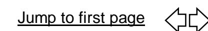
Boolean Function Representation and Manipulation

Ordinary Binary Decision Diagram

Chapter 6

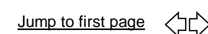
Jump to first page



BDD – Based Upon Shannon Expansion

- Notations
  - $f(x_1, x_2, \ldots, x_n)$ - n-input function, $x_i = 0$ or 1
  - $f_{|xi=b}(x_1, \ldots, x_n) = f(x_1,\ldots,x_{i-1},b,x_{i+1},\ldots,x_n)$, b=0 or 1
- Shannon Expansion – fix a variable
  - $f = \mathbf{x_i}\ f_{|xi=1}(x_1, \ldots, x_n) + \mathbf{x'_i}\ f_{|xi=0}(x_1, \ldots, x_n)$

Jump to first page



Example

- f = x1 x2 + x3 x4 + x1’ x2 x5
  - f(x1=0) = x3 x4 + x2 x5
  - f(x1=1) = x2 + x3 x4
  - f = x1’ (x3 x4 + x2 x5) + x1 (x2 + x3 x4)
- Similarly
  - f = x2’ (x3 x4) + x2 (x1 + x3 x4 + x1’ x5)
- To represent f, we use a “divide-and-conquer” approach to represent f(xi=0) and f(xi=1)

xi

f(xi=0) f(xi=1)

Li-C. Wang’s Lecture

Jump to first page

Decision Tree

- f = x1 x2 + x3
- f = x1 (x2 + x3) + x1’ (x3)

g1 g2

- g1 = x2 (1) + x2’ (x3)
- g2 = x3

x2 x3 1 0 1

g1

x3 0 1

g2

x1 x3 x2 0 1 x3 1 0 1

f

In **f**, g2 appears twice, 1 appears 3 times, and 0 appears twice

Li-C. Wang’s Lecture

Jump to first page

OBDD

- From root (x1) to a terminal vertex (0 or 1), the ordering of *the splitting variables* is the same
- Each sub-function appears only once
  - two sub-functions are the same iff their OBDD graphs are the same ⇔ canonical representation

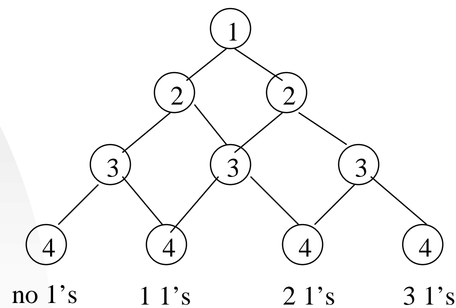
Classical OBDD Example

- 2n+1 vertices
- Variable ordering is not important

$f = x1 \oplus x2 \oplus ... \oplus xn$

Classical OBDD Example

- To represent a symmetric function
  - variable ordering is not important
  - at level i, we need at most i vertices
  - total nodes is < $O(n^2)$



no 1's 1 1's 2 1's 3 1's

Li-C. Wang's Lecture

Jump to first page

Ordering Example

- f = x1 x2 + x3 x4 + x5 x6
  - ordering 1 : x1 x2 x3 x4 x5 x6 (linear result)
    - If (x1 x2)= 1, f = 1 ⇒ no need to read further!
    - Otherwise we can "forget about x1 x2"
    - The "*memoryless effect*" makes BDD small
      - A node in a BDD is a memory cell to remember the intermediate results during the computation of the function
  - ordering 2 : x1 x3 x5 x2 x4 x6 (exponential result)
    - If (x1=x3=x5=1) ⇒ we need to remember them all!
    - Because nothing can be decided based upon the information we have

Li-C. Wang's Lecture

Jump to first page

Ordering and OBDD Size

- A node in a function graph is a place to remember important information about the computation
- $f = x_1 x_2 + x_3 x_4 + \ldots + x_{(n-1)} x_n$
  - *proceed in this direction* $\longrightarrow$
- when reach x_2
  - need to remember the value of x_1
- when reach x_3
  - only when $x_1 x_2 = 0$
  - if $x_1 x_2 = 0$, then $x_1 x_2$ can be ignored and from x_3 the computation restarts *fresh*

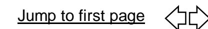
OBDD Size

- $f = x_1 x_{n/2+1} + x_2 x_{n/2+2} + \ldots + x_{n/2} x_n$
- proceed ordering as $x_1 x_2 x_3 x_4 \ldots x_{(n-1)} x_n$
- when reach $x_{n/2}$
  - all values in $x_1 x_2 x_3 \ldots x_{(n-1)/2}$ will need to be remembered because a different assignment may lead to different result
  - nothing can be ignored and nothing for sure to be said about the evaluation yet
  - need a complete binary tree for (n-1)/2 variables
    - exponential result

Worst-Case Example

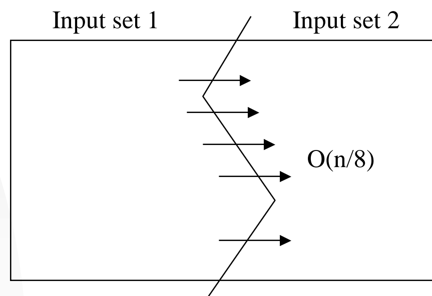
- Worst-case example - multiplication
  - f = A*B
  - for any input ordering, OBDD(f) requires an exponential number of nodes
- $(a_n\ a_{n-1}\ \ldots\ a_1) * (b_n\ b_{n-1}\ \ldots\ b_1)$
  - π is an ordering
  - Theorem: for any π, OBDD size $\geq 2^{n/8}$
- Potential solutions
  - word-level BDD
  - Binary moment diagram (BMD)

Li-C. Wang's Lecture

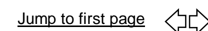


Idea of Proof

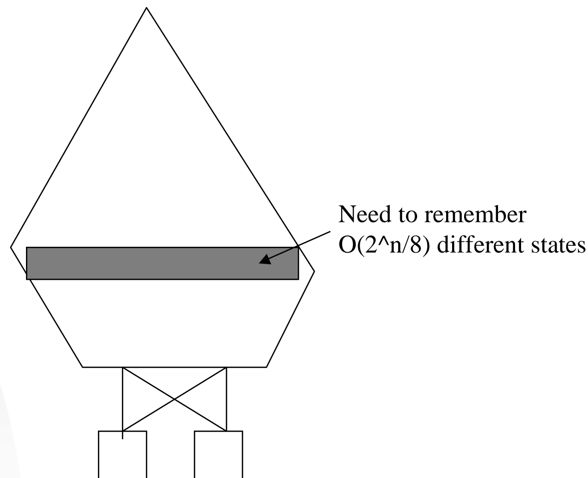
- No matter how you order the input variables, you can always find a cut that
  - The width is O(n)
  - The information flow is O(n/8)



Li-C. Wang's Lecture



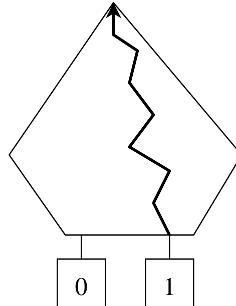
Equivalent to Say



More Terminology

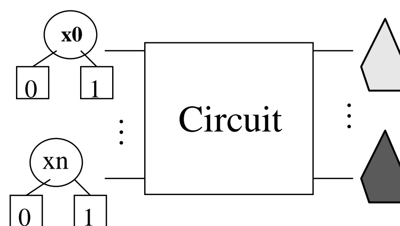
- Composition
  - f(xi=g)(x1…xn) = f(x1,…, g(x1…xn), …, xn)
  - ex. f=x1x2+x3x4 and g=x2+x3
  - f(x1=g) = x2 + x2x3 + x3x4 = x2 + x3x4
- Dependency Set
  - I_f = { i | f(xi=0) ≠ f(xi=1) } ⇒ f depends on xi
  - f = x1 x2 x3 + x1' x2 x3 ⇒ f is independent of x1
- Satisfying Set
  - S_f = { (x1 … xn) | f(x1 … xn) = 1 }
  - all assignments that made f = 1

Satisfiability



- Satisfy-one: find a path from 1 to the root
- Satisfy-all : find all different paths from 1 to the root
- Satisfy-count : count the # of satisfying assignments

Computing OBDDs



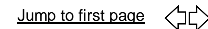
- Initialize all Pis
- Compute the result of a boolean operation as merging two OBDDs
  - and, or, xor, etc.
  - Use procedures *Apply* and *Reduce*
- Follow topological order until reaching all POs

OBDD Operation Complexity

- Reduce (make canonical, minimal form) - O(G log G)
  - (DA) graphs isomorphism
- Apply (merge two OBDDs) - $O(G_1 G_2)$
  - Boolean operation = {Apply ⇒ Reduce}
- Restrict {f(xi=b)} - O(G log G)
- Compose {f(xi=g)} - $O(G_1^2 G_2)$
- Satisfy-One - O(n)
- Satisfy-all - $O(n |S_f|)$
- Satisfy-Count - O(G)
  - This is a #P-complete problem

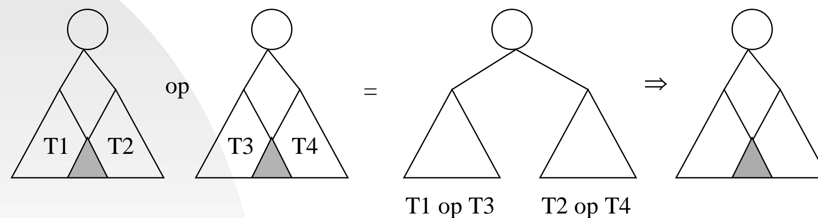
Li-C. Wang's Lecture

Jump to first page



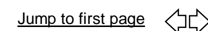
Apply

- f_1 *op* f_2 =
  - x_i' [$f_1(x_i=0)$ *op* f_2 $(x_i=0)$] + x_i [$f_1(x_i=1)$ *op* $f_2(x_i=1)$]



Li-C. Wang's Lecture

Jump to first page



Example

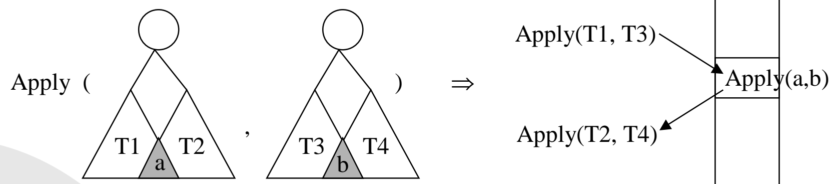
1 3 0 1 NAND 2 3 0 1 = 1 2 3 1 0

Complexity of Apply

i nodes op j nodes = k nodes

- $k \leq i * j$
  - i nodes represent i different sub-functions
  - the worst case - (each of i nodes) op (each of j nodes) are different
  - Total complexity O(G1 G2)

Hash Table in Apply



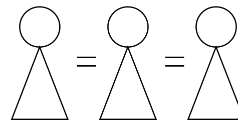
- When doing Apply(T1,T3), we have done Apply(a,b)
  - Apply(a,b) should not be repeated
- Need a hash table to store results of from Apply of all sub-functions.

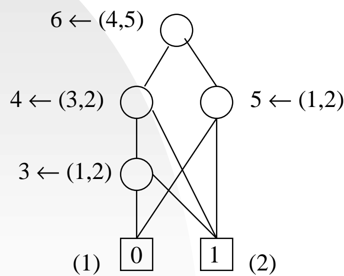
Li-C. Wang's Lecture

Jump to first page

Reduce

- DA Graph Isomorphism
  - we can do a pair-wise comparison for all nodes
  - but that is too slow
    - instead: by labeling bottom-up





Each graph has a unique labeling scheme
⇒ same label at the root = same graph

Li-C. Wang's Lecture

Jump to first page

Labeling Scheme

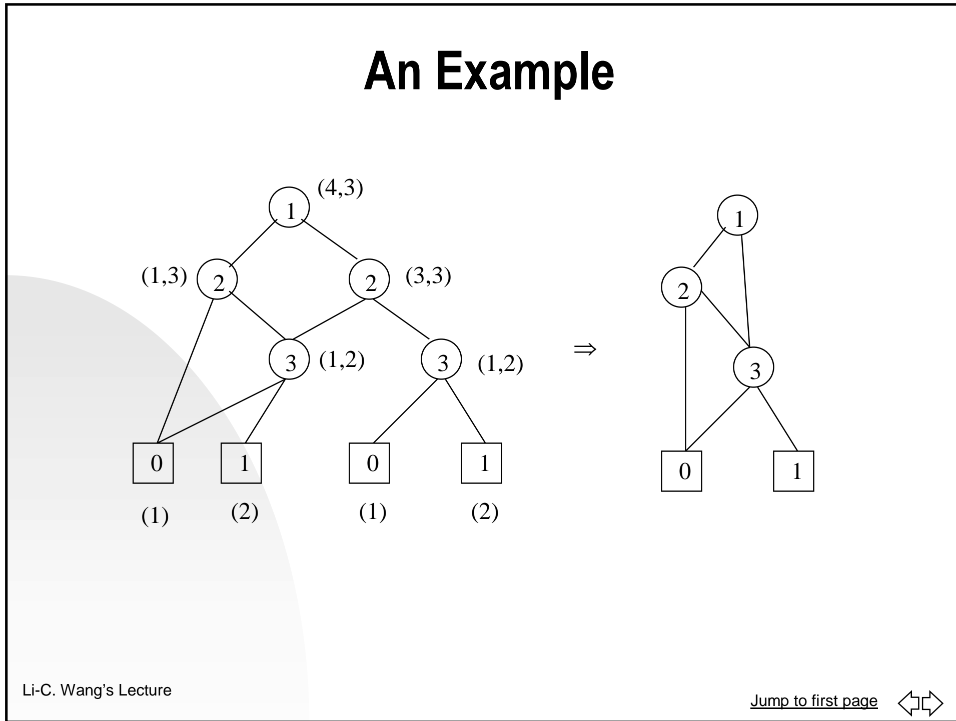
i i ... i
i+1 i+1 ... i+1
i+2 i+2 ... i+2
⇒
i = i
a b c d
iff a=c && b=d

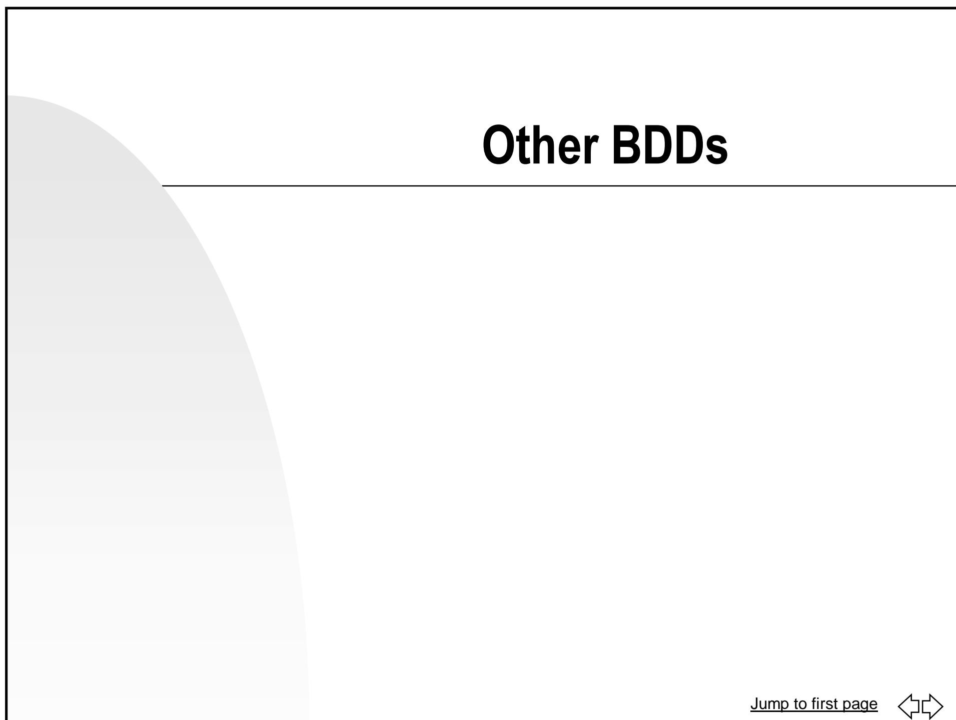
- Why working bottom-up?
  - So that when working at level i,
    - each node below already points to a unique graph already

Redundant Nodes Removal

i i i i
i+1 = i+1 ⇒ i+1
i-1
i
i+1 i+1
=
⇒
i-1
i+1

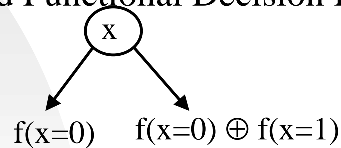
- At each level, how do we know how many nodes are identical in the labeling scheme?
  - by sorting the labels at that level
  - O(n log n) time complexity


An Example
(4,3)
1
(1,3)
2
2
(3,3)
3
(1,2)
3
(1,2)
⇒
0
1
0
1
(1)
(2)
(1)
(2)
1
2
3
0
1
Li-C. Wang's Lecture
Jump to first page


Other BDDs
Jump to first page

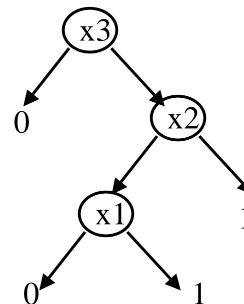
Function Decomposition Rules

- OBDD
  - $f = x' f(x=0) + x f(x=1)$
- Reed-Muller Decomposition
  - $f = f(x=0) \oplus x [f(x=0) \oplus f(x=1)]$
  - $f = f(x=1) \oplus x [f(x=0) \oplus f(x=1)]$
- Ordered Functional Decision Diagram



OFDD

- $F = (x1 \oplus x2) x3$
  - $F(x3=0) = 0$
  - $F(x3=1) = x1 \oplus x2$
    - difference $D3 = x1 \oplus x2$
  - $D3(x2=0) = x1$
  - $D3(x2=1) = x1'$
    - difference $D2 = 1$



- OFDD requires a different Reduction rule
- For some functions, OFDD can be exponentially smaller than OBDD and vice verse
- Evaluation of the graph is different from OBDD
  - Can no longer be simply tracing from root to 0's or 1's

Free BDD

- The ordering along each path can be different
- Main issues
  - Deciding that two trees are representing the same functions can not be done by graph isomorphism.
  - It can only be done via a probabilistic algorithms
- Free BDD give flexibility in the representation but sacrifice in the complexity of manipulate the BDD date structures

Represent Numeric-Valued Functions

- Arithmetic Decision Diagram (ADD)
  - Associate more values as terminal nodes
- Edge-Valued BDD (EVBDD)
  - Bring values from the terminal nodes to edges
- Binary Moment Diagram (BMD)
  - Decomposition : f = f(x=0) + x (δf/δx)

eg. $f=x_0+2x_1+4x_2$

Linearly Inductive BDD

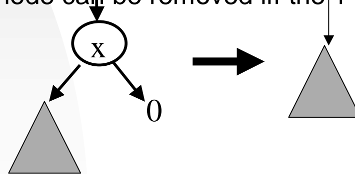
- Linearly Inductive Function
  - Recursively Defined
    - X^i: the inputs to the i-instance function
    - F^i : the i-instance function
    - B^i : an arbitrary boolean function
    - $F^1 = B^1(x^1)$ and $F^i = B^i (X^i, G^{i-1})$
    - $B^i = B^j$ for all i,j
- Example - Adder
  - $Sum^1 = a1 \oplus b1 \oplus c_in$
  - $Sum^i = ai \oplus bi \oplus Carry^{i-1}$
  - $Carry^1 = a1\ b1 + (a1+b1)\ c_in$
  - $Carry^i = ai\ bi + (ai+bi)\ Carry^{i-1}$

LIBDD

- Require different rules for *Apply* and *Reduce*
- Applicable to special case when good sequentiality can be captured
- Function definition has to be known in advance

Zero-Suppressed BDD

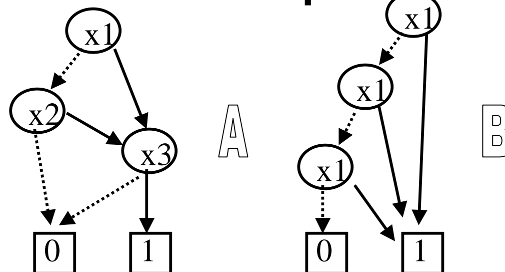
- Inspiration
  - To represent a collection of bit vectors
    - eg. (0001) (0100) (1010)
    - Such a representation is often encountered in some combinatorial problems
  - Usually, the 1's in the bit vectors is sparse
- ZBDD
  - Change the Reduction rule
    - A node can be removed iff the 1-edge points to 0



Li-C. Wang's Lecture

Jump to first page

ZBDD Interpretation



- In ZBDD sense
  - A = { 101, 011 } and B = { 100, 010, 001 }
- ZBDD is good for sets manipulation
- ZBDD can be used to represent a function as a collection of *cube* sets
  - x1,x2,x3={01-,-11,-00} => {011000,001010,000101}
  - To ensure sparse => 0: 01, 1:10, -: 00

Li-C. Wang's Lecture

Jump to first page