BDD – Based Upon Shannon Expansion

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

- \( f = x_1 \ x_2 + x_3 \ x_4 + x_1' \ x_2 \ x_5 \)
  - \( f(x_1=0) = x_3 \ x_4 + x_2 \ x_5 \)
  - \( f(x_1=1) = x_2 + x_3 \ x_4 \)
  - \( f = x_1' \ (x_3 \ x_4 + x_2 \ x_5) + x_1 \ (x_2 + x_3 \ x_4) \)

- Similarly
  - \( f = x_2' \ (x_3 \ x_4) + x_2 \ (x_1 + x_3 \ x_4 + x_1' \ x_5) \)

- To represent \( f \), we use a “divide-and-conquer” approach to represent \( f(x_i=0) \) and \( f(x_i=1) \)

Decision Tree

- \( f = x_1 \ x_2 + x_3 \)
- \( f = x_1 \ (x_2 + x_3) + x_1' \ (x_3) \)
- \( g_1 = x_2 \ (1) + x_2' \ (x_3) \)
- \( g_2 = x_3 \)

In \( f \), \( g_2 \) appears twice, 1 appears 3 times, and 0 appears twice
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
  - Also known as “read-once branch program” in comp. theory

```
Li-C. Wang’s Lecture
```

Classical OBDD Example

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

```
f = x1 ⊕ x2 ⊕ ... ⊕ xn
```

```
Li-C. Wang’s Lecture
```
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 $< O(n^2)$

```
4 2 3
no 1's 1 1's 2 1's 3 1's
```

Ordering Example

- $f = x_1 \cdot x_2 + x_3 \cdot x_4 + x_5 \cdot x_6$
  - ordering 1: $x_1 \cdot x_2 \cdot x_3 \cdot x_4 \cdot x_5 \cdot x_6$ (linear result)
    - If $(x_1 \cdot x_2) = 1$, $f = 1 \Rightarrow$ no need to read further!
    - Otherwise we can “forget about $x_1 \cdot x_2$”
    - The “memoryless effect” makes BDD small
  - ordering 2: $x_1 \cdot x_3 \cdot x_5 \cdot x_2 \cdot x_4 \cdot x_6$ (exponential result)
    - If $(x_1=x_3=x_5=1) \Rightarrow$ we need to remember them all!
    - Because nothing can be decided based upon the information we have
Ordering and OBDD Size

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

OBDD Size

- \( f = x_1 \cdot x_{n/2+1} + x_2 \cdot x_{n/2+2} + \ldots + x_{n/2} \cdot x_n \)
- proceed ordering as \( x_1 \cdot x_2 \cdot x_3 \cdot x_4 \ldots \cdot x_{(n-1)} \cdot x_n \)
- when reach \( x_{n/2} \)
  - all values in \( x_1 \cdot x_2 \cdot x_3 \ldots \cdot x_{(n-1)/2} \) 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
**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
Worst-Case Example

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

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 \(O(n/8)\)
More Terminology

- **Composition**
  - \( f(x_i=g)(x_1 \ldots x_n) = f(x_1, \ldots, g(x_1 \ldots x_n), \ldots, x_n) \)
  - ex. \( f=x_1x_2+x_3x_4 \) and \( g=x_2+x_3 \)
  - \( f(x_1=g) = x_2 + x_2x_3 + x_3x_4 = x_2 + x_3x_4 \)

- **Dependency Set**
  - \( I_i = \{ i \mid f(x_i=0) \neq f(x_i=1) \} \Rightarrow f \) depends on \( x_i \)
  - \( f = x_1x_2 x_3 + x_1' x_2 x_3 \Rightarrow f \) is independent of \( x_1 \)

- **Satisfying Set**
  - \( S_f = \{ (x_1 \ldots x_n) \mid f(x_1 \ldots x_n) = 1 \} \)
  - all assignments that made \( f = 1 \)
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 \Rightarrow 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

Apply and Reduce

- OBDD1 op OBDD2
- The basic algorithms to do so are
  - Apply(OBDD1, OBDD2) -> BDD3
  - Reduce(BDD3) -> OBDD3
Apply

- \( f_1 \, op \, f_2 = \)
  - \( x_i \, [ \, f_1(x_i=0) \, \, op \, \, f_2(x_i=0) \, ] \, + \, x_i \, [ \, f_1(x_i=1) \, \, op \, \, f_2(x_i=1) \, ] \)

Example

- \( F = X_1' + X_1 \, X_3' \)
Complexity of Apply

- $k \leq i \times j$
  - $i$ 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.
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

Labeling Scheme

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

iff \(a = c \&\& b = d\)
Redundant Nodes Removal

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

\[
\begin{align*}
\text{(1)} & \quad \text{(2)} & \quad \text{(1)} & \quad \text{(2)} \\
\text{(1,3)} & \quad \text{(3,3)} & \quad \text{(1,2)} & \quad \text{(1,2)} \\
0 & \quad 1 & \quad 0 & \quad 1 \\
1 & \quad 2 & \quad 2 & \quad 3 \\
1 & \quad 2 & \quad 3 & \quad 3 \\
\end{align*}
\]
Other BDDs

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**
  - \( f(x=0) \)
  - \( f(x=0) \oplus f(x=1) \)
OFDD

- \( F = (x_1 \oplus x_2) \cdot x_3 \)
  - \( F(x_3=0) = 0 \)
  - \( F(x_3=1) = x_1 \oplus x_2 \)
    - difference \( D_3 = x_1 \oplus x_2 \)
  - \( D_3(x_2=0) = x_1 \)
  - \( D_3(x_2=1) = x_1' \)
    - difference \( D_2 = 1 \)

- OFDD requires a different Reduction rule
- For some functions, OFDD can be exponentially smaller than OBDD and vice versa
- 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 cannot 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 data structures
Represent Numeric-Valued Functions

eg. \( f = x_0 + 2x_1 + 4x_2 \)

- 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: \( f = f(x=0) + x \left( \frac{\partial f}{\partial x} \right) \)

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
  - \( \text{Sum}^1 = a_1 \oplus b_1 \oplus c_{\text{in}} \)
  - \( \text{Sum}^i = a_i \oplus b_i \oplus \text{Carry}^{i-1} \)
  - \( \text{Carry}^1 = a_1 \oplus b_1 + (a_1+b_1) c_{\text{in}} \)
  - \( \text{Carry}^i = a_i b_i + (a_i+b_i) \text{Carry}^{i-1} \)
Li-C. Wang's Lecture

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
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
- \( x_1, x_2, x_3 = \{01-, -11, -00\} \Rightarrow \{011000, 001010, 000101\} \)
- To ensure sparse \( \Rightarrow 0: 01, 1: 10, -: 00 \)