Reduced Ordered Binary Decision Diagrams

- represents a logic function by a graph. *(many logic functions can be represented compactly - usually better than SOP’s)*

- **canonical** form (**important**) *(only canonical if an ordering of the variables is given)*

- Many logic operations can be preformed **efficiently** on BDD’s *(usually linear in size of result - tautology and complement are constant time)*

- **size** of BDD critically dependent on **variable ordering**.
ROBDD’s

- directed acyclic graph (DAG)
- one root node, two terminals 0,1
- each node, two children, and a variable
- Shannon co-factoring tree, except 

**Reduced:**

1. any node with identical children is removed
2. two nodes with isomorphic BDD’s are merged

**Ordered:** Co-factoring variables (splitting variables) always follow the same order

\[ x_{i_1} < x_{i_2} < x_{i_3} < \ldots < x_{i_n} \]
Two different orderings, same function.
Efficient Implementation of BDD’s

(Reference: Brace, Rudell, Bryant - DAC 1990)

Hash-Table: hash-fcn(key) = value

Strong canonical form: A ”unique-id” is associated (through a hash table) uniquely with each element in set.

With BDD’s the set is the set of all logic functions. A BDD node is a function. Thus each function has a unique-id in memory.

BDD is compressed Shannon co-factoring tree.
**ROBDD**

**Ordered BDD (OBDD)** Input variables are ordered - each path from root to sink visits nodes with labels (variables) in ascending order.

Reduced Ordered BDD - reduction rules:

1. if the two children of a node are the same, the node is eliminated - $f = cf + \overline{c}f$.

2. if two nodes have isomorphic graphs, they are replaced by one of them.

These two rules make it so that each node represents a distinct logic function.

**Theorem 1** (Bryant - 1986) ROBDD’s are canonical

Thus two functions are the same iff their ROBDD’s are equivalent graphs (isomorphic). Of course must use **same order** for variables.
Function is Given by Tracing All Paths to 1

\[ f = \overline{b} + \overline{ac} = \overline{ab} + \overline{ac}b + \overline{ac} \]  all paths to the 1 node

Notes:

- By tracing paths to the 1 node, we get a cover of pairwise disjoint cubes.
- The power of the BDD representation is that it does not explicitly enumerate all paths; rather it represents paths by a graph whose size is measured by its nodes and not paths.
- A DAG can represent an exponential number of paths with a linear number of nodes.
- Each node is given by its Shannon representation: \( f = af_a + \overline{a}f_{\overline{a}}. \)
Variables are totally ordered: If \( v < w \) then \( v \) occurs "higher" up in the ROBDD (call it BDD from now on).

**Definition 1** Top variable of a function \( f \) is a variable associated with its root node.

**Example:** \( f = ab + \overline{a}bc + \overline{a}b\overline{c} \). Order is \((a, b, c)\).

\[
\begin{align*}
&f_a = b \\
&f_{\overline{a}} = b \\
&\text{b is top variable of f}
\end{align*}
\]

\( f \) does not depend on \( a \), since \( f_a = f_{\overline{a}} \).

Each node is written as a triple: \( f = (v, g, h) \) where \( g = f_v \) and \( h = f_{\overline{v}} \). We read this triple as

\[
\begin{align*}
f &= \text{if} \ v \ \text{then} \ g \ \text{else} \ h \\
\text{v is top variable of f}
\end{align*}
\]

\[
\begin{align*}
&f = \text{ite}(v, g, h) = vg + \overline{v}h \\
&v \text{ is top variable of f}
\end{align*}
\]
ITE Operator

\[ \text{ite}(f, g, h) = fg + \overline{fh} \]

ite operator can implement any two variable logic function. There are 16 such functions corresponding to all subsets of vertices of \( B^2 \): \( \overline{fg}, \overline{f}g, f\overline{g}, fg \)

<table>
<thead>
<tr>
<th>Table</th>
<th>Subset</th>
<th>Expression</th>
<th>Equivalent Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>AND(f, g)</td>
<td>fg</td>
<td>ite(f, g, 0)</td>
</tr>
<tr>
<td>0010</td>
<td>f &gt; g</td>
<td>fg</td>
<td>ite(f, 1, g)</td>
</tr>
<tr>
<td>0011</td>
<td>f</td>
<td>f</td>
<td>f</td>
</tr>
<tr>
<td>0100</td>
<td>f &lt; g</td>
<td>\overline{fg}</td>
<td>ite(f, 0, g)</td>
</tr>
<tr>
<td>0101</td>
<td>g</td>
<td>g</td>
<td>g</td>
</tr>
<tr>
<td>0110</td>
<td>XOR(f, g)</td>
<td>f \oplus g</td>
<td>ite(f, \overline{g}, g)</td>
</tr>
<tr>
<td>0111</td>
<td>OR(f, g)</td>
<td>f + g</td>
<td>ite(f, 0, \overline{g})</td>
</tr>
<tr>
<td>1000</td>
<td>NOR(f, g)</td>
<td>f + \overline{g}</td>
<td>ite(f, g, \overline{g})</td>
</tr>
<tr>
<td>1001</td>
<td>XNOR(f, g)</td>
<td>f \oplus \overline{g}</td>
<td>ite(f, g, \overline{g})</td>
</tr>
<tr>
<td>1010</td>
<td>NOT(g)</td>
<td>\overline{g}</td>
<td>ite(f, g, 0, 1)</td>
</tr>
<tr>
<td>1011</td>
<td>f \geq g</td>
<td>f + \overline{g}</td>
<td>ite(f, 1, \overline{g})</td>
</tr>
<tr>
<td>1100</td>
<td>NOT(f)</td>
<td>\overline{f}</td>
<td>ite(f, 0, 1)</td>
</tr>
<tr>
<td>1101</td>
<td>f \leq g</td>
<td>\overline{f} + g</td>
<td>ite(f, g, 1)</td>
</tr>
<tr>
<td>1110</td>
<td>NAND(f, g)</td>
<td>\overline{fg}</td>
<td>ite(f, \overline{g}, 1)</td>
</tr>
<tr>
<td>1111</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Before a node \((v, g, h)\) is added to BDD data base, it is looked up in the "unique-table". If it is there, then existing pointer to node is used to represent the logic function. Otherwise, a new node is added to the unique-table and the new pointer returned.

Thus a strong canonical form is maintained. The node for \(f = (v, g, h)\) exists \textit{iff} \((v, g, h)\) is in the unique-table. There is only one pointer for \((v, g, h)\) and that is the address to the unique-table entry.

Unique-table allows single multi-rooted DAG to represent all users’ functions:
Recursive Formulation of ITE

\[ v = \text{top-most variable among the three BDD's } f, g, h \]

\[
\begin{align*}
\text{ite}(f, g, h) & = fg + \overline{f}h \\
& = \overline{v}(fg + \overline{f}h) + \overline{v}(fg + \overline{f}h) \\
& = \overline{v}(fv + \overline{f}hv) + \overline{v}(fv + \overline{f}hv) \\
& = \text{ite}(v, \text{ite}(fv, g, hv), \text{ite}(fv, g, hv)) \\
& = (v, \text{ite}(fv, g, hv), \text{ite}(fv, g, hv)) \\
& = (v, \tilde{f}, \tilde{g}) = R
\end{align*}
\]

Terminal cases:
\[ (0, g, f) = (1, f, g) = f \]
\[ \text{ite}(f, g, g) = g \]

\[
\begin{align*}
\text{ite}(f, g, h) & \text{ if(terminal case)} \{ \\
& \text{return result;} \\
\} \text{ else if (computed-table has entry } (f, g, h) \{ \\
& \text{return result;} \\
\} \text{ else} \{ \\
& \text{let } v \text{ be the top variable of } (f, g, h); \\
& \tilde{f} \leftarrow \text{ite}(fv, g, hv); \\
& \tilde{g} \leftarrow \text{ite}(fv, g, hv); \\
& \text{if } (\tilde{f} \text{ equals } \tilde{g}) \text{ return } \tilde{g}; \\
& R \leftarrow \text{find_or_add_unique_table}(v, \tilde{f}, \tilde{g}); \\
& \text{insert_computed_table} \{ \{ f, g, h \}, R \}; \\
& \text{return } R; \} \}
\end{align*}
\]

The "insert_computed_table" is a cache table where \text{ite} results are cached.
Example

\[ \begin{align*}
I & = \text{ite}(F, G, H) \\
& = (a, \text{ite}(F_a, G_a, H_a), \text{ite}(F_{\overline{a}}, G_{\overline{a}}, H_{\overline{a}})) \\
& = (a, \text{ite}(1, C, H), \text{ite}(B, 0, H)) \\
& = (a, C, (b, \text{ite}(B_b, 0_b, H_b), \text{ite}(B_{\overline{b}}, 0_{\overline{b}}, H_{\overline{b}}))) \\
& = (a, C, (b, \text{ite}(1, 0, 1), \text{ite}(0, 0, D))) \\
& = (a, C, (b, 0, D)) \\
& = (a, C, J)
\end{align*} \]

Check:

\[ \begin{align*}
F & = a + b \\
G & = ac \\
H & = b + d
\end{align*} \]

\[ \begin{align*}
\text{ite}(F, G, H) & = (a + b)(ac) + \overline{a}b(b + d) \\
& = ac + \overline{a}bd
\end{align*} \]
Computed Table

Keep a record of \((F, G, H)\) triplets already computed by the \(\text{ite}\) operator in a hash-based cache ("cache" table). This means that the collision chain is not used (if collision, old entry thrown away).

However, this is wasteful since the BDD nodes and collision chain can be merged.
Here the BDD nodes and the collision chain are merged. On average, only 4 pointers per BDD node.
Extension - Complement Edges

Can combine by making complement edges:

only one dag using complement pointer
To maintain strong canonical form, need to resolve 4 equivalences:

Solution: Always choose one on left, i.e. the ”then” leg must have no complement edge.
Ambiguities in Cache Table

Standard Triples:

\[
\begin{align*}
\text{ite}(F,F,G) & \implies \text{ite}(F,1,G) \\
\text{ite}(F,G,F) & \implies \text{ite}(F,G,0) \\
\text{ite}(F,G,F') & \implies \text{ite}(F,G,1) \\
\text{ite}(F,F',G) & \implies \text{ite}(F,0,G) \\
\end{align*}
\]

To resolve equivalences:

\[
\begin{align*}
\text{ite}(F,1,G') & \equiv \text{ite}(G,1,F) \\
\text{ite}(F,0,G) & \equiv \overline{\text{ite}(G,0,F')} \\
\text{ite}(F,G,0) & \equiv \text{ite}(G,F,0) \\
\text{ite}(F,G,1) & \equiv \overline{\text{ite}(G,F,1')} \\
\text{ite}(F,G,G') & \equiv \overline{\text{ite}(G,F,F')} \\
\end{align*}
\]

1. first argument is chosen with smallest top variable.

2. break ties with smallest address pointer.

Triples:

\[
\text{ite}(F,G,H) \equiv \overline{\text{ite}(F,H,G)} \equiv \overline{\text{ite}(F,G,H')} \equiv \overline{\text{ite}(F,H,G')}
\]

Choose the one such that the first and second argument of ite should not be complement edges (i.e. the first one above).
Tautology Checking

Tautology returns 0, 1, or NC (not constant).

\[
\text{ITE\_constant}(F, G, H)\{
\begin{array}{l}
\text{if (trivial case) } \{
\text{return result (0, 1, or NC);}
\}\text{ else if (cache table has entry for } (F, G, H)) \{
\text{return result;}
\}\text{ else } \{
\text{let } v \text{ be the top variable of } F, G, H;
\text{ } R \leftarrow \text{ITE\_constant}(F_v, G_v, H_v);
\text{if } (R = \text{NC}) \{
\text{insert\_cache\_table} \{F, G, H\}, \text{NC};
\text{return NC;}
\}\}
\text{ } E \leftarrow \text{ITE\_constant}(F_\overline{v}, G_\overline{v}, H_\overline{v});
\text{if } (E = \text{NC} \text{ or } R \neq E)\{
\text{insert\_cache\_table} \{F, G, H\}, \text{NC};
\text{return NC;}
\text{insert\_cache\_table} \{F, G, H\}, E);
\text{return } E;
\end{array}
\}
\]

Note, that in computing \text{ITE\_constant}, we set up a temporary cache-table for storing results of the \text{ITE\_constant} operator. When done, we can throw away this table if we like.
Compose

Compose is an important operation for building the BDD of a circuit.

\[
\text{compose}(F, v, G) : \ F(v, x) \to F(G(x), x)
\]

Means substitute \(v = G(x)\).

```plaintext
compose (F, v, G) {  
  (in F replace v with G)
  if (top_var(F) > v) return F;
  (because F does not depend on v)
  if (top_var(F) == v) return ITE(G, F1, F0);
  R ← compose(F1, v, G);
  E ← compose(F0, v, G);
  return ITE(top_var(F), R, E);
  (note that we call ITE on this rather than )
  (returning (top_var(F'), R, E) )   }
```

Notes:

1. \(F_1\) is the 1-child of \(F\), \(F_0\) the 0-child

2. \(G, R, E\) are not functions of \(v\)

3. if \(\text{top}_\text{var}\) of \(F\) is \(v\), then \(\text{ite}(G, R, E)\) does the replacement of \(v\) by \(G\).
Multivalued Decision Diagrams (MDD’s) - ”BDD’s” for MV-functions

There is an equivalent theory (canonical etc.) for MDD’s:

Typically, we encode the multi-valued variable with $\log_2(|P_v|)$ binary variables and use unused codes as ”don’t cares” in a particular way

Sets and Graphs:
Thus we can represent and manipulate general sets and graphs.

Set $\iff$ characteristic function of set

\[ (f(v) = 1) \iff (v \in S \subseteq P_v) \]

Graph: $((f(x, y) = 1) \iff (x, y)$ is an edge in graph
where $x$ and $y$ are multi-valued variables representing nodes in the graph.

ZBDD’s were invented by Minato to efficiently represent sparse sets. they have turned out to be extremely useful in implicit methods for representing primes (which usually are a sparse subset of all cubes).
Zero Suppressed BDD’s - ZBDD’s

Different reduction rules:

- **BDD**: eliminate all nodes where *then* edge and the *else* edge point to the same node.

- **ZBDD**: eliminate all nodes where the *then* node points to 0. Connect incoming edges to *else* node.

- For Both: share equivalent nodes.
Canonicity

**Theorem 2 (Minato)** ZBDD’s are canonical given a variable ordering and the support set.

**Example:**

![Diagram showing BDD and ZBDD examples]

- BDD
  - Variable order: $x_1, x_2, x_3$
  - Support: $x_1, x_2, x_3$

- ZBDD if support is $x_1, x_2$
  - Variable order: $x_1, x_3$

- ZBDD if support is $x_1, x_2, x_3$
  - Variable order: $x_3, x_2, x_1$