Binary Decision Diagrams

May 3, 2004
Overview

• Boolean functions

• Possible representations
  – Binary decision trees
  – Binary decision diagrams
  – Ordered binary decision diagrams
  – Reduced ordered binary decision diagrams

• Operations on ordered binary decision diagrams
  – Validity, satisfiability, function equivalence
  – Function composition operations
Boolean functions

A boolean function maps boolean inputs (0s and 1s) to a boolean result (0 or 1).

Basic functions:

- \( \overline{0} = 1, \overline{1} = 0 \)
  (complementation).
- \( x + y = 0 \Leftrightarrow x = y = 0 \)
  (disjunction, like addition).
- \( x \cdot y = 1 \Leftrightarrow x = y = 1 \)
  (conjunction, like multiplication).
- \( x \oplus y = 1 \Leftrightarrow x \neq y \)
  (exclusive or).

Complex functions: \( h(x, y, z) = z + y \cdot (x \oplus y) \)

Function composition: \( h = f + \overline{g} \)

Goal: represent models and CTL formulas.
Representing boolean functions

Goal: efficiently implement useful operations:

• Validity
• Satisfiability
• Equivalence of functions

Approaches:

• Truth tables.
• Propositional formulas.
• Propositional formulas in a canonical form. (e.g. CNF, DNF)
Binary decision trees

Truth-table representation:

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>z</th>
<th>((x + y) \cdot z)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Tree representation:

Evaluation:

- Use the dotted edge if the variable is 0.
- Use the solid edge if the variable is 1.

Structure depends on the input/output relation, not on the formula definition.
Problems

Trees no more efficient than truth tables.

Tree structure depends on the variable ordering:

Repeated variables can create unreachable edges.
Binary decision diagrams (BDDs)

BDDs: Trees with sharing (DAGs).

Reduction:

- Eliminate redundant information from a BDD.
- Share redundant subgraphs.
Reduction rules

Multiple 0’s and 1’s clearly redundant.

Example:

\[
\begin{array}{ccccccc}
0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\
\end{array}
\]

becomes:

\[
\begin{array}{ccccccc}
0 & 1 \\
\end{array}
\]
Reduction rules, contd.

If a node has the same child at both outgoing edges, the node is redundant.

\[
\begin{array}{c}
\begin{array}{c}
\text{0} \quad \text{1} \\
\vdots \quad \vdots \\
\end{array}
\end{array}
\Rightarrow
\begin{array}{c}
\begin{array}{c}
\text{0} \quad \text{1} \\
\vdots \quad \vdots \\
\end{array}
\end{array}
\]

If two nodes represent the same variable and have the same descendants, one is redundant.

\[
\begin{array}{c}
\begin{array}{c}
\text{0} \quad \text{1} \\
\vdots \quad \vdots \\
\end{array}
\end{array}
\Rightarrow
\begin{array}{c}
\begin{array}{c}
\text{0} \quad \text{1} \\
\vdots \quad \vdots \\
\end{array}
\end{array}
\Rightarrow
\begin{array}{c}
\begin{array}{c}
\text{0} \quad \text{1} \\
\vdots \quad \vdots \\
\end{array}
\end{array}
\]

9
Variable ordering

Reduction opportunities depend on variable ordering.

Examples:

```
0 0 0 1 0 1 0 1
```

```
0 0 0 1 0 0 1 1
```

```
0 0 0 1 0 1 0 1
```

```
0 0 0 1 0 0 1 1
```
Summary

Reduction rules:

- Coalescing duplicate terminal nodes.
- Removal of redundant tests.
- Coalescing duplicate non-terminals.

No rules introduce terminals, so the first rule can be applied all at once.

Otherwise, one application can make opportunities for others:

The rules are confluent.

11
Operations on BDDs

Desirable predicates:

- Validity
- Satisfiability
- Equivalence of functions

Desirable operations:

- $\overline{f}$
- $f + g$
- $f \cdot g$
- $f \oplus g$
Predicates

- Validity: Is the result always 1?
- Satisfiability: Is the result ever 1?

Ideally:

- Validity: check for absence of 0.
- Satisfiability: check for presence of 1.

Problem: Repeated variables imply that some edges might be unreachable.

Example: BDD for \( x + \overline{x} \):

```
      x
     / \x
    0   1
```

Solution: Evaluate on all inputs (truth-table approach).
Equivalence of functions

Ideally: Check whether BDDs have the same structure.

Problem: structure sensitive to:

- Variable ordering
- Repeated variables
- Redundant nodes

Solution: Evaluate both functions on all inputs (again, the truth-table solution).
Operations

• \( \overline{f} \): interchange 0’s and 1’s.

• \( f + g \):
  
  – If 1 reached in \( f \), return 1.
    Otherwise, check \( g \).

  – Replace the 0 nodes of \( B_f \) by \( B_g \).

\[ \begin{align*}
  x : & \quad x \\
  \quad & \quad 0 \quad 1 \\
\end{align*} \]
\[ \begin{align*}
  \overline{x} : & \quad x \\
  \quad & \quad 1 \quad 0 \\
\end{align*} \]
\[ \begin{align*}
  x + \overline{x} : & \quad x \\
  \quad 1 & \quad 1 \quad 0 \\
\end{align*} \]

• \( f \cdot g \):

  – If 0 reached in \( f \), return 0.
    Otherwise, check \( g \).

  – Replace the 1 nodes of \( B_f \) by \( B_g \).

Calculations efficient, but variables occur multiply in resulting paths.
Ordered BDDs (OBDDs)

Problems with BDDs:

- Repeated variables cause inaccessible edges.
- Structure depends on variable ordering.

Solution: Require a fixed order for all paths.

Ordered: \([y, z, x]:\)

Not ordered:

Any BDD can be converted to any order.
Properties of OBDDs

No duplication implies no inaccessible edges.

- **Validity**: absence of 0.

- **Satisfiability**: presence of 1.

- **Implication**:
  
  - $f \rightarrow g \equiv \neg f \lor g \equiv \neg (f \land \neg g)$.
  
  - Check $\overline{f \cdot \overline{g}}$ for validity.

  - Equivalently, check that $f \cdot \overline{g}$ has no 1 (always false).
Properties of reduced OBDDs

• Absence of redundant variables: A reduced OBDD contains no redundant variables.

Example:

– OBDD:

\[
\begin{array}{c}
\text{x} \\
\text{y} \\
0 \\
1
\end{array} \rightarrow
\begin{array}{c}
\text{x} \\
\text{y} \\
0 \\
1
\end{array} \rightarrow
\begin{array}{c}
0 \\
1
\end{array}
\]

– non-OBDD:

\[
\begin{array}{c}
\text{x} \\
\text{y} \\
\text{y} \\
0 \\
1
\end{array}
\]

• Equivalence of functions:
  Reduced OBDDs must be identical.
  Complexity linear in OBDD size.
Properties of reduced OBDDs, contd.

- Validity: Reduced OBDD equivalent to 1.

- Satisfiability: Reduced OBDD not equivalent to 0.

- Implication: $f \rightarrow g$ iff reduced OBDD of $f \cdot \overline{g}$ is equivalent to 0.

Constant time operations.
OBDDs and variable ordering

Variable ordering affects reduced OBDD size, which affects manipulation costs.

\[(x_1 + x_2) \cdot (x_3 + x_4) \cdot (x_5 + x_6)\]

\[[x_1, x_2, x_3, x_4, x_5, x_6] : \quad [x_1, x_3, x_5, x_2, x_4, x_6] :\]

For \((x_1 + x_2) \cdot (x_3 + x_4) \cdot \ldots \cdot (x_{2n-1} + x_{2n})\):

- \([x_1, x_2, \ldots, x_{2n}] \Rightarrow 2n + 2 \text{ nodes.}\)
- \([x_1, x_3, \ldots, x_{2n-1}, x_2, x_4, \ldots x_{2n}] \Rightarrow 2^{n+1} \text{ nodes.}\)

Problem: the same ordering must be used for all functions.
Reduction for OBDDs

Efficient bottom-up algorithm.

Idea: label nodes, giving redundant nodes the same label.

Labeling allows efficient comparison of subgraphs.

Terminology:

- \( l(n) \): the child at the dotted edge of \( n \).
- \( h(n) \): the child at the solid edge of \( n \).
Labeling algorithm

1. Label all 0’s with 1, all 1’s with 1.

2. If \( l(n) = h(n) \), label \( n \) with \( l(n) \).

3. If there is a node \( m \) with the same variable such that \( l(n) = l(m) \) and \( h(n) = h(m) \), label \( n \) with \( m \)’s label.

4. Otherwise, pick a fresh label for \( n \).

Apply reduction rules, bottom-up. If two children have the same label, they are the same.
Complexity

- Visit each node: $O(n)$

- Check whether children identical: $O(1)$

- Search for a predecessor with the same variable and the same children. By first sorting: $O(\log n)$
Boolean functions

Goal: combine OBDDs so that the result is an OBDD.

Operations: \( f + g, f \cdot g, f \oplus g \).

Observation: \( f \circ g \) trivial if \( f \) and \( g \) are constant functions.

Example:

\[
f = 0, g = 1 \Rightarrow f + \overline{g} = 0 + \overline{1} = 0
\]

Inductive strategy:

- Compute \( f \circ g \) on \( n \) variables in terms of \( f' \circ g' \) on \( n - 1 \) variables.
- Eventually reach a constant function.
Shannon expansion

Law of the excluded middle: \( x = 0 \) or \( x = 1 \).

\( f[0/x], f[1/x] \) have fewer variables than \( f \).

Idea: redefine \( f \) in terms of \( f[0/x] \) and \( f[1/x] \):

- + combines, but \( f[0/x] + f[1/x] \) is wrong.
- Drop the component that does not use \( x \)'s value.

\[- \ x \cdot x \cdot \alpha: \ x = 0 \Rightarrow \text{drop } \alpha. \]
\[\text{ } \ x = 1 \Rightarrow \text{keep } \alpha. \]

\[- \ x \cdot \alpha: \ x = 0 \Rightarrow \text{keep } \alpha. \]
\[\text{ } \ x = 1 \Rightarrow \text{drop } \alpha. \]

\[f = \ x \cdot f[0/x] + x \cdot f[1/x] \]

\[f \ op g = \ x \cdot (f \ op g)[0/x] + x \cdot (f \ op g)[1/x] = \ x \cdot (f[0/x] \ op g[0/x]) + \ x \cdot (f[1/x] \ op g[1/x]) \]
Implementing $\text{apply}(\text{op}, B_f, B_g)$

$$f \text{ op } g = \overline{x} \cdot (f[0/x] \text{ op } g[0/x]) + x \cdot (f[1/x] \text{ op } g[1/x])$$

Construct OBDDs for $f[0/x]$, $f[1/x]$ from an OBDD for $f$.

If $x$ is at the root:

- $f[0/x]$: the target of the dotted edge.
- $f[1/x]$: the target of the solid edge.

Inductive algorithm eliminates all variables, so choose the root each time.

OBDDs, so $f$ and $g$ variables in the same order.
apply algorithm

Cases:

- Both roots are terminals:
  Apply $op$ directly.

- Both roots associated with $x$:

$$x \quad \text{app}(op, l(r_f), l(r_g)) \quad \text{app}(op, h(r_f), h(r_g))$$

- Incompatible roots: deferred.

These rules are sufficient if all paths contain all variables.

Result: a full binary tree. Reduction required.
Connection to Shannon expansion

\[ f \mathop{\otimes} g = \overline{x} \cdot f[0/x] \mathop{\otimes} g[0/x] + x \cdot f[1/x] \mathop{\otimes} g[1/x] \]

BDDs:

\[
\begin{align*}
\overline{x} : & \quad x \\
& \quad 1 \quad 0 \\
\end{align*}
\]

\[
\begin{align*}
x : & \quad x \\
& \quad 0 \quad 1 \\
\end{align*}
\]

\[
\begin{align*}
\overline{x} \cdot f[0/x] \mathop{\otimes} g[0/x] : & \quad x \\
& \quad 0 \quad \text{app}(\mathop{\otimes}, l(r_f), l(r_g)) \\
\end{align*}
\]

\[
\begin{align*}
x \cdot f[1/x] \mathop{\otimes} g[1/x] : & \quad x \\
& \quad 0 \quad \text{app}(\mathop{\otimes}, h(r_f), h(r_g)) \\
\end{align*}
\]

Final sum:

\[
\begin{align*}
\text{app}(\mathop{\otimes}, l(r_f), l(r_g)) \\
& \quad x \\
& \quad 0 \quad \text{app}(\mathop{\otimes}, h(r_f), h(r_g)) \\
\end{align*}
\]
Example

Compute product:

\[ x \oplus y : \quad x^{1_a} \leftrightarrow y^{1_b} \]

\[ y^{2_a} \leftrightarrow y^{3_a} \leftrightarrow 0^{4_a} \rightarrow 1^{5_a} \]

\[ y^{2_b} \leftrightarrow y^{3_b} \leftrightarrow 0^{4_b} \rightarrow 1^{5_b} \]
Completing the apply algorithm

Remaining cases:
- One root a variable, the other a constant.
- The two roots represent different variables.

OBDD property: If the variable earlier in the ordering is not the root of one tree, it is not in that tree.

Idea:

\[
\begin{array}{cccccc}
  x & \overset{\text{op}}{\rightarrow} & y & \equiv & x & \overset{\text{op}}{\rightarrow} & x \\
  0 & 1 & 0 & 1 & 0 & 1 & y \\
  & & & & & 0 & 1
\end{array}
\]

Rules:

\( r_f < r_g \):

\[
\text{app}(\text{op}, l(r_f), r_g) \quad \text{app}(\text{op}, h(r_f), r_g)
\]

\( r_g < r_f \):

\[
\text{app}(\text{op}, r_f, l(r_g)) \quad \text{app}(\text{op}, r_f, h(r_g))
\]
Example

Compute product:

\[ \begin{array}{c}
  x^{1_a} \\
  \downarrow \\
  z^{3_a} \\
  0^{4_a} 1^{5_a}
\end{array} \quad \begin{array}{c}
  x^{1_b} \\
  \downarrow \\
  y^{2_b} \\
  \downarrow \\
  z^{3_b} \\
  0^{4_b} 1^{5_b}
\end{array} \]
Complexity

Observation: apply algorithm often creates a full binary tree.

- $2^n$ nodes, for $n$ variables.

Problems:

- Repeated computations.
- Distinct computations give the same result.

Memoisation addresses the first problem.

- Only compute each app($op, n, m$) once.
- Complexity $O(|B_f| \cdot |B_g|)$.
Quantifiers

• $$(\exists x.f)(x_1, \ldots, x_n) = 1$$ iff:
  – $f$ returns 1 for these values when $x = 0$, or
  – $f$ returns 1 for these values when $x = 1$.
  – Encoding: $\exists x.f = f[0/x] + f[1/x]$

• $$(\forall x.f)(x_1, \ldots, x_n) = 1$$ iff:
  – $f$ returns 1 for these values when $x = 0$, and
  – $f$ returns 1 for these values when $x = 1$.
  – Encoding: $\forall x.f = f[0/x] \cdot f[1/x]$
Computing \( \text{restrict}(c, x, B) \)

For apply, \( x \) is always the root variable.

- \( f[0/x] \) is the left child of the root.
- \( f[1/x] \) is the right child of the root.

For restrict, \( x \) can be any variable.

- For each \( x \), connect all incoming edges to "c" child.

Example:

\[
\neg y \wedge (x \vee z) : \\
\quad \begin{array}{c}
0 \\
\end{array}
\quad \begin{array}{c}
1 \\
\end{array}
\end{array}
\quad \begin{array}{c}
\quad x \\
0 \\
\end{array}
\quad \begin{array}{c}
1 \\
\end{array}
\]

\[
\quad [0/y] \\
\quad \begin{array}{c}
\quad y \\
0 \\
\end{array}
\quad \begin{array}{c}
1 \\
\end{array}
\]
Implementing quantification

∃x.f = \text{app}(+, \text{restrict}(0, x, B_f), \text{restrict}(1, x, B_f))

∀x.f = \text{app}(·, \text{restrict}(0, x, B_f), \text{restrict}(1, x, B_f))

Example, ∃d.f:

Restrictions:

Sum:

\[
\begin{align*}
0 + 0 &= 0 \\
1 + 1 &= 1 \\
0 + e_r &= e
\end{align*}
\]
Improving efficiency

Most of the tree is not affected by restrictions.

Thus most of the tree is duplicated in the result (e.g. $b_l + b_r$).

More efficient approach: replace quantified node by the sum of the children.

Example:

\[
\begin{array}{c}
\begin{array}{c}
 a \\
 b \\
 c \\
 d \\
 e
\end{array} \\
0 \\
1
\end{array} \quad \rightarrow \quad
\begin{array}{c}
\begin{array}{c}
 a \\
 b \\
 c \\
 e
\end{array} \\
0 \\
1
\end{array}
\]

Nested quantification NP-complete.
## Complexity summary

<table>
<thead>
<tr>
<th>Operation</th>
<th>Input</th>
<th>Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>reduce</td>
<td>$B$</td>
<td>$O(</td>
</tr>
<tr>
<td>apply</td>
<td>$B_r, B_g$</td>
<td>$O(</td>
</tr>
<tr>
<td>restrict</td>
<td>$B_r$</td>
<td>$O(</td>
</tr>
<tr>
<td>exists</td>
<td>$B_r$</td>
<td>NP-complete</td>
</tr>
</tbody>
</table>

$B_r$: a reduced OBDD.
## Comparison with other approaches

<table>
<thead>
<tr>
<th></th>
<th>Compact</th>
<th>Sat.</th>
<th>Val.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prop. formulas</td>
<td>+</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>DNF formulas</td>
<td>0</td>
<td>+</td>
<td>–</td>
</tr>
<tr>
<td>CNF formulas</td>
<td>0</td>
<td>–</td>
<td>+</td>
</tr>
<tr>
<td>Truth tables</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Reduced OBDDs</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>+</th>
<th>.</th>
<th>–</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prop. formulas</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>DNF formulas</td>
<td>–</td>
<td>+</td>
<td>–</td>
</tr>
<tr>
<td>CNF formulas</td>
<td>+</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Truth tables</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Reduced OBDDs</td>
<td>0</td>
<td>0</td>
<td>+</td>
</tr>
</tbody>
</table>