EECS 219C: Computer-Aided Verification
Boolean Satisfiability Solving III
& Binary Decision Diagrams

Sanjit A. Seshia
EECS, UC Berkeley

Acknowledgments: Lintao Zhang

Announcement

• Project proposals due Feb 16
  – If you have a project topic (or area) in mind, discuss it with me in the next week
• Selected suggested project topics will be discussed in class/posted by next week.
• Homework 1 will be out ~ Feb 10.
DLL Algorithm Pseudo-code

```
DLL_iterative()
{
    status = preprocess();
    if (status!=UNKNOWN)
        return status;
    while(1) {
        decide_next_branch();
        while (true)
        {
            status = deduce();
            if (status == CONFLICT)
                {
                    blevel = analyze_conflict();
                    if (blevel < 0)
                        return UNSATISFIABLE;
                    else
                        backtrack(blevel);
                }
            else if (status == SATISFIABLE)
                return SATISFIABLE;
            else break;
        }
    }
}
```

Main Steps:
- Pre-processing
- Branching
- Unit propagation (apply unit rule)
- Conflict Analysis & Backtracking
A Classification of SAT Algorithms

• Davis-Putnam (DP)
  – Based on resolution

• Davis-Logemann-Loveland (DLL/DPLL)
  – Search-based
  – Basis for current most successful solvers

• Stalmarck’s algorithm
  – More of a “breadth first” search, proprietary algorithm

• Stochastic search
  – Local search, hill climbing, etc.
  – Unable to prove unsatisfiability (incomplete)

• Stalmarck’s algorithm: described on whiteboard (see reading for more details)

→Note: Stalmarck’s algorithm is for tautology (validity) checking:
  Is formula F always 1 (on any input assignment)?
Proof Generation

• If the SAT solver returns “satisfiable”, we can check that solution by evaluating the circuit
• If it returns “unsatisfiable”, what then?

Proof

• Starting from facts (clauses), the SAT solver has presumably derived “unsatisfiable” (the empty clause)
• So there must be a way of going step-by-step from input clauses to the empty clause using rules
  – In fact, there’s only one rule: **resolution**
Resolution as a Cut in Implication Graph

\[(V_2 + V_3' + V_5' + V_6)\]

\[(V_3' + V_6 + V_4)\]
\[(V_6' + V_5' + V_1')\]
\[(V_2' + V_4' + V_6 + V_1)\]
\[(V_2' + V_4' + V_6 + V_5')\]
\[(V_2 + V_3' + V_5' + V_6)\]

Resolution Graph

- Nodes are clauses
- Edges are applications of resolution
Proof Checker

• Given resolution graph, how to check it?
• Traverse it, checking that each node is correctly obtained from its predecessor nodes using resolution
Incremental SAT Solving

• Suppose you have not just one SAT problem to solver, but many “slightly differing” problems over the same variables
• Can we re-use the search over many problems?
  – i.e. perform only “incremental” work

Operations Needed

1. Adding clauses
2. Deleting clauses

• Which is easy and which is hard?
  – If previous problem is unsat, how does an operation change it?
  – If previous is sat?
Deleting Clauses

S. A. Seshia

Original Clauses
Learned Clauses
Engineering Issues

• Too expensive to traverse graph
• Instead, group original clauses into groups
• Each derived clause belongs to all groups that it is resolved from
  – Implement with bit-vector

Binary Decision Diagrams
Boolean Function Representations

- Syntactic: e.g.: CNF, DNF, Circuit
- Semantic: e.g.: Truth table, Binary Decision Tree, BDD

Reduced Ordered BDDs

- Invented by Randal E. Bryant in mid-80s
  - IEEE Transactions on Computers 1986 paper is one of the most highly cited papers in EECS
- Useful data structure to represent Boolean functions
  - Applications in synthesis, verification, program analysis, planning, …
- Commonly known simply as BDDs
- Many variants of BDDs have proved useful in other tasks
- Links to coding theory (trellises), etc.
Cofactors

• A Boolean function $F$ of $n$ variables $x_1, x_2, \ldots, x_n$
  
  $F : \{0,1\}^n \rightarrow \{0,1\}$

• Suppose we define new Boolean functions of $n-1$ variables as follows:
  
  $F_{x_1}(x_2, \ldots, x_n) = F(1, x_2, x_3, \ldots, x_n)$
  
  $F_{x_1'}(x_2, \ldots, x_n) = F(0, x_2, x_3, \ldots, x_n)$

• $F_{x_1}$ and $F_{x_1'}$ are called cofactors of $F$.
  $F_{x_1}$ is the positive cofactor, and $F_{x_1'}$ is the negative cofactor

Shannon Expansion

• $F(x_1, \ldots, x_n) = x_i \cdot F_{x_i} + x_i' \cdot F_{x_i'}$

• Proof?
Shannon expansion with many variables

- \( F(x, y, z, w) = \)
  \( xy F_{xy} + x'y F_{x'y} + xy' F_{xy'} + x'y' F_{x'y'} \)

Properties of Cofactors

- Suppose you construct a new function \( H \) from two existing functions \( F \) and \( G \): e.g.,
  - \( H = F' \)
  - \( H = F \cdot G \)
  - \( H = F + G \)
  - Etc.

- What is the relation between cofactors of \( H \) and those of \( F \) and \( G \)?
Very Useful Property

- Cofactor of NOT is NOT of cofactors
- Cofactor of AND is AND of cofactors
- …
- Works for any binary operator

BDDs from Truth Tables

Truth Table → Binary Decision Tree → Binary Decision Diagram (BDD) → Ordered Binary Decision Diagram (OBDD) → Reduced Ordered Binary Decision Diagram (ROBDD, simply called BDD)
Example: Odd Parity Function

Nodes & Edges

- How is $F$ related to $\alpha$, $F_1$, $F_2$?
Ordering

Impose arbitrary order:
\[ a < b < c < d \]

Why didn’t this part change?

Reduction

- Identify Redundancies

- 3 Rules:
  1. Merge equivalent leaves
  2. Merge isomorphic nodes
  3. Eliminate redundant tests
Merge Equivalent Leaves

"a" is either 0 or 1

Merge Isomorphic Nodes

Redirect

stays same down here
Eliminate Redundant Tests

Example
Example

Final ROBDD for Odd Parity Function
What can BDDs be used for?

- Uniquely representing a Boolean function
  - And a Boolean function can represent sets

- Satisfiability solving!
(RO)BDDs are canonical

- Theorem (R. Bryant): If $G$, $G'$ are ROBDD's of a Boolean function $f$ with $k$ inputs, using same variable ordering, then $G$ and $G'$ are identical.

Sensitivity to Ordering

- Given a function with $n$ inputs, one input ordering may require exponential # vertices in ROBDD, while other may be linear in size.
- Example: $f = x_1 x_2 + x_3 x_4 + x_5 x_6$
  - $x_1 < x_2 < x_3 < x_4 < x_5 < x_6$
  - $x_1 < x_4 < x_5 < x_2 < x_3 < x_6$
Creating BDDs

• Idea: Build a few core operators and define everything else in terms of those

Advantage:
• Less programming work
• Easier to add new operators later by writing “wrappers”

Core Operators

• Just two of them!
  1. Restrict(Function F, variable v, constant k)
      • Shannon cofactor of F w.r.t. v=k
  2. ITE(Function I, Function T, Function E)
      • “if-then-else” operator
ITE

- Just like:
  - “if then else” in a programming language
  - A mux in hardware
- ITE(I(x), T(x), E(x))
  - If I(x) then T(x) else E(x)

\[
ITE(I(x), T(x), E(x)) = I(x) \cdot T(x) + I'(x) \cdot E(x)
\]
What good is the ITE?

• How do we express
  • NOT?

• OR?

• AND?

How do we implement ITE?

• Divide and conquer!

• Use Shannon cofactoring…
  • Recall: Operator of cofactors is Cofactor of operators…
ITE Algorithm

ITE (bdd I, bdd T, bdd E) {
    if (terminal case) { return computed result; }
    else { // general case
        Let x be the topmost variable of I, T, E;
        PosFactor = ITE(Iₙ, Tₙ, Eₙ);
        NegFactor = ITE(Iₙ', Tₙ', Eₙ');
        R = new node labeled by x;
        R.low = NegFactor; // set 0-child
        R.high = PosFactor; // set 1-child
        Reduce(R); // apply reduction rules
        return R;
    }
}

Terminal Cases

- ITE(1, T, E) =
- ITE(0, T, E) =
- ITE(I, T, T) =
- ITE(I, 1, 0) =
- …
General Case

• Still need to do cofactor (Restrict)

• Easy operation in the case of ITE
  – We are cofactoring w.r.t. the top variable only
  – Cofactoring w.r.t. an internal variable is trickier

Practical Issues

• Previous calls to ITE are cached
  – “memoization”

• Every BDD node created goes into a “unique table”
  – Before creating a new node R, look up this table
  – Avoids need for reduction
Sharing: Multi-Rooted DAG

- BDD for 4-bit adder
- Each output bit (of the sum & carry) is a distinct rooted BDD
- But they share sub-DAGs

More on BDDs

- Circuit width and bounds on BDD size
- Dynamically changing variable ordering
- Some BDD variants
Bounds on BDD Size: Warm-up

• Suppose the number of nodes at any level in a BDD is bounded above by $B$
• Then, what is an upper bound on the total number of nodes in the BDD?

Cross-section of a BDD at level $i$

• Suppose a BDD represents Boolean function $F(x_1, x_2, \ldots, x_n)$ with variable order $x_1 < x_2 < \ldots < x_n$
• Size of cross section of the BDD at level $i$ is the number of distinct Boolean functions $F'$ that depend on $x_i$ given by
  $$F'(x_i, x_{i+1}, \ldots, x_n) = F(v_1, v_2, \ldots, v_{i-1}, x_i, \ldots, x_n)$$
  for some Boolean constants $v_i$'s (in $\{0,1\}$)
Circuit Width

- Consider a circuit representation of a Boolean function $F$
- Impose a linear order on the gates of the circuit
  - Primary inputs and outputs are also considered as "gates" and primary output is at the end of the ordering
  - Forward cross section at a gate $g$: set of wires going from output of $g_1$ to input of $g_2$ where $g_1 \leq g < g_2$
  - Similarly define reverse cross section: set of wires going from output of $g_1$ to input of $g_2$ where $g_2 \leq g < g_1$
  - Forward width ($w_f$): maximum forward cross section size
  - Similarly, reverse width $w_r$

BDD Upper Bounds from Circuit Widths

- Theorem: Let a circuit representing $F$ with $n$ variables have forward width $w_f$ and reverse width $w_r$, for some linear order $L$ on its gates. Then, there is a BDD representing $F$ of size bounded above by

$$n \cdot 2^{w_f} \cdot 2^{w_r}$$
BDD Ordering in Practice

- If we can derive a small upper bound using circuit width, then that’s fine
  - Use the corresponding linear order on the variables
- What if we can’t?

- There are many BDD variable ordering heuristics around, but the most common way to deal with variable ordering is to start with something “reasonable” and then swap variables around to improve BDD size
  - DYNAMIC VARIABLE REORDERING → SIFTING

Sifting

- Dynamic variable re-ordering, proposed by R. Rudell
- Based on a primitive “swap” operation that interchanges $x_i$ and $x_{i+1}$ in the variable order
  - Key point: the swap is a local operation involving only levels $i$ and $i+1$
- Overall idea: pick a variable $x_i$ and move it up and down the order using swaps until the process no longer improves the size
  - A “hill climbing” strategy
Some BDD Variants

• **Free BDDs (FBDDs)**
  – Relax the restriction that variables have to appear in the same order along all paths
  – How can this help?
  – Is it canonical?

Some BDD Variants

• **MTBDD (Multi-Terminal BDD)**
  – Represents function of Boolean variables with non-Boolean value (integer, rational)
    • E.g., input-dependent delay in a circuit, transition probabilities in a Markov chain
  – Similar reduction / construction rules to BDDs
Some BDD packages

- CUDD – from Colorado University, Fabio Somenzi’s group
- BuDDy – from IT Univ. of Copenhagen