CS 267: Automated Verification

Lecture 6: Binary Decision Diagrams

Instructor: Tevfik Bultan
Binary Decision Diagrams (BDDs)  
[Bryant 86]

• Reduced Ordered Binary Decision Diagrams (BDDs)
  – An efficient data structure for representing Boolean functions (or truth sets of Boolean formulas) and manipulating them

• BDDs are a canonical representation for Boolean functions
  – given two Boolean logic formulas F and G, if F and G are equivalent (i.e. if their truth sets are the same), then their BDD representations will be the same
BDDs for Symbolic Model Checking

• BDD data structure can be used to implement the symbolic model checking algorithm we discussed earlier

• BDDs support all the operations we need for symbolic model checking
  – take conjunction of two BDDs
  – take disjunction of two BDDs
  – test equivalence of two BDDs
  – test subsumption between two BDDs
  – negate a BDD
  – test if a BDD satisfiable
  – test if a BDD is a tautology
  – existential variable elimination
Binary Decision Trees

Given a variable order, in each level of the tree, branch on the value of the variable in that level.

• Examples for boolean formulas on two variables

  Variable order: x, y

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>F</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>

x ∨ y

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>F</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>

x ∧ y

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>F</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>

x

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>F</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>

False
Reduced and Ordered Binary Decision Diagrams

• We are interested in Reduced and Ordered Binary Decision Diagrams

• Reduced:
  – Merge all identical sub-trees in the binary decision tree (converts it to a directed-acyclic graph)
  – Remove redundant tests (if the false and true branches for a node go to the same place, remove that node)

• Ordered
  – We pick a fix order for the Boolean variables:
    \[ x_0 < x_1 < x_2 < \ldots \]
  – The nodes in the BDD are listed based on this ordering
BDDs

• Repeatedly apply the following transformations to a binary decision tree:

1. Remove duplicate terminals
2. Remove duplicate non-terminals
3. Remove redundant tests

• These transformations transform the tree to a directed acyclic graph
Binary Decision Trees vs. BDDs

- **$x \lor y$**
  - Tree structure: $x$ (top), $y$ (bottom)
  - Values: $F$, $T$
  - Outcome:
    - Left branch: $F$
    - Right branch: $T$
    - Bottom: $F$, $T$

- **$x \land y$**
  - Tree structure: $x$ (top), $y$ (bottom)
  - Values: $F$, $T$
  - Outcome:
    - Left branch: $F$
    - Right branch: $T$
    - Bottom: $F$, $F$

- **$x$**
  - Tree structure: $x$ (top), $y$ (bottom)
  - Values: $F$, $T$
  - Outcome:
    - Left branch: $F$
    - Right branch: $T$
    - Bottom: $F$, $T$

- **False**
  - Tree structure: $x$ (top), $y$ (bottom)
  - Values: $F$
  - Outcome:
    - Left branch: $F$
    - Right branch: $T$
    - Bottom: $F$


Good News About BDDs

- Given BDDs for two boolean logic formulas $F$ and $G$
  - The BDDs for $F \land G$ and $F \lor G$ are of size $|F| \times |G|$ (and can be computed in that time)
  - The BDD for $\neg F$ is of size $|F|$ (and can be computed in that time)
  - $F \equiv G$ can be checked in linear time
  - Satisfiability of $F$ can be checked in constant time
    - No, this does not mean that you can solve SAT in constant time
Bad News About BDDs

• The size of a BDD can be exponential in the number of boolean variables

• The sizes of the BDDs are very sensitive to the variable ordering. Bad variable ordering can cause exponential increase in the size of the BDD

• There are functions which have BDDs that are exponential for any variable ordering (for example binary multiplication)

• Pre condition (EX) computation requires a sequence of existential variable eliminations
  – A sequence of existential variable eliminations can cause an exponential blow-up in the size of the BDD
BDDs are Sensitive to Variable Ordering

Identity relation for two variables: \((x' \leftrightarrow x) \land (y' \leftrightarrow y)\)

Variable order: \(x, x', y, y'\)  

For \(n\) variables, \(3n+2\) nodes

Variable order: \(x, y, x', y'\)

For \(n\) variables, \(3 \times 2^n - 1\) nodes
BDDs from Another Perspective

• Any Boolean formula \( f \) on variables \( x_1, x_2, \ldots, x_n \) can be written as (called Shannon expansion):

\[
f = x_i \land f [\text{True}/x_i] \lor \neg x_i \land f [\text{False}/x_i]
\]

(this is an if-then-else)

• BDDs use this idea

This node corresponds to the formula \( y \), which comes from the Shannon expansion:

\[
\text{False} \equiv x \land y [\text{False}/x]
\]

This node corresponds to the formula \( y \), which comes from the Shannon expansion:

\[
y \equiv x \land y [\text{True}/x]
\]
How to Implement BDDs?

• First, let’s pick a data structure for a BDD node:

```pascal
type vertex = record
  low, high: vertex;
  index: 1 .. n+1;
  val: {0, 1, X};
  id: integer;
end;
```

High is the true branch, Low is the false branch
Index from the variable ordering, each index value corresponds to one variable
n is the number of Boolean variables
0 corresponds to false, 1 corresponds to true, X corresponds to no value
This field will be used to generate a unique id for each unique function
How to Implement a BDD?

- A BDD is a rooted directed graph with vertex set $V$ containing two types of vertices
  - nonterminal vertices:
    - $\text{index} \in \{1, 2, \ldots, n\}$
    - $\text{low}(v)$ and $\text{high}(v)$ are in $V$
    - $\text{val}(v)$ is $X$
  - terminal vertices:
    - $\text{index}(v) = n+1$
    - $\text{val}(v)$ is 0 or 1
    - $\text{low}(v)$ and $\text{high}(v)$ are null
BDD Reduce algorithm

- Start from the terminal nodes and go towards to root
  1. Remove duplicate terminals: If two terminal vertices have the same value, remove one of them redirect all the incoming arcs to the other one
  2. Remove duplicate non-terminals: If two nonterminal vertices, v and u have the same index and v.low = u.low and v.high = u.high then eliminate one of them and redirect all incoming arcs to the other one
  3. Remove redundant tests: If a nonterminal vertex v has v.low = v.high then remove v and redirect all incoming arcs to low(v)
Reduce Algorithm

function Reduce(v: vertex): vertex;
  var subgraph: array[1 .. |G|] of vertex;
  var vlist: array[1 .. n+1] of list of vertex;

begin
  Put each vertex u reachable from v on list vlist[u.index]
  nextid := 0;

  for i := n+1 downto 1 do
  begin
    Q := empty set;

    for each u in vlist[i] do
      if u.index = n+1
      then add <key,u> to Q where key=(u.value) {terminal}
      else if u.low.id=u.high.id
      then u.id := u.low.id; {redundant vertex}
      else add <key,u> to Q where key =(u.low.id,u.high.id);
  end
end
Sort elements of Q by keys;

oldkey := (-1,-1); {unmatchable key}
for each <key,u> in Q removed in order do
    if key = oldkey
        then u.id := nextid; {matches existing vertex}
    else begin {unique vertex}
        nextid :=nextid+1;
        u.id:=nextid;
        subgraph[nextid]:=u;
        u.low := subgraph[u.low.id];
        u.high := subgraph[u.high.id];
        oldkey := key;
    end;
end; {end of the outmost for loop}
return(subgraph[v.id]);
end;
Reduce Algorithm

• Reduce algorithm can be done in linear time
  – There is a linear time lexicographic sorting method based on bucket sorting for sorting the vertices according to their keys
  – Assuming a linear time sorting routine, processing at each level requires time proportional to the number of vertices at that level, and each level is processed once, resulting in linear time complexity
Apply Algorithm

• We can use the Shannon expansion to recursively compute any binary Boolean operation

• Given BDDs for f and g and binary operation op

\[ f \text{ op } g = x_i \land (f[T/x_i] \text{ op } g[T/x_i]) \lor \neg x_i \land (f[F/x_i] \text{ op } g[F/x_i]) \]

• So the idea is to start at the root of f and g and recursively call true and false branches
  – Key idea: Use memoization!
Apply Algorithm

var T: array[1 .. |G1|, 1 .. |G2|] of vertex; {init. to null}
function Apply(v1, v2: vertex; <op> operator): vertex;
begin
  u := T[v1.id, v2.id];
  if u != null then return(u); {already computed}
  u := new vertex; T[v1.id, v2.id] := u;
  u.value := v1.value <op> v2.value;
  if u.value != X
    then u.index := n+1; u.low :=null; u.high :=null; {terminal}
    else begin {nonterminal}
      u.index := Min(v1.index, v2.index);
      if v1.index = u.index
        then vlow1 := v1.low; vhigh1 := v1.high;
      else vlow1 := v1; vhigh1 := v1;
      if v2.index = u.index
        then vlow2 := v2.low; vhigh2 :=v2.high;
      else vlow2 := v2; vhigh2 := v2;
      u.low := Apply(vlow1, vlow2);
      u.high := Apply(vhigh1, vhigh2);
    end;
  return(u);
end;
Apply Algorithm

- Given two input BDDs $f$ and $g$ the size of the resulting BDD for $f \langle \text{op} \rangle g$ is $|f| \times |g|$ and it can be computed in that time.

- The important point is to use memoization so that for any pair of vertices (one from each input BDD) the computation is done exactly once.

- Since there are at most $|f| \times |g|$ pairs the complexity is $|f| \times |g|$.
  - We do constant amount of work for each pair of nodes.