CS357 Lecture: BDD basics

David Dill
BDDs (Boolean/binary decision diagrams)

BDDs are a very successful representation for Boolean functions.

A BDD represents a Boolean function on variables $x_1, x_2, \ldots, x_n$.

BDDs have many applications in CAD for digital systems, and formal verification (and current context-sensitive analysis, according to Alex).

Boolean functions are fundamental to computation, so there are many other applications as well.

Don Knuth: "one of the only really fundamental data structures that came out in the last twenty-five years"
**Ordered decision tree**

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_3$</th>
<th>$f$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</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>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</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>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Crucial constraint:**
*There is a global total order on the variables.*
Variables appear in this order along all paths in the BDD.
Space optimization 1: share identical subtrees

Nodes can be merged bottom-up
Space optimization 2: delete nodes with identical children

![Diagram showing space optimization](image-url)
Multiplexer tree interpretation

BDDs can be regarded as digital circuits
Canonicity of BDDs

A BDD is a canonical representation of a Boolean function.

- There is only one BDD for a Boolean function
- Boolean expressions are *not* canonical:
  \[(a \land b) \lor (a \land c)\] is same as \[(a \land (b \lor c))\]
- Easy to check for
  - Boolean equivalence (same BDD)
  - Tautology (= 1)
  - Unsatisfiability (= 0)
Implementation

BDD data structure is either "0", "1" or pointer to a record with \texttt{var}, \texttt{thenbdd}, \texttt{elsebdd}.

BDDs are constructed bottom-up.

For subtree sharing, maintain a hash table that maps \texttt{<var, thenptr, elseptr>} to \texttt{bddptr}.
Implementation

Assumes var < any vars appearing in thenbdd, elsebdd

make_bdd(var, thenbdd, elsebdd):
    if thenbdd = elsebdd return thenbdd;
    else if (lookup(var, thenbdd, elsebdd))
        return old bdd from hash table;
    else {
        b = new_node(var, thenbdd, elsebdd);
        insert_hash(<var, thenbdd, elsebdd>, b);
        return b;
    }
Cofactors and Shannon decomposition

Cofactoring a Boolean function restricts the function to a particular value of a variable.

\[ f \downarrow x_i \] is the cofactor of \( f \) when \( x_i = 1 \).

This is the function you get if you replace \( x_i \) by 1 everywhere in an expression for \( f \).

\[ f \downarrow \neg x_i \] is the cofactor of \( f \) when \( x_i = 0 \).

\( x_i \) does not appear in \( f \downarrow x_i \) or in \( f \downarrow \neg x_i \).
Shannon decomposition

Allows recursive decomposition of functions

\[ f = \text{ite}(x_i, (f \downarrow x_i), (f \downarrow \neg x_i)) \]

This identity holds regardless of where \( x_i \) appears in the variable order,

but Shannon decomposition is a lot faster when applied to the top-most variable.
Logical operations on BDDs.

All propositional connectives are implemented in the following way (many other operations are, too).

1. Use Shannon decomposition to define a recursive function (on trees).
2. Save and reuse intermediate results (“memo-ize”).
Logical operations on BDDs.

\[ \text{AND}(b_1, b_2): \]
\[
\begin{align*}
\text{if } b_1 &= 0 \text{ or } b_2 = 0 \text{ return } 0; \\
\text{else if } b_1 &= 1 \text{ return } b_2; \\
\text{else if } b_2 &= 1 \text{ return } b_1; \\
\text{else if } b_1.\text{var} &= b_2.\text{var} \\
&\quad \text{return } \text{make_bdd}(b_1.\text{var}, \text{AND}(b_1.\text{t}, b_2.\text{t}), \text{AND}(b_1.\text{e}, b_2.\text{e})); \\
\text{else if } b_1.\text{var} &< b_2.\text{var} \\
&\quad \text{return } \text{make_bdd}(b_1.\text{var}, \text{AND}(b_1.\text{t}, b_2), \text{AND}(b_1.\text{e}, b_2)); \\
\text{else if } b_2.\text{var} &< b_1.\text{var} \\
&\quad \text{return } \text{make_bdd}(b_2.\text{var}, \text{AND}(b_1, b_2.\text{t}), \text{AND}(b_1, b_2.\text{e}));
\end{align*}
\]
Logical operations on BDDs.

However, time to do this is proportional to tree size, which might be exponentially more than DAG size. So, use dynamic programming ("memo-ization").

\[
\text{AND}(b_1, b_2):
\]

\[
\ldots
\]

\[
\text{if lookup_in_AND_table}(b_1, b_2)
\]

\[
\text{return old value;}
\]

\[
\text{else}
\]

\[
\text{build new BDD } "b"
\]

\[
\text{insert_in_AND_table}(<b_1, b_2>, b);
\]
Logical operations on BDDs.

After this optimization, cost is proportional to product of BDD sizes.

Same approach can be used for OR, NAND, XOR, etc.

Also, for $\text{ite}(b1, b2, b3)$ - "if then else".
Logical operations on BDDs.

Translation from a logical expression to a BDD:
   Bottom-up evaluation of expression, using BDD operations.

BDD_trans(f):
   if f is a variable,
       build and return the BDD for it.
   else if f = g & h,
       return
       AND(BDD_trans(g), BDD_trans(h));
   etc.
BDD size

• Conversion from expression to BDD can cause exponential blowup
  – Each logical operation can result in a BDD that is product of input BDDs (details depend on how you measure size).
  – \( n \) operations on BDD’s of size 2: \( O(2^k) \)
  – Any smaller result would have exciting implications for NP completeness.
  – Sometimes, BDD’s are small
    Usually, you have to work at it!
BDD size

BDD size is strongly influenced by variable order
- BDD’s are almost never small unless variable order is good.

Some functions always have small BDDs
- Symmetric functions (order of inputs doesn’t matter).
- Addition
- Bitwise equivalence

“Most” functions have large BDDs
- Multiplication (any output bit).
- FFT, division, . . .
Variable order

• A good variable order is usually essential for compact BDDs
• Static variable ordering: Use heuristics to find a good order, i.e.
  – Interleave bits of operands of operators
  – Operand-specific orders (e.g., addition low-order to high-order or vice versa)
  – Put “control variables” at the top of the order (e.g. variables in conditionals that totally change functional behavior).
Dynamic Variable Ordering

• Variable ordering can also be done dynamically as BDD operations proceed.
• Optimal variable order problem is NP-complete.
• Many heuristics proposed. Rudell’s “sifting” is widely used.
  – Try moving a variable to all other positions, leaving the others fixed. Then place variable in the position that minimizes BDD size.
  – Do this for all variables.
• Sifting is enabled by fast “swapping” of two adjacent variables in a BDD.
Dynamic Variable Ordering

- Enabled a leap in BDD effectiveness.
- Can be SLOW. Feels like garbage collection (BDD operations stop while it reorders), but slower.
- Sometimes, for a particular problem, you can save the order found by sifting and reuse it effectively.
BDDs for symmetric functions

• symmetric: function is the same even if variables change.
• Examples:
  – $\text{AND}(x_1, x_2, \ldots, x_n)$ [OR, NAND, NOR]
  – $\text{PARITY}(x_1, x_2, \ldots, x_n)$ [n-ary XOR]
  – Threshold functions (at least $k$ out of $n$ bits are 1)
• Size is same regardless of variable ordering (why?)
Symmetric function example

Majority function: output is majority value of three inputs.

Parity: 1 iff odd number of true variables.

Intuition: BDD nodes only need to “remember” number of variables that are 1.
Boolean quantification

\[ \exists x . \ f(x, y) \text{ is equivalent to “} f(x, y) \text{ holds for some value of } x \text{”} \]
\[ \forall x . \ f(x, y) \text{ is equivalent to “} f(x, y) \text{ holds for all values of } x \text{”} \]

Boolean quantifiers can be regarded as BDD operations:

\[ \exists x . \ f(x,y) = f \downarrow x \lor f \downarrow \neg x \]
\[ \forall x . \ f(x,y) = f \downarrow x \land f \downarrow \neg x \]
Boolean Quantification Implementation

BDD_exists(x, f) :

if (f.var != x) then
    make_bdd(f.var,
            BDD_exists(x, b.t),
            BDD_exists(x, b.e))
else
    bdd_OR(b.t, b.e));

(but memo-ize, of course)

forall(x,f) is similar
Predicates and Relations

For many applications, it is more convenient to use “word-level” notation.

For this, we can use “vector variables”, which are fixed-length vectors of BDD variables (notation: $\mathbf{x}$). Bit-vector expressions can be represented as vectors of BDDs, $F(\mathbf{x})$.

This is just like bit-blasting with SAT.
A binary relation on $Q$ is a subset of $Q \times Q$. We can represent by creating two copies of the variables: $x$ ("present state vars") and $x'$ ("next state vars").

Then BDD $R(x, x')$ can represent the relation.

**Notation:** $P(x)$ -- predicate on vectors of free variables (BDD vars) $x$ returns a single-bit value.
Predicates and Relations

Example relation: $\mathbf{x} = \mathbf{x}'$, where $\mathbf{x}$ is a vector of Boolean variables:

$$\mathbf{x} = \mathbf{x}' \text{ is } \bigwedge_{1 \leq i \leq n} (x_i \leftrightarrow x_i')$$

“$\leftrightarrow$” is “iff” = “not xor”
Fake First-order logic

- Use vectors variables and vectors of BDDs.
- Functions, predicates, and quantifiers can be defined on vectors.
- Universal and existential quantifiers
- Not really first-order because vectors are fixed-length
  - all variable and expression types are finite
Operations on BDD vectors

Bitwise logical operations: e.g. $a \oplus b$

- $a, b$ must be of same length
- Result is bitvector of XORs of corresponding bits from $a$ and $b$.

Same idea can be used for all other Boolean operators: AND, OR, NAND, etc.
Predicates

- Def: **Support** of a Boolean function is set of variables that can affect value (i.e. vars appearing in BDD).
- Notation: $P(x,y)$: $x,y$ are vectors of variables containing superset of support of $P$.
- $P(y)$ is $P(x)$ with $y$ variables substituted for corresponding $x$ variables.
- Equality: $x = y$: $(x_0=y_0) \land (x_1=y_1) \land ... \land (x_n=y_n)$
Quantifiers

- Boolean quantification operations (previous lecture) can be extended to vectors of variables.
- $\exists \mathbf{x}. P(\mathbf{x}, \mathbf{y}) = \exists x_0 \exists x_1 \ldots P(x_0, x_1, \ldots, y_0, y_1, \ldots)$
- Existential quantification is *projection*
Variable renaming

Variables in BDD predicate can be renamed using the equality relation and existential quantification.

\[ P(y) = \exists x. (x = y) \land P(x) \]

Note that this makes logical sense (when should \( P(y) \) hold?), but also describes BDD operations.
Symbolic Breadth-First Search

• Search a graph without explicitly representing any vertices.
• \( G = \langle V, E \rangle \)
• \( V \) represented with bitvectors of sufficient length.
• \( E \) is a binary relation in \( V \times V \)
• **Reachability**: Find the set of all vertices that are reachable from an arbitrary vertex \( i \).
Symbolic breadth first search

• Encode graph. For convenience, consider $V$ be the set of all bitvectors of length $k$.

• $E(x, y)$ holds if there is an edge from $x$ to $y$.

• $I(x)$ says “$x$ is an initial vertex.”

• Reachability:
  – $R_0(x) = I(x)$
  – $R_{n+1}(x) = R_n(x) \lor \exists y \ (x=y \land \exists x \ [ R_n(x) \land R(x, y) ])$
  – Continue until $R_{n+1}(x) = R_n(x) = \text{all reachable states.}$

• Computing the last thing (the “image computation”) efficiently is the central computational challenge in many verification problems.