15-213
“The course that gives CMU its Zip!”

Verifying Programs with BDDs
Sept. 21, 2004

Topics

- Representing Boolean functions with Binary Decision Diagrams
- Application to program verification
Decision Structures

Truth Table

<table>
<thead>
<tr>
<th>x₁</th>
<th>x₂</th>
<th>x₃</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>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>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>

Decision Tree

- Vertex represents decision
- Follow green (dashed) line for value 0
- Follow red (solid) line for value 1
- Function value determined by leaf value.
Variable Ordering

- Assign arbitrary total ordering to variables
  - e.g., $x_1 < x_2 < x_3$
- Variables must appear in ascending order along all paths

Properties
- No conflicting variable assignments along path
- Simplifies manipulation
Reduction Rule #1

Merge equivalent leaves

\[a\] \rightarrow [a]

\[x_1 \rightarrow x_2 \rightarrow x_3 \rightarrow 0 \rightarrow 0 \rightarrow 1 \rightarrow 0 \rightarrow 1 \rightarrow 0 \rightarrow 1\]

\[x_1 \rightarrow x_2 \rightarrow x_3 \rightarrow 0 \rightarrow 1\]
Reduction Rule #2

Merge isomorphic nodes
Reduction Rule #3

Eliminate Redundant Tests

\[ x \rightarrow y \]

\[ x^2 \cdot x^3 = x^3 \cdot x^2 \]

\[ y \cdot x \cdot x^2 = y \cdot x^3 \cdot x^2 \]

\[ x_1 \cdot x_2 \cdot x_3 = x_1 \cdot x_2 \cdot x_3 \]

\[ x_1 \cdot x_2 = x_2 \cdot x_1 \]

\[ x_1 \cdot x_2 \cdot x_3 = x_3 \cdot x_2 \cdot x_1 \]
Example OBDD

**Canonical representation of Boolean function**

- For given variable ordering
  - Two functions equivalent if and only if graphs isomorphic
    - Can be tested in linear time
  - Desirable property: *simplest form is canonical.*
Example Functions

Constants

- 0: Unique unsatisfiable function
- 1: Unique tautology

Typical Function

- \((x_1 \lor x_2) \land x_4\)
- No vertex labeled \(x_3\)
- Independent of \(x_3\)
- Many subgraphs shared

Variable

- Treat variable as function

Odd Parity

- Linear representation

- Many subgraphs shared
More Complex Functions

Functions
- Add 4-bit words a and b
- Get 4-bit sum s
- Carry output bit Cout

Shared Representation
- Graph with multiple roots
- 31 nodes for 4-bit adder
- 571 nodes for 64-bit adder
- Linear growth!
Apply Operation

Concept
- Basic technique for building OBDD from Boolean formula.

Arguments $A$, $B$, $op$
- $A$ and $B$: Boolean Functions
  - Represented as OBDDs
- $op$: Boolean Operation (e.g., $^\wedge$, $\&$, $|$)

Result
- OBDD representing composite function
- $A \ op \ B$
Apply Execution Example

Optimizations
- Dynamic programming
- Early termination rules
Recursive calling structure implicitly defines unreduced BDD

- Apply reduction rules bottom-up as return from recursive calls
Program Verification

Straight-Line Evaluation

```c
int bitOr(int x, int y)
{
    return ~(~x & ~y);
}
```

```c
int test_bitOr(int x, int y)
{
    return x | y;
}
```

Do these functions produce identical results?

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td></td>
</tr>
<tr>
<td>y</td>
<td></td>
</tr>
<tr>
<td>v1 = ~x</td>
<td></td>
</tr>
<tr>
<td>v2 = ~y</td>
<td></td>
</tr>
<tr>
<td>v3 = v1 &amp; v2</td>
<td></td>
</tr>
<tr>
<td>v4 = ~v3</td>
<td></td>
</tr>
<tr>
<td>v5 = x</td>
<td>y</td>
</tr>
<tr>
<td>t = v4 == v5</td>
<td></td>
</tr>
</tbody>
</table>
Symbolic Execution

(3-bit word size)

\[ v1 = \sim x \]

\[ v2 = \sim y \]
Symbolic Execution (cont.)

\[ v_3 = v_1 \& v_2 \]

\[ v_4 = \neg v_3 \]

\[ v_5 = x | y \]

\[ t = v_4 == v_5 \]
Counterexample Generation

int bitOr(int x, int y)
{
    return ~(~x & ~y);
}

int bitXor(int x, int y)
{
    return x ^ y;
}

Find values of \(x\) & \(y\) for which these programs produce different results

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(x)</td>
<td></td>
</tr>
<tr>
<td>(y)</td>
<td></td>
</tr>
<tr>
<td>(v_1 = \sim x)</td>
<td></td>
</tr>
<tr>
<td>(v_2 = \sim y)</td>
<td></td>
</tr>
<tr>
<td>(v_3 = v_1 &amp; v_2)</td>
<td></td>
</tr>
<tr>
<td>(v_4 = \sim v_3)</td>
<td></td>
</tr>
<tr>
<td>(v_5 = x ^ y)</td>
<td></td>
</tr>
<tr>
<td>(t = v_4 == v_5)</td>
<td></td>
</tr>
</tbody>
</table>
Symbolic Execution

\[ v_4 = \neg v_3 \]

\[ v_5 = x \oplus y \]

\[ t = v_4 == v_5 \]

\[ x = 111 \]
\[ y = 001 \]
Performance: Good

```c
int addXY(int x, int y)
{
    return x+y;
}

int addYX(int x, int y)
{
    return y+x;
}
```
Performance: Bad

```c
int mulXY(int x, int y)
{
    return x*y;
}
```

```c
int mulYX(int x, int y)
{
    return y*x;
}
```

![Graph showing comparison between Enumerate and BDD methods for different word sizes. The x-axis represents word size, and the y-axis represents time in seconds. The graph shows that BDD method outperforms Enumerate method as word size increases.](image-url)
What if Multiplication were Easy?

```
int factorK(int x, int y) {
    int K = XXXX...X;
    int rangeOK =
        1 < x && x <= y;
    int factorOK =
        x*y == K;
    return
        !(rangeOK && factorOK);
}

int one(int x, int y) {
    return 1;
}
```
Evaluation

Strengths
- Provides 100% guarantee of correctness
- Performance very good for Datalab functions

Weaknesses
- Important integer functions have exponential blowup
- Not practical for programs that build and operate on large data structures
Some History

Origins

- Lee 1959, Akers 1976
  - Idea of representing Boolean function as BDD
- Hopcroft, Fortune, Schmidt 1978
  - Recognized that ordered BDDs were like finite state machines
  - Polynomial algorithm for equivalence
- Bryant 1986
  - Proposed as useful data structure + efficient algorithms
- McMillan 1993
  - Developed symbolic model checking
  - Method for verifying complex sequential systems
- Bryant 1991
  - Proved that multiplication has exponential BDD
  - No matter how variables are ordered