EECS 219C: Computer-Aided Verification

Boolean Satisfiability Solving Part II: DPLL-based (CDCL) Solvers

Sanjit A. Seshia
EECS, UC Berkeley

With thanks to Lintao Zhang (MSR)

A Classification of SAT Algorithms

- Davis-Putnam (DP)
  - Based on **resolution**
- Davis-Logemann-Loveland (DLL/DPLL)
  - Search-based
  - Basis for current most successful solvers
  - Also called "Conflict Driven Clause Learning" (CDCL)
- Stalmarck's algorithm
  - More of a "breadth first" search, proprietary algorithm
- Stochastic search
  - Local search, hill climbing, etc.
  - Unable to prove unsatisfiability (incomplete)

Resolution

- Two CNF clauses that contain a variable x in opposite phases (polarities) imply a new CNF clause that contains all literals except x and x'

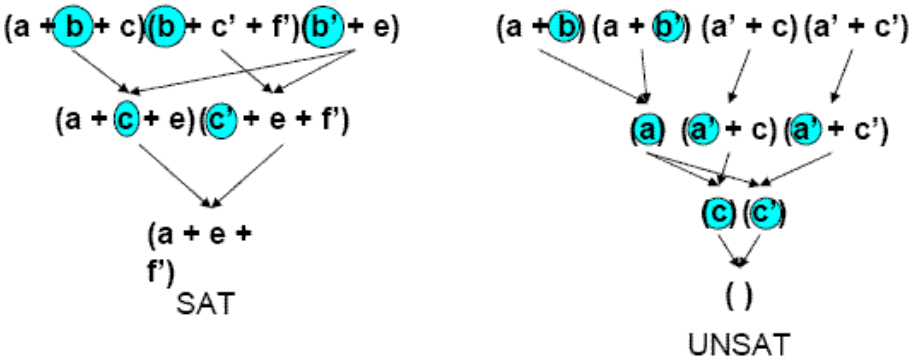
  $(a + b)\ (a' + c) \Rightarrow (b + c)$

  $(a + b)\ (a' + c) = (a + b)(a' + c)(b + c)$
- Why is this true?

The Davis-Putnam Algorithm

- Iteratively select a variable x to perform resolution on
- Retain only the newly added clauses and the ones not containing x
- Termination: You either
  - Derive the empty clause (conclude UNSAT)
  - Or all variables have been selected

Resolution Example



How many clauses can you end up with?
(at any iteration)

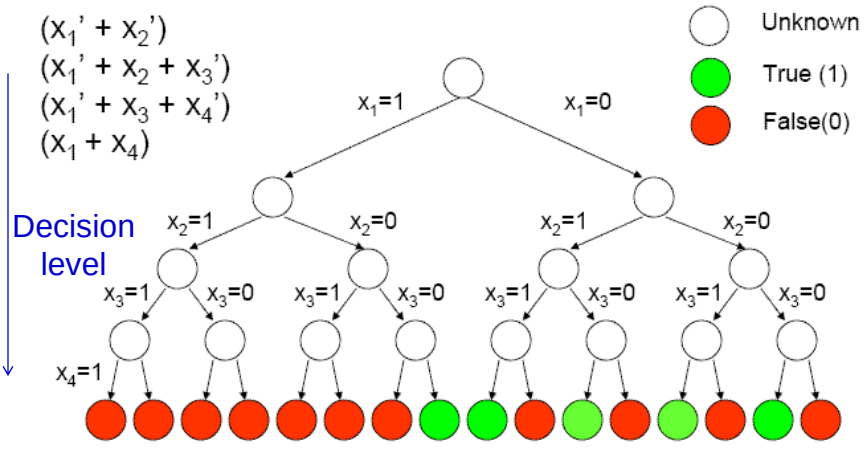
A Classification of SAT Algorithms

- Davis-Putnam (DP)
  - Based on **resolution**
- Davis-Logemann-Loveland (DLL/DPLL)
  - Search-based
  - Basis for current most successful solvers
  - Also called "Conflict Driven Clause Learning" (CDCL)
- Stalmarck's algorithm
  - More of a "breadth first" search, proprietary algorithm
- Stochastic search
  - Local search, hill climbing, etc.
  - Unable to prove unsatisfiability (incomplete)

DLL/CDCL Algorithms Today: Exploration + Generalization

- EXPLORATION: Iteratively set variables until
  - you find a satisfying assignment (done!)
  - you reach a conflict (backtrack and try different value)
- GENERALIZATION
  - When a conflict is reached, LEARN a clause that "remembers" the reason for the conflict
- This paradigm underlies much of the recent advances in Verification technology

Search Tree



DLL Example 1

DLL Algorithm Pseudo-code (Chaff)

```
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

Pre-processing: Pure Literal Rule

- If a variable appears in only one phase throughout the problem, then you can set the corresponding literal to 1
  - E.g. if we only see a' in the CNF, set a' to 1 (a to 0)
- Why is this sound?

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

Conflicts & Backtracking

- Chronological Backtracking
  - Proposed in original DLL paper
  - Backtrack to highest (largest) decision level that has not been tried with both values
    - But does this decision level have to be the reason for the conflict?

Non-Chronological Backtracking

- Jump back to a decision level "higher in the search tree" than the last one
- Also combined with "conflict-driven learning"
  - Keep track of the reason for the conflict
- Proposed by Marques-Silva and Sakallah in 1996
  - Similar work by Bayardo and Schrag in '97

DLL Example 2

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

Branching

- Which variable (literal) to branch on (set)?
- This is determined by a “decision heuristic”
- What makes a “decision heuristic” good?

Decision Heuristic Desiderata

- If the problem is satisfiable
  - Find a short partial satisfying assignment
  - GREEDY: If setting a literal will satisfy many clauses, it might be a good choice
- If the problem is unsatisfiable
  - Reach conflicts quickly (rules out bigger chunks of the search space)
  - Similar to above: need to find a short partial falsifying assignment
- Also: Heuristic must be cheap to compute!

Sample Decision Heuristics

- RAND
  - Pick a literal to set at random
  - What's good about this? What's not?
- Dynamic Largest Individual Sum (DLIS)
  - Let cnt(l) = number of occurrences of literal l in unsatisfied clauses
  - Set the l with highest cnt(l)
  - What's good about this heuristic?
  - Any shortcomings?

DLIS: A Typical Old-Style Heuristic

- Advantages
  - Simple to state and intuitive
  - Targeted towards satisfying many clauses
  - Dynamic: Based on current search state
- Disadvantages
  - Very expensive!
  - Each time a literal is set, need to update counts for all other literals that appear in those clauses
  - Similar thing during backtracking (unsetting literals)
- Even though it is dynamic, it is "Markovian" – somewhat static
  - Is based on current state, without any knowledge of the search path to that state

VSIDS: The Chaff SAT solver heuristic

- Variable State Independent Decaying Sum
  - For each literal l, maintain a VSIDS score
  - Initially: set to cnt(l)
  - Increment score by 1 each time it appears in an added (conflict) clause
  - Divide all scores by a constant (2) periodically (every N backtracks)
- Advantages:
  - Cheap: Why?
  - Dynamic: Based on search history
  - Steers search towards variables that are common reasons for conflicts (and hence need to be set differently)

Current State of Heuristics

- VSIDS has been improved upon, but mostly minor improvements
  - E.g. MiniSat (2006 champion) decays score after each conflict by a smaller fraction (5%)

Key Ideas so Far

- Data structures: Implication graph
- Conflict Analysis: Learn (using cuts in implication graph) and use non-chronological backtracking
- Decision heuristic: must be dynamic, low overhead, quick to conflict/solution

- Principle: Keep #(memory accesses)/step low
  - A step → a primitive operation for SAT solving, such as a branch

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

Unit Propagation

- Also called Boolean constraint propagation (BCP)
- Set a literal and propagate its implications
  - Find all clauses that become unit clauses
  - Detect conflicts
- Backtracking is the reverse of BCP
  - Need to unset a literal and 'rollback'
- In practice: Most of solver time is spent in BCP
  - Must optimize!

BCP

- Suppose literal l is set. How much time will it take to propagate just that assignment?
- How do we check if a clause has become a unit clause?
- How do we know if there's a conflict?

- Introductory BCP slides

Detecting when a clause becomes unit

- Watch only two literals per clause. Why does this work?
- If one of the watched literals is assigned 0, what should we do?
- A clause has become unit if
  - Literal assigned 0 *must* continue to be watched, other watched literal unassigned
- What if other watched literal is 0?
- What if a watched literal is assigned 1?

- Lintao's BCP example

2-literal Watching

- In a L-literal clause, L $\geq$ 3, which 2 literals should we watch?

Comparison:

Naïve 2-counters/clause vs 2-literal watching

- When a literal is set to 1, update counters for all clauses it appears in
- Same when literal is set to 0
- If a literal is set, need to update each clause the variable *appears* in
- During backtrack, must update counters

- No need for update
- Update watched literal
- If a literal is set to 0, need to update only each clause it is *watched* in
- No updates needed during backtrack! (why?)

Overall effect: Fewer clauses accesses in 2-lit

zChaff Relative Cache Performance

		1dlx_c_mc_ex_bp_f		Hanoi4	
		Num Access	Miss Rate	Num Access	Miss Rate
Z-Chaff	L1	24,029,356	4.75%	364,782,257	5.38%
	L2	1,659,877	4.63%	30,396,519	11.65%
SATO (-g100)	L1	188,352,975	36.76%	465,160,957	41.76%
	L2	79,422,894	9.74%	202,495,679	16.77%
Grasp	L1	415,572,501	32.89%	876,250,978	32.53%
	L2	153,490,555	50.25%	335,713,542	51.15%

The programs are compiled with –O3 using g++ 2.8.1(for GRASP and Chaff) or gcc 2.8.1 (for Sato3.2.1) on Sun OS 4.1.2 Trace was generated with QPT quick tracing and profiling tool. Trace was processed with dineroIV, the memory configuration is similar to a Pentium III processor:
L1: 16K Data, 16K Instruction, L2: 256k Unified. Both have 32 byte cache line, 4 way set associativity.

Key Ideas in Modern DLL SAT Solving

- Data structures: Implication graph
- Conflict Analysis: Learn (using cuts in implication graph) and use non-chronological backtracking
- Decision heuristic: must be dynamic, low overhead, quick to conflict/solution
- Unit propagation (BCP): 2-literal watching helps keep memory accesses down

- Principle: Keep #(memory accesses)/step low
  - A step → a primitive operation for SAT solving, such as a branch

Other Techniques

- Random Restarts
  - Periodically throw away current decision stack and start from the beginning
    - Why will this change the search on restart?
  - Used in most modern SAT solvers
    - Recently found effective to do this more often [Biere, 2007]
- Clause deletion
  - Conflict clauses take up memory
    - What's the worst-case blow-up?
  - Delete periodically based on some heuristic ("age", length, etc.)

Next Class

- Finishing up SAT: other techniques, incremental SAT, proof generation.
- (if time permits) Start BDDs