container classes" to store objects.

➔ Next, we will introduce a data structure to represent "(Boolean) functions".

Function Representation and Operations

- Many of the real life problems need to deal with "function representation and operations"
  - Boolean, integer, real numbers, mixed,... etc
  - (+, -, *, %), (!, &, |, ^), differential/integral,... etc
  - Linear, nonlinear,... etc
  - ➔ We will focus on "Boolean" functions
- For computer to store and operate on Boolean functions
  - Memory efficiency --- how to store?
  - Time efficiency --- how to operate?

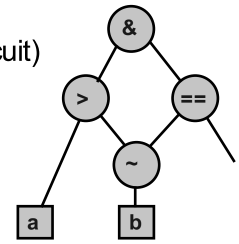
Function Representation and Operations

- Truth table
  - Tabular format
  - Karnaugh map
  - Binary decision tree
- Two-level logic
  - Sum of product (SOP), or called disjunctive normal form (DNF)
  - Product of sum (POS), or called conjunctive normal form (CNF)
- Multi-level logic
  - Boolean expression
  - Circuit netlist
- Binary Decision Diagram (BDD)

Boolean Function Representation

- In general, Boolean functions can be expressed as…
  - f = (a && b || ((c + d) > e)) ^ !g && (a > b)?...

- How to record them in computers?
  1. String? (string f = "(a && b || ((c + d) > e)) ^ !g...";)
     ➔ No, not friendly for computation
  2. Expression (syntax) tree? (e.g. circuit)
     ➔ No, not friendly for computation
     ➔ e.g. How do we know "f == g"?
     (canonicity requirement)



Enumeration Method (Truth Table)

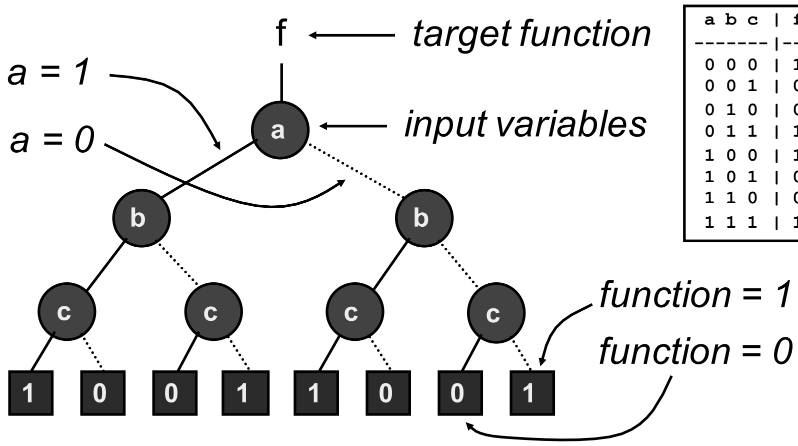
1. For each variable, enumerate its possible values
   - Usually convert a variable into Boolean variables
2. Explicitly write down all the combinations of variable values
   - e.g. 2-bit adder

```
  a1 a0  b1 b0 |  f
---------------|---
   0  0   0  0 |  0
   0  0   0  1 |  1
   .........   |
   1  1   1  0 |  1
   1  1   1  1 |  0
```

- If the truth table can be constructed, to prove the tautology or find a test vector is trivial
- However, the size of the truth table is exponential in terms of input size (like simulation)

A better data structure to represent truth table?

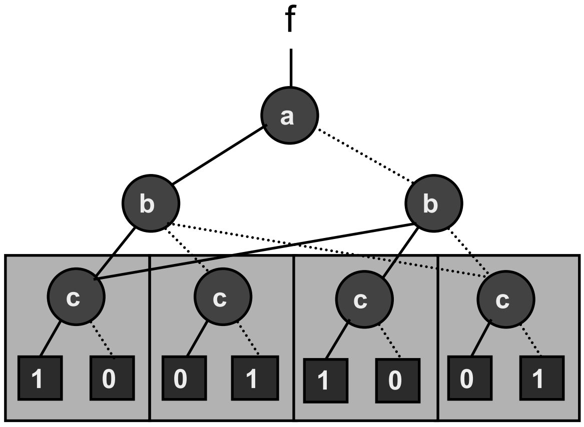
Binary Decision Tree



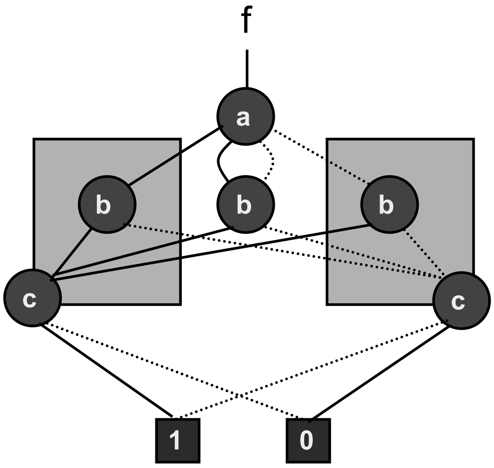
a	b	c	f
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

- Still exponential in size

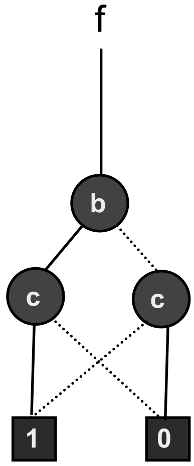
Binary Decision Diagram



Binary Decision Diagram

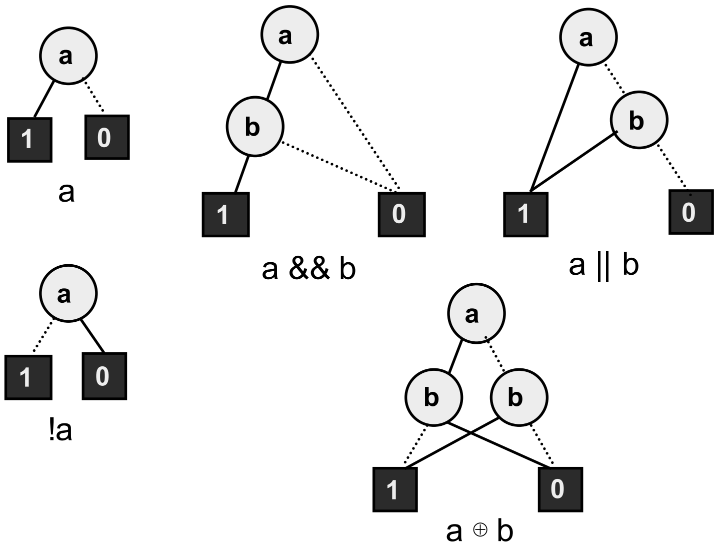


Reduced Ordered Binary Decision Diagram (ROBDD)



a	b	c	f
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

More BDD Examples



Reduced Ordered Binary Decision Diagram (ROBDD)

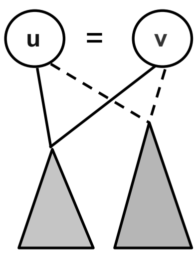
- A graphical representation of truth table
  1. Each level corresponds to an input variable
     → Set of inputs is called "<u>support</u>"
  2. Each node (and its sub-graph) represents a function
  3. Each path represents *a* *<u>cube</u>* of the function
     → i.e. an input pattern for this function
  4. Functions with equivalent functionality (sub-graph) are merged together
     - Always canonical

ROBDD Reduction Rules (1/2)

1. Uniqueness
   - No two distinct nodes u and v have the same level and left- and right-successors
   1. level(u) = level(v)
   2. left(u) = left(v)
   3. right(u) = right(v)

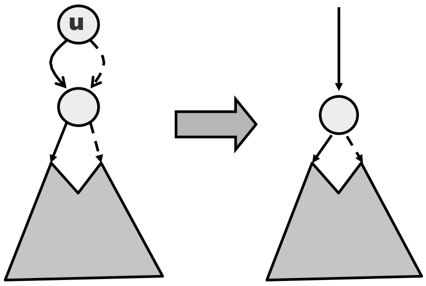
   → node(u) = node(v)

   Hash(level, left, right) → BDD node



ROBDD Reduction Rules (2/2)

2. Non-redundant tests
   - No variable node u has identical left- and right- successor.
     - For each node, left(u) ≠right(u)



ROBDD Characteristics

1. Canonicity
   - f = g iff BDD(f) = BDD(g)
   - e.g.

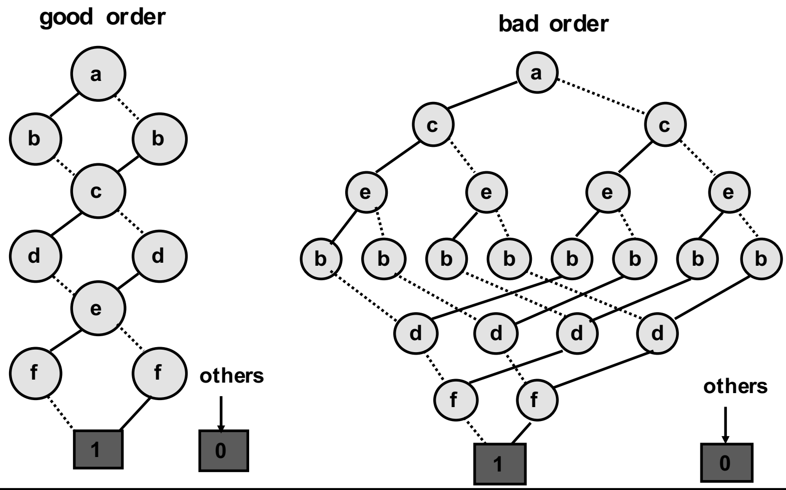
     f = ab + ac

     g = a(b+c)

     ➔ BDD(f) = BDD(g)
2. Ordering dependency
   - Given the same functionality, different input variable orderings may lead to significant difference in the number of BDD nodes

Example on Variable Ordering

$$z = (a \oplus b) \cdot (c \oplus d) \cdot (e \oplus f)$$



The Influence of Variable Ordering

- Size of BDD
  - Can vary from linear to exponential in the number of the variables, depending on the ordering
- Hard-to-Build BDD
  - Data path components (e.g., multipliers) cannot be represented in polynomial space, regardless of the variable ordering
- Heuristics of Ordering

  (1) Put variables that influence most on the top of BDD

  (2) Minimize the distance between strongly related variables

  (e.g., x1x2 + x2x3 + x3x4)

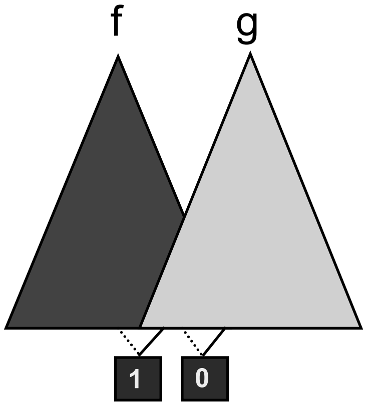
  x1→x2→x3→x4 is better than x1→x4→x2→x3
- Dynamic variable reordering (e.g. “sifting”)

Now we know BDD can be used to represent functions

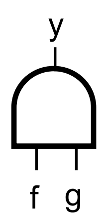
Then given a Boolean function,
➔ How to construct BDD for it?

The basic step: building BDD for a Boolean operator

Given y = AND(f, g), and BDDs for f and g ---



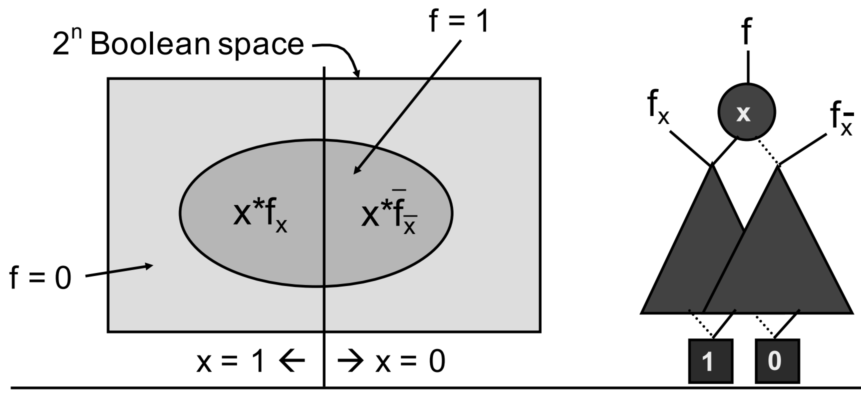
What's the BDD for y = (f && g)?



BDD(f && g) = ?

BDD Cofactors: Binary Function Decomposition

- Shannon expansion of f
  - $f = x * f_x + \overline{x} * f_{\overline{x}}$
  - ➔ f_x and $f_{\overline{x}}$ are called positive/negative cofactors



Cofactor Examples

- $f = a\overline{b}c + \overline{a}d + b\overline{d}$

➔

$f_a = \overline{b}c + b\overline{d}$ // remove a, drop $\overline{a}$ term,
// keep non-a term (i.e. $f|_{a=1}$)

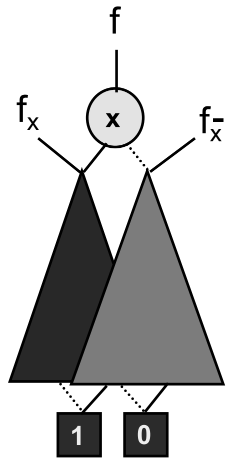
$f_{\overline{a}} = d + b\overline{d}$

$a * f_a = ?$

$\overline{a} * f_{\overline{a}} = ?$

$a * f_a + \overline{a} * f_{\overline{a}} = ?$ What is f_e?

BDD Operations

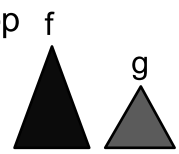


- Shannon expansion of f
  - $f = x * f_x + \overline{x} * f_{\overline{x}}$
- $f * g$
  $= (x * f_x + \overline{x} * f_{\overline{x}}) *$
  $\quad (x * g_x + \overline{x} * g_{\overline{x}})$
  $= x * (f_x * g_x) + \overline{x} * (f_{\overline{x}} * g_{\overline{x}})$
- $f + g$
  $= x * (f_x + g_x) + \overline{x} * (f_{\overline{x}} + g_{\overline{x}})$
- Operation:
  perform on <u>cofactors</u> individually ➔ Recursion

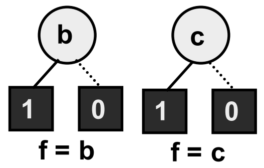
Recursive Function Operations

- Boolean operation of 2 BDDs can be performed recursively on their cofactors
  - $f * g = x * (f_x * g_x) + \overline{x} * (f_{\overline{x}} * g_{\overline{x}})$
  - $f + g = x * (f_x + g_x) + \overline{x} * (f_{\overline{x}} + g_{\overline{x}})$

  ➔ Terminal cases for this recursion are
  the operations on constants “0” and “1”
- Let u, v be the <u>top variables</u> of BDDs f, and g
  - If (u > v) // i.e. u is closer to the top
    $f * g = u * (f_u * g) + \overline{u} * (f_{\overline{u}} * g)$

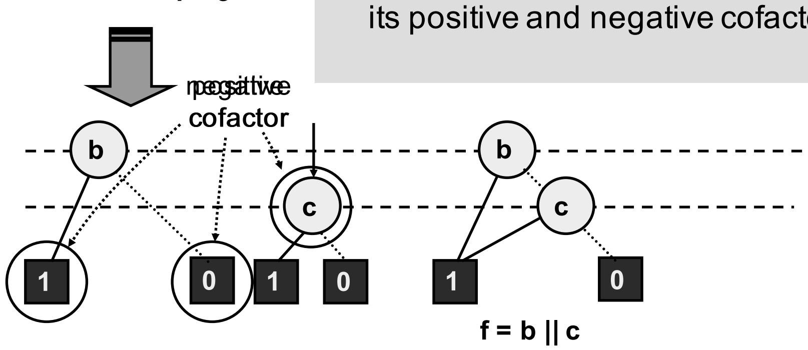


BDD Operation Example

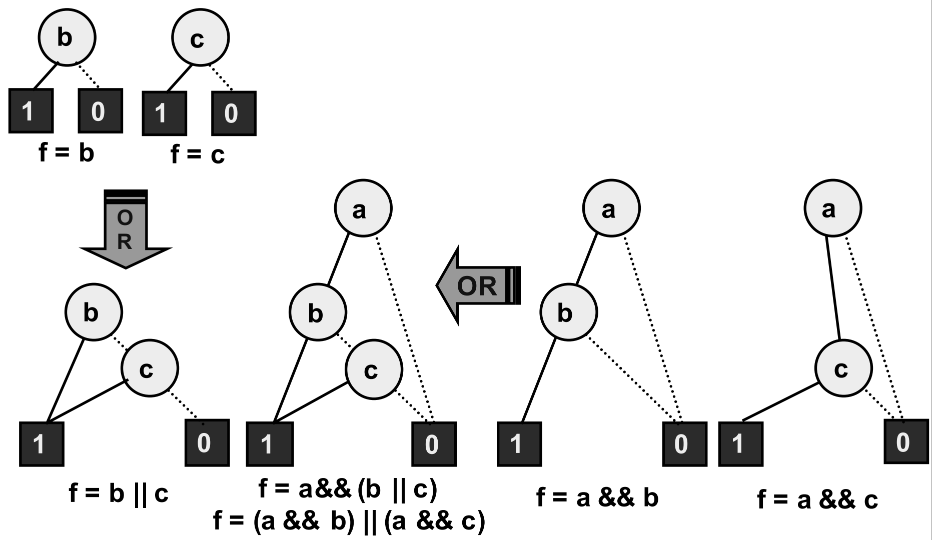


(e.g. Perform "b || c")

- Place variables in individual levels
- Start from the top variable, recursively apply the computation for its positive and negative cofactors



BDD Operation Example



In short,

1. f = a && (b || c)
2. f = (a && b) || (a && c)

➔ will result in the same BDD

➔ independent of building orders

(ROBDD canonicity characteristics)

Recursive BDD Operations on Cofactors

- Do we implement BDD operations this way?
  - (repeated computation)

    For example, $\overline{(f * g)}$ and $(\overline{f} + \overline{g})$ result in the same BDD. However, we need to recursively compute it twice...

    ➔ No "caching" effect....
  - (not good enough)

    For each new operator, need to define the evaluation of its terminal cases

Introducing the "ITE Operator" --- to enhance BDD construction efficiency

- ITE stands for "If-Then-Else"
- ITE(F, G, H) = F * G + $\overline{F}$ * H
  - e.g. For Shannon Expansion
    - f = ITE(x, f_x, $f_{\overline{x}}$) = x * f_x + $\overline{x}$ * $f_{\overline{x}}$ // x: top variable
- All unary/binary Boolean operations can be implemented by "ite" operators
  - AND(F, G) = ITE(F, G, 0)
  - OR(F, G) = ITE(F, 1, G)
  - NOT(F) = ITE(F, 0, 1)

Using ITE Operators for Boolean Functions

Table	Name	Expression	Equivalent form
0000	0	0	0
0001	AND(F,G)	$F \cdot G$	$ite(F, G, 0)$
0010	F > G	$F \cdot \overline{G}$	$ite(F, \overline{G}, 0)$
0011	F	F	F
0100	F < G	$\overline{F} \cdot G$	$ite(F, 0, G)$
0101	G	G	G
0110	XOR(F,G)	$F \oplus G$	$ite(F, \overline{G}, G)$
0111	OR(F,G)	$F + G$	$ite(F, 1, G)$
1000	NOR(F,G)	$\overline{F + G}$	$ite(F, 0, \overline{G})$
1001	XNOR(F,G)	$\overline{F \oplus G}$	$ite(F, G, \overline{G})$
1010	NOT(G)	$\overline{G}$	$ite(G, 0, 1)$
1011	F ≥ G	$F + \overline{G}$	$ite(F, 1, \overline{G})$
1100	NOT(F)	$\overline{F}$	$ite(F, 0, 1)$
1101	F ≤ G	$\overline{F} + G$	$ite(F, G, 1)$
1110	NAND(F,G)	$\overline{F \cdot G}$	$ite(F, \overline{G}, 1)$
1111	1	1	1

Using ITE to enhance BDD construction efficiency

- For example,

  BDD $(\overline{f * g})$ $= \overline{\text{ITE}(f, g, 0)}$

  BDD $(\overline{f} + \overline{g})$ $= \text{ITE}(\overline{f}, 1, \overline{g})$ $= \text{ITE}(f, \overline{g}, 1)$ $= \overline{\text{ITE}(f, g, 0)}$

  Same ITE operation!!

  ➔ DeMorgan's rule $\overline{(f * g)} = (\overline{f} + \overline{g})$

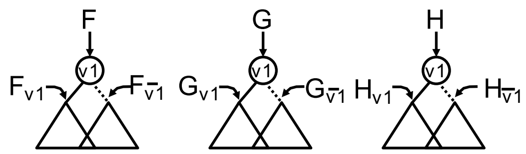
  ➔ ITE conversion rules to be covered later

How to apply ITE in BDD construction?

1. Given two BDDs, A and B
   ➔ Convert the Boolean operation on (A, B) to ITE representation, say ITE(F, G, H)
2. Normalize ITE(F, G, H) to ITE(F', G', H')
   ➔ Check: has ITE(F', G', H') been computed?
   ➔ If yes, retrieve the previously computed result
3. Otherwise, recursively compute ITE(F', G', H')
   ➔ Record the { ITE(F', G', H'), result } pair in a cache/hash

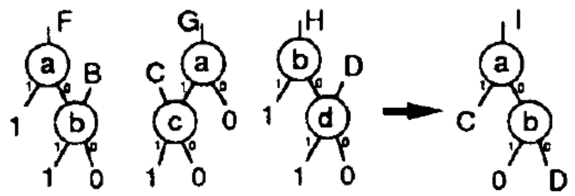
Recursive Algorithm for ITE Operation

- Let Z = ITE(F, G, H), and { v1, v2, ... } be the variable order of Z from top to bottom
  - Z = ITE(F, G, H)
    = ITE(v1, ITE(F_{v1}, G_{v1}, H_{v1}), ITE($F_{\overline{v1}}$, $G_{\overline{v1}}$, $H_{\overline{v1}}$)) // 2 cases



= ITE(v1, ITE(v2, ITE(F_{v1v2}, G_{v1v2}, H_{v1v2}), // 4 cases
ITE($F_{v1\overline{v2}}$, $G_{v1\overline{v2}}$, $H_{v1\overline{v2}}$)),
ITE(v2, ITE($F_{\overline{v1}v2}$, $G_{\overline{v1}v2}$, $H_{\overline{v1}v2}$),
ITE($F_{\overline{v1}\overline{v2}}$, $G_{\overline{v1}\overline{v2}}$, $H_{\overline{v1}\overline{v2}}$)))
= ... // until terminal cases

Example of ITE Operations



$$
\begin{aligned}
I &= ite(F, G, H) \\
&= (a, ite(F_a, G_a, H_a), ite(F_{\bar{a}}, G_{\bar{a}}, H_{\bar{a}})) \\
&= (a, ite(1, C, H), ite(B, 0, H)) \\
&= (a, C, (b, ite(B_b, 0_b, H_b), ite(B_{\bar{b}}, 0_{\bar{b}}, H_{\bar{b}}))) \\
&= (a, C, (b, ite(1, 0, 1), ite(0, 0, D))) \\
&= (a, C, (b, 0, D))
\end{aligned}
$$

Terminal Cases for ITE Algorithm

- What do the terminal cases mean?
  - ITE on constant values?
- No, actually we can terminate earlier
  - F = ite(G, F, F) = ite(1, F, G) = ite(0, G, F) = ite(F, 1, 0)

  ➔ Return F

  (FYI, the only 4 cases you need to check in your BDD pkg)
- What are NOT terminal cases?
  - ite(F, G, 0) = F*G; ite(F, G, 1) = not(F) + G
  - ite(F, 0, 1) = not(F)
  - ite(0/1, 0/1, H) ➔ covered by terminal cases
  - How about ite(F, 0, 0) and ite(F, 1, 1)??

How to apply ITE in BDD construction?

1. Given two BDDs, A and B

   ➔ Convert the Boolean operation on (A, B) to ITE representation, say ITE(F, G, H)

2. Normalize ITE(F, G, H) to ITE(F', G', H')

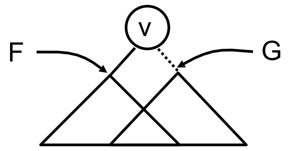
   ➔ Check: has ITE(F', G', H') been computed?

   ➔ If yes, retrieve the previously computed result

3. Otherwise, recursively compute ITE(F', G', H')

   ➔ Record the { ITE(F', G', H'), result } pair in a cache/hash

Hash/Cache in BDD constructions



1. Unique table
   v: top variable
   F, G: BddNode
   ➔ Hash(v, F, G) to an unique BddNode
   ➔ Ensure "canonicity"
2. Computed table (cache)
   F, G, H: BddNode
   ➔ Cache { ITE(F, G, H), result } to the computed BddNode
   ➔ Don't compute the same thing again!!

Complexity of ITE(F, G, H) can be as good as O(|F|*|G|*|H|)
(Why??)

Remember: using ITE to enhance sharing

- For example,

BDD ($\overline{f * g}$) $= \overline{ITE (f, g, 0)}$

BDD ($\bar{f} + \bar{g}$) $= ITE (\bar{f}, 1, \bar{g})$ $= ITE (f, \bar{g}, 1)$ $= \overline{ITE (f, g, 0)}$

Same ITE operation!!

➔What about ITE(F, 0, 1) and ITE(F, 1, 0)?
➔Do they share the same sub-BDD nodes?

Improving the BDD node sharing

- Complexity to compute $\overline{F}$?

➔ Fact: F and $\overline{F}$ are different nodes in BDD

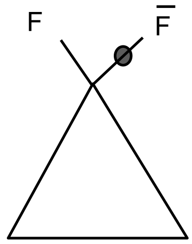
- NOT(F) = ite(F, 0, 1) is not a terminal case
  - Need to go through every path of BDD(F) and at the end, construct BDDs with interchanged terminal nodes (0, 1)

- Any better way?

  ➔ Complement edge

BDDs with Complement Edges

- A complement edge is a flagged edge which denotes the function is complemented (inverted)
  - Usually use a "bubble" to denote the inverted edge



  - [Note] Only constant "1" node is kept; constant "0" is denoted with complement edge
- Can we still guarantee the canonicity?

Equivalence with Complement Edges

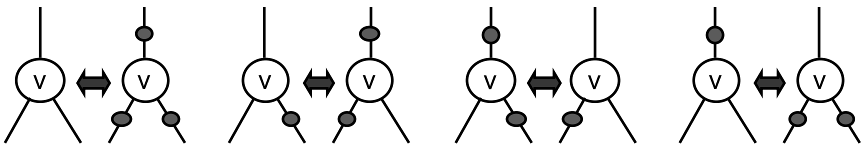
- $(v, F_v, F_{\bar{v}}) = v * F_v + \bar{v} * F_{\bar{v}}$

$$= \overline{\overline{v * F_v + \bar{v} * F_{\bar{v}}}}$$

$$= \overline{(\bar{v} + \overline{F_v}) * (v + \overline{F_{\bar{v}}})}$$

$$= \overline{v * \overline{F_v} + \bar{v} * \overline{F_{\bar{v}}}}$$

$$= \overline{(v, \overline{F_v}, \overline{F_{\bar{v}}})}$$



Canonicity with Complement Edges

- What's the problem?
  - If we allow $(v, F_v, F_{\bar{v}})$ and $(v, \overline{F_v}, \overline{F_{\bar{v}}})$ to co-exist in BDDs, they will have different hash keys and thus hash to different nodes (why?)

  ➔ But by using complement edges, they should point to the same node!!

- Solution
  - The "then" child should NOT have bubble

  ➔ If happens, apply the rules in the previous page

Improving ITE Cache Hit Rate

- Observation
  - There may be ITE(F1, F2, F3) = ITE(G1, G2, G3), where F_i != G_i for some i
    - e.g. ITE(F, G, 0) = $\overline{\text{ITE}(\overline{F}, 1, \overline{G})}$ // AND(F, G) = NOR($\overline{F}$, $\overline{G}$)
    - e.g. F + G
      ITE(F, 1, G) = ITE(G, 1, F) = ITE(F, F, G) = ITE(G, G, F)

  ➔ ITE function may be recursively called many times and return the same result
- Objective
  - Rearrange the ITE parameters so that the cache hit rate of the computed table can be higher

  [Keypoint] Think how the computed cache works!!

Equivalent ITE Operations

- Identical parameters ➔ Constant parameter
  - ite(F, F, G) ➔ ite(F, 1, G) // F + G
    ite(F, G, F) ➔ ite(F, G, 0) // F * G
    ite(F, G, $\overline{F}$) ➔ ite(F, G, 1)
    ite(F, $\overline{F}$, G) ➔ ite(F, 0, G)
- Symmetrical parameters
  - ite(F, 1, G) = ite(G, 1, F)
    ite(F, G, 0) = ite(G, F, 0)
    ite(F, G, 1) = ite($\overline{G}$, $\overline{F}$, 1)
    ite(F, 0, G) = ite($\overline{G}$, 0, $\overline{F}$)
    ite(F, G, $\overline{G}$) = ite(G, F, $\overline{F}$)
- Complement parameters
  - ite(F, G, H) = ite($\overline{F}$, H, G) = $\overline{\text{ite}(F, \overline{G}, \overline{H})}$ = $\overline{\text{ite}(\overline{F}, \overline{H}, \overline{G})}$

Which one to choose??

Equivalent ITE Operation Rules

1. If contains identical or complement parameters
   ➔ Reduce to constant parameter
2. If contains symmetrical parameters
   ➔ The first parameter is given with the smallest "top" variable;
   ➔ If tied, choose the one with smaller pointer address
3. If contains complement edge parameters
   ➔ The first and second parameters cannot be complement edges

Recursive Algorithm for ITE Operation

```
ite(F, G, H) {
   Standardize parameters (F, G, H);
   if (terminal case)
      return result;
   if (ite(F, G, H) in computed table)
      return result;
   let v be the top variable;
   T = ite(Fv, Gv, Hv);
   E = ite(Fv̄, Gv̄, Hv̄);
   Process complement edge info for T & E
   if (T == E) return T;
   if ((v, T, E) in the unique table)
      return result;
   let R = BddNode(v, T, E);
   insert R into unique table;
   return R;
}
```

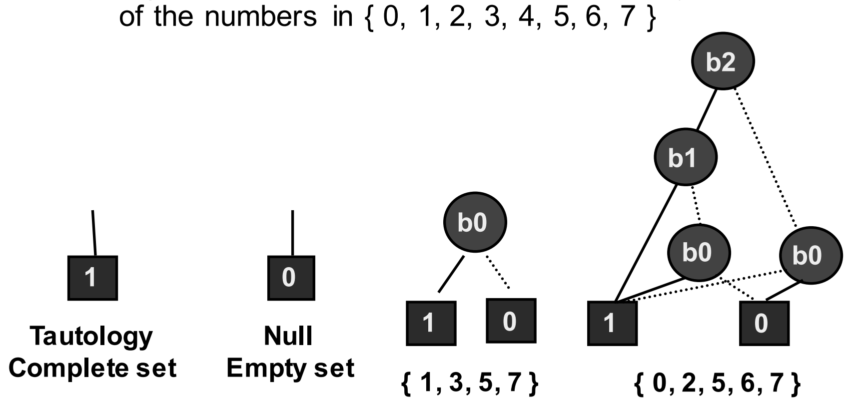
Other than logic functions,

BDDs can also be used to represent

“Sets” and “Relations”

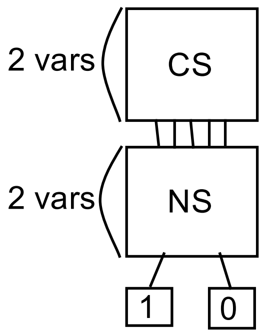
BDD to Represent a Set

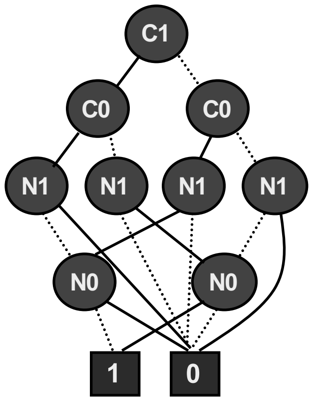
- Other than Boolean function, BDD can also represent sets over Boolean variables
  - e.g. Use 3-variable BDDs to represent any subset of the numbers in { 0, 1, 2, 3, 4, 5, 6, 7 }



BDD to Represent Relations

- e.g. 2-bit ring counter: NS = (CS + 1) % 4
  R: { (0 → 1), (1 → 2), (2 → 3), (3 → 0) }





Other types of decision diagrams

- ZDD
- FDD
- MTBDD(ADD)
- BMD

Zero-Suppressed BDD (ZDD)

- However, BDD may not be good for sparse set representation, set cover/union operation, etc
  ➔ ZDD can do better

- Proposed: S. Minato, DAC 93
- A good tutorial:
  - "An Introduction to Zero-Suppressed Binary Decision Diagrams", Alan Mishchenko, 2001

BDD vs. ZDD

- BDD: function representation
- ZDD: set cover representation
  ➔ Conversion between BDD and ZDD is easy
- Remember, BDD has two reduction rules
  1. Uniqueness: hash(level, left, right)
  2. Non-redundant test: eliminate the node whose positive and negative children pointing to the same node

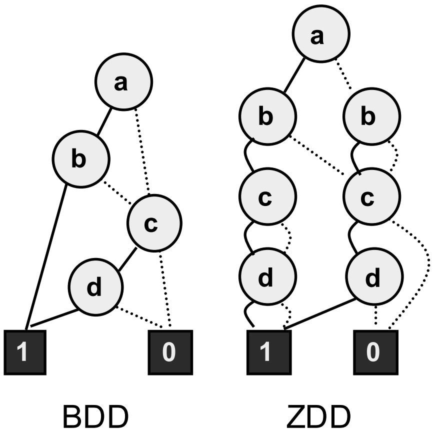
  ➔ ZDD does NOT have (2)
- Instead, ZDD's non-redundancy rule
  - Remove the node whose "positive" edge pointing to a constant '0' node

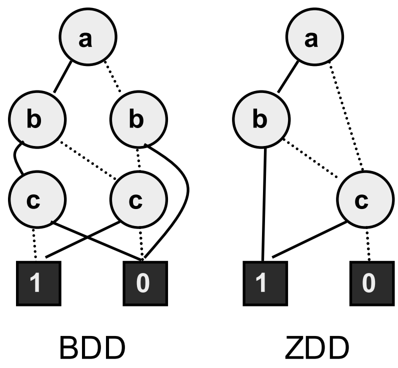
ZDD Examples

◆ Function: ab + cd

◆ set of subsets:
{ {a,b}, {a,c}, {c} }

➔ **characteristic function:**
{ (abc) | (110), (101), (001) }





Intuitions on the ZDD non-redundancy rule

◆ If we treat the “cubes” in BDDs as strings of 1/0’s, what does ZDD “suppress” in its representation?
- e.g. characteristic function:
  { (abc) | (110), (101), (001) }

◆ The suppressed ZDD nodes are ---
1. Ending 0’s
2. Consecutive 0’s, maybe except for the leading 0
   - e.g. 100010010 ➔ 1(0)--1(0)-1-

Complexity in Representing Set of Subsets

- ZDD upper bound
  - Total number of elements appearing in all subsets of a set
- BDD upper bound
  - The number of subsets multiplied by the number of all elements that can appear in them

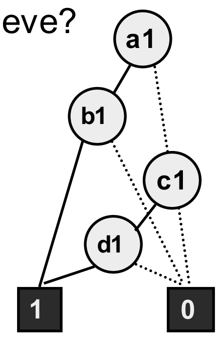
ZDD to Represent Cube Covering

- Goal: use DDs to record the cubes and perform operations (union/intersect, etc) on them
  - E.g. F = ab + cd has 2 cubes
  - How does BDD represent them?
  - Are they same cubes when we retrieve?
- Approach:
  - Each input is split into 2 variables ➔ positive and negative literals

  ➔ Char. function for cover { ab, cd }

  $\chi = a1\overline{a0}b1\overline{b0}\,\overline{c1}\,\overline{c0}\,\overline{d1}\,\overline{d0}$
  $+ \overline{a1}\,\overline{a0}\,\overline{b1}\,\overline{b0}c1\overline{c0}d1\overline{d0}$
  $= 10100000 + 00001010$

ZDD

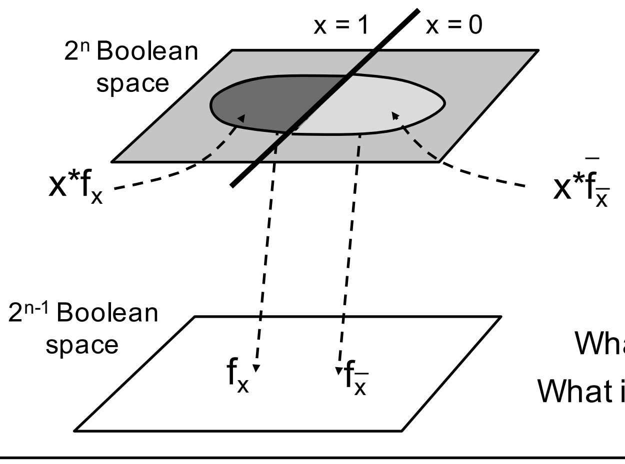


Other types of decision diagrams

- ZDD
- FDD
- MTBDD(ADD)
- BMD

Something about co-factors...

- Fallacy: $f_x \wedge f_{\bar{x}} = \emptyset$



What is $x*f_x$?

What is ($f_x \oplus f_{\bar{x}}$) ?

Reed-Muller Expansion (ExOR-polynomial)

- Let $f(x, Y) = x \cdot f_x(Y) + \overline{x} \cdot f_{\overline{x}}(Y)$
- Let $f_1(Y)$, $f_2(Y)$ be some functions of Y. Then f can be decomposed as $f_1 \oplus f_2 \cdot x$
  - What are f_1 and f_2?
  - What do they mean? What does $\oplus$ mean?
- Rearrange them, we have: $f = x \cdot (f_1 \oplus f_2) + \overline{x} \cdot f_1$
  - So, what are f_1 and f_2?
- Recursively decompose f_1 and f_2, we have:
  - $f_1 = f_3 \oplus f_4 \cdot y$ $\quad f_2 = f_5 \oplus f_6 \cdot y$
  - $f = f_1 \oplus f_2 \cdot x = a_1 \oplus a_2 \cdot x \oplus a_3 \cdot y \oplus a_4 \cdot xy$

Function Decomposition

- Shannon Expansion (sum of product form)
  - $f = x\, f_x + \overline{x}\, f_{\overline{x}}$
- Reed-Muller Expansion (ExOR-polynomial)
  - $f = a_0 \oplus a_1 \cdot x_0 \oplus a_2 \cdot x_1 \oplus a_3 \cdot x_0 \cdot x_1 \oplus \ldots$
    $\oplus\, a_{2^n-1} \cdot x_0 \cdot x_1 \cdot x_2 \cdot x_3 \ldots \cdot x_{n-1}$

  [In recursive form]
  - $f = f_{\overline{x}} \oplus (f_x \oplus f_{\overline{x}}) \cdot x$ (positive davio)
    $= f_x \oplus (f_x \oplus f_{\overline{x}}) \cdot \overline{x}$ (negative davio)

  → f = (negative cofactor) ⊕ (Boolean difference) · x
  = (positive cofactor) ⊕ (Boolean difference) · $\overline{x}$

[Sideline Help] Extracting Cofactors

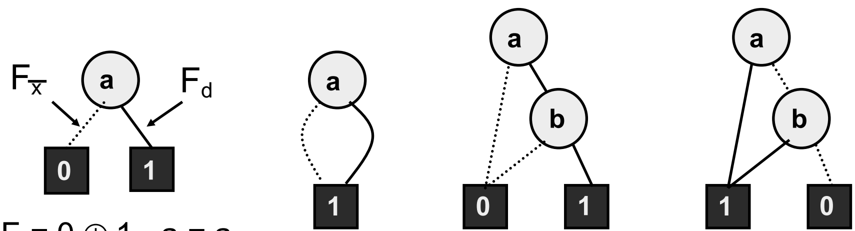
- Let $F = x \cdot A + \overline{x} \cdot B + C$
  - ➔ positive cofactor: $F_x = A + C$
  - ➔ negative cofactor: $F_{\overline{x}} = B + C$
  - ➔ Boolean difference: $F_d = F_x \oplus F_{\overline{x}} = ??$
    $F_d = (A \oplus B) \cdot \overline{C}$

Any intuitive explanation?

Ordered Functional Decision Diagram

[OFDD, Kebschull, EDAC 92]

- Structurally similar to BDD, but each node use "positive (or negative) davio to explain
  - Still canonical
  - Good for XOR-like circuits



$F = 0 \oplus 1 \cdot a = a$

??? ??? ???

Other types of decision diagrams

- ZDD
- FDD
- MTBDD(ADD)
- BMD

DDs for Arithmetic Manipulation

- BDD, FDD, OKFDD, etc are useful in verifying Boolean functions
- What if the property contains word-level signals and arithmetic operations?
  - e.g. 8a + 5ab = 20
  - Word-level signals → decomposed into bits
- e.g. A truth table for arithmetic functions

```
an... a2 a1   bm... b2 b1  | f
---------------------------|---
 0...  0  0    0...  0  0  | 12
 0...  0  0    0...  0  1  | -8
 ..........................| ..
 1...  1  1    1...  1  1  |  7
```

→ Decompose 'f' into bits and use multiple functions (BDDs) for operation ??

- f_0(an,..., a2, a1, bm,.., b2, b1), f_1(an,..., a2, a1, bm,.., b2, b1),...

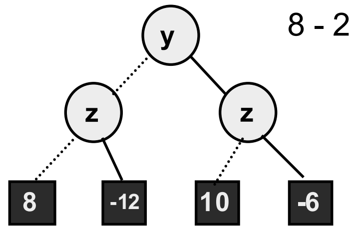
Multi-Terminal BDD (also called ADD)

◆ Expanding the terminal of BDDs to multi-value nodes

e.g. $8 - 20z + 2y + 4yz$, where y and z are Boolean variables

y	z	F
0	0	8
0	1	-12
1	0	10
1	1	-6

MTBDD



$8 - 20z + 2y + 4yz$

Boolean Movement Diagram (BMD)

[Bryant, DAC95]

◆ Just like FDD to BDD, where we use "Reed-Muller" (Boolean difference) instead of "Shannon Expansion"

➔ We can apply "Reed-Muller Expansion" on MTBDD too

◆ Let x be Boolean ➔ $\bar{x} = (1 - x)$

➔ $f = x \cdot f_x + \bar{x} \cdot f_{\bar{x}} = x \cdot f_x + (1 - x) \cdot f_{\bar{x}}$

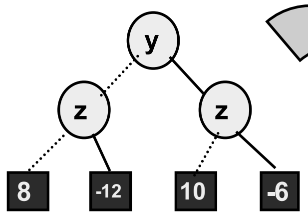
$= f_{\bar{x}} + x \cdot (f_x - f_{\bar{x}})$ ← postive davio form

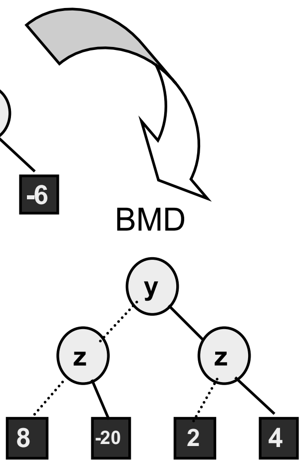
From MTBDD to BMD

Function

y	z	F
0	0	8
0	1	-12
1	0	10
1	1	-6

MTBDD





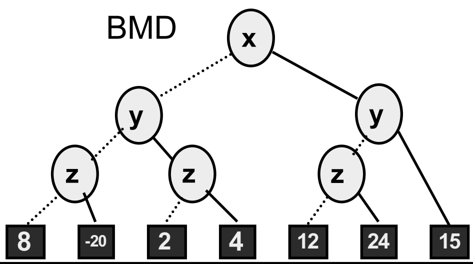
$$F(y, z) = \begin{bmatrix} 8\ (1-y)(1-z) + \\ -12\ (1-y)\ z + \\ 10\ y\ (1-z) + \\ -6\ y\ z \end{bmatrix} \begin{matrix} ...(00) \\ ...(01) \\ ...(10) \\ ...(11) \end{matrix}$$

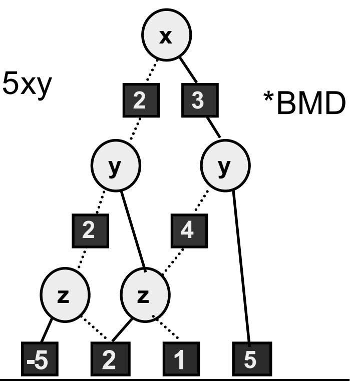
$$= 8 - 20z + 2y + 4yz$$

*BMD

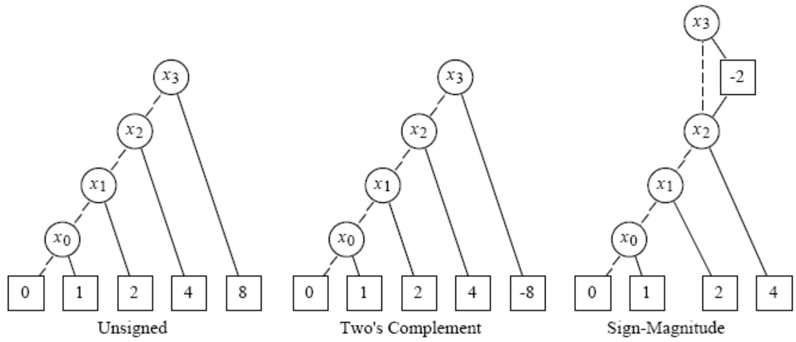
- Similar to EVBDD, where an integer value is annotated on the edge as the weight
  ➔ We can factor out the terminal nodes and lift the common divisors to the edge values

Function:
$8 - 20z + 2y + 4yz + 12x + 24xz + 15xy$

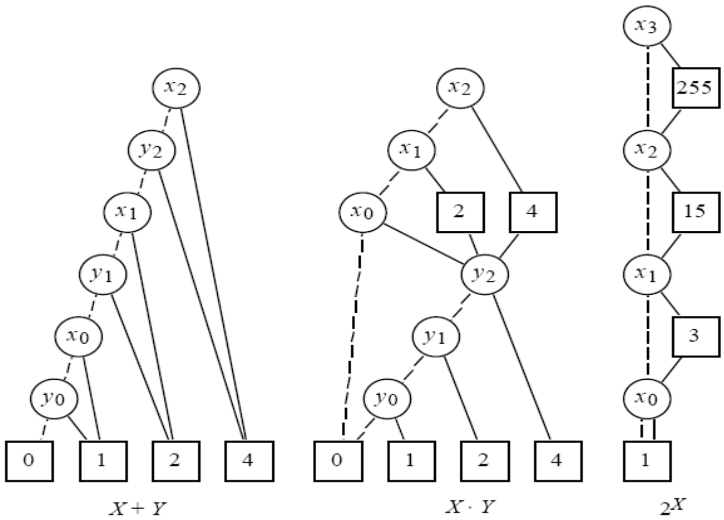




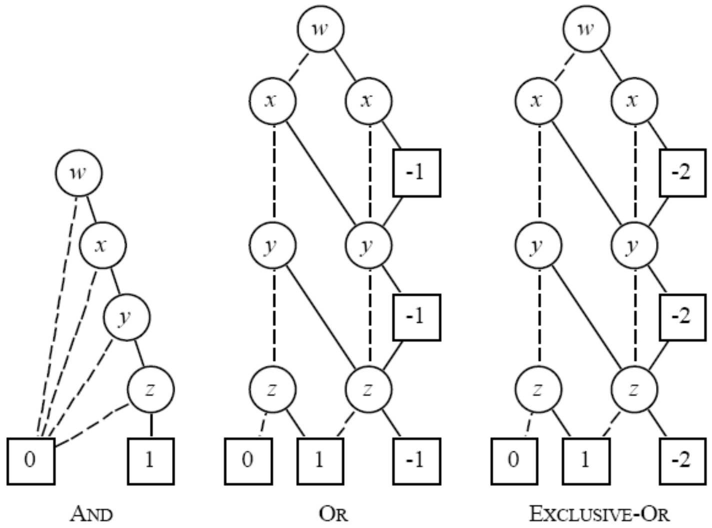
*BMD for Representation of Integers



*BMD for Word-Level Sum, Product, and Exponentiation

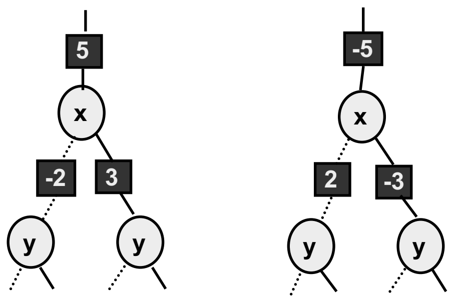


*BMD for Boolean Functions



Canonicity for BMC*

- Are these the same?



➔ Need rules to ensure the canonicity!

➔ Negative edge value on “pos-edge” only

Conclusion on various DDs

- We have seen various types decision diagrams
  - BDD
  - FDD
  - OKFDD
  - MTBDD (ADD)
  - EVBDD
  - BMD
  - *BMD

  ➔ There are more that are not covered in this class...
- A comprehensive study on the comparison / classification of the various DDs can be found at:
  - "Decision Diagrams for Discrete Functions: Classication and Unied Interpretation", Stankovic and Sasao, ASPDAC 98

Computing Cofactors on BDD

- Given a function f

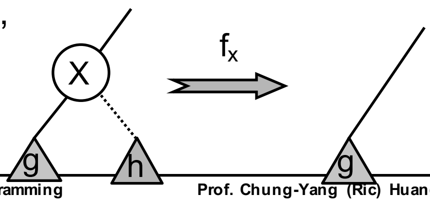
  ➔find its positive/negative cofactor f_x / $f_{\bar{x}}$
  - e.g. Let f = a $\bar{c}$ + b c

    ➔ f_c = b

    ➔ $f_{\bar{c}}$ = a

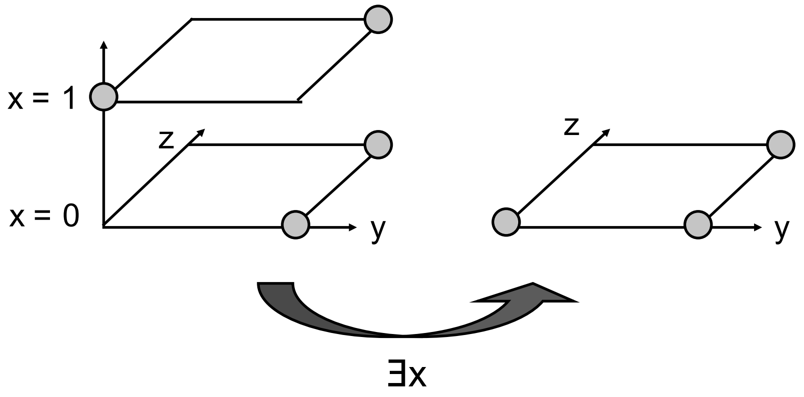
    ➔ $f_{\bar{a}}$ = b c
- If x is top variable

  ➔ cofactors = left and right children
- Otherwise,



Existential Quantification

- $\exists x.f = f_x + f_{\bar{x}}$ (← What does this mean?)



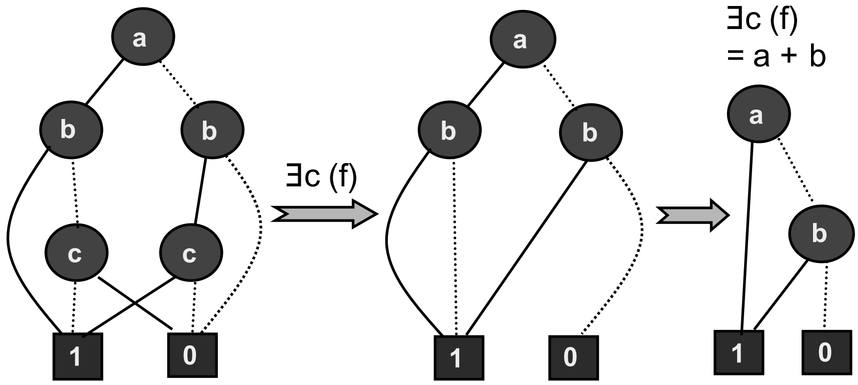
Existential Quantification on BDD

- $\exists x (f) = f_x + f_{\bar{x}}$ (← How to perform it on BDD?)
- If x is top variable
  → Perform an "OR" on its cofactors
- If x is bottom variable
  → Replace non-zero nodes with '1' (why?)
- If x is middle variable
  → ???

Which one is better??

Existential Quantification

- $\exists x\,(f) = f_x + f_{\bar{x}}$
  - e.g. $f = a\,b + a\,\bar{b}\,\bar{c} + \bar{a}\,b\,c$



Cofactors, Boolean Difference, Existential Quantification

- Let $F = x \cdot A + \bar{x} \cdot B + C$

1. Cofactors
   - $F_x = A + C$
   - $F_{\bar{x}} = B + C$
2. Boolean difference
   - $F_d = (A \oplus B) \cdot \bar{C}$
3. Existential quantification
   - $\exists x.F = A + B + C$

- What if the formula is represented in PoS form?

References

1. (Original BDD paper) R. E. Bryant, "Graph-based algorithms for Boolean function manipulation", IEEE Trans. on Comp., 35(8):677-691, 1986.
2. (BDD implementation classic) K. S. Brace, R. L. Rudell, and R. E. Bryant. “Efficient implementation of a BDD package”. In Proceedings of the 27th Design Automation Conference, pages 40-45, Orlando, FL, June 1990.

A Exemplar BDD Package: CUDD

- Implemented in University of Colorado at Boulder
  - Prof. Fabio Somenzi
- The most widely used BDD package in various researches and EDA tools
- http://vlsi.colorado.edu/~fabio/CUDD/cuddIntro.html

BDD Complexity

- In general, the size of BDD nodes is still exponential to the size of input supports
  - Usually can only build BDDs for circuit with #input = 100 ~ 200

    (But consider $2^{100} \sim 2^{200}$)

- Many many optimization techniques, and different variations of DDs… not to be covered in this class...

  (GIEE: “SoC Verification”)