

EECS 219C: Computer-Aided Verification

Boolean Satisfiability Solving

Sanjit A. Seshia
EECS, UC Berkeley

The Boolean Satisfiability Problem (SAT)

- Given:
 - A Boolean formula $F(x_1, x_2, x_3, \dots, x_n)$
- Can F evaluate to 1 (true)?
 - Is F satisfiable?
 - If yes, return values to x_i 's (satisfying assignment) that make F true

Why is SAT important?

- Theoretical importance:
 - First NP-complete problem (Cook, 1971)
- Many practical applications:
 - Model Checking
 - Automatic Test Pattern Generation
 - Combinational Equivalence Checking
 - Planning in AI
 - Automated Theorem Proving
 - Software Verification
 - ...

Terminology

- Variable, Literal
- Operators: AND, OR, NOT
- Clause, Cube/Monomial
- Conjunctive Normal Form (CNF)
- Disjunctive Normal Form (DNF)

An Example

- Inputs to SAT solvers are usually represented in CNF

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

An Example

- Inputs to SAT solvers are usually represented in CNF

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

Why CNF?

Why CNF?

- Input-related reason
 - Can transform from circuit to CNF in linear time & space (HOW?)
- Solver-related: Most SAT solver variants can exploit CNF
 - Easy to detect a conflict
 - Easy to remember partial assignments that don't work (just add 'conflict' clauses)
 - Other “ease of representation” points?
- Any reasons why CNF might NOT be a good choice?
 - Loses structural information present in circuits
 - Some relations, such as XOR, might be better exposed to the solver

Complexity of k-SAT

- A SAT problem with input in CNF with at most k literals in each clause
- Complexity for non-trivial values of k :
 - 2-SAT: in P
 - 3-SAT: NP-complete
 - > 3 -SAT: ?

Worst-Case Complexity

The WORST-CASE SCENARIO Survival Handbook



Beyond Worst-Case Complexity

- What we really care about is “typical-case” complexity
- But how can one measure “typical-case”?
- Two approaches:
 - Is your problem a restricted form of 3-SAT?
That might be polynomial-time solvable
 - Experiment with (random/domain-specific) SAT instances and analyze solver run-time vs formula parameters (#vars, #clauses, ...)

Special Cases of 3-SAT that are polynomial-time solvable

- 2-SAT
 - T. Larrabee observed that many clauses in ATPG tend to be 2-CNF
- Horn-SAT
 - A clause is a Horn clause if at most one literal is positive
 - If all clauses are Horn, then problem is Horn-SAT
 - E.g. Application:- Checking that one finite-state system refines (implements) another

2-SAT Algorithm

- Linear-time algorithm (Aspvall, Plass, Tarjan, 1979)
 - Think of clauses as implications
 - Think of a graph with literals as nodes
 - Find strongly connected components
 - Variable and its negation should not be in the same component

- Example 1:

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

- Example 2:

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

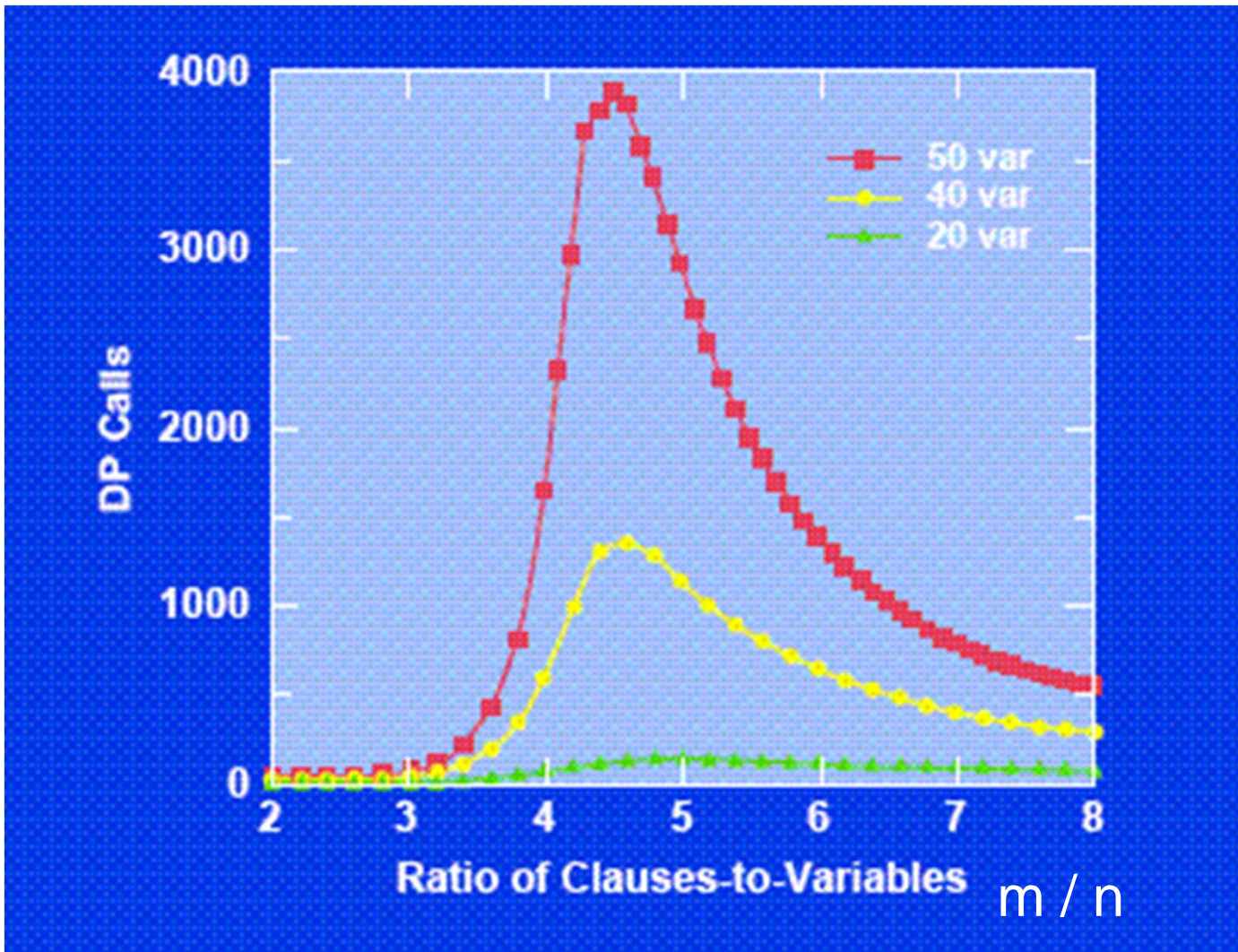
Horn-SAT

- Can we solve Horn-SAT in polynomial time? How? [homework]
 - Hint: again view clauses as implications.
- Variants:
 - Negated Horn-SAT: Clauses with at most one literal negative
 - Renamable Horn-SAT: Doesn't look like a Horn-SAT problem, but turns into one when polarities of some variables are flipped

Phase Transitions in k-SAT

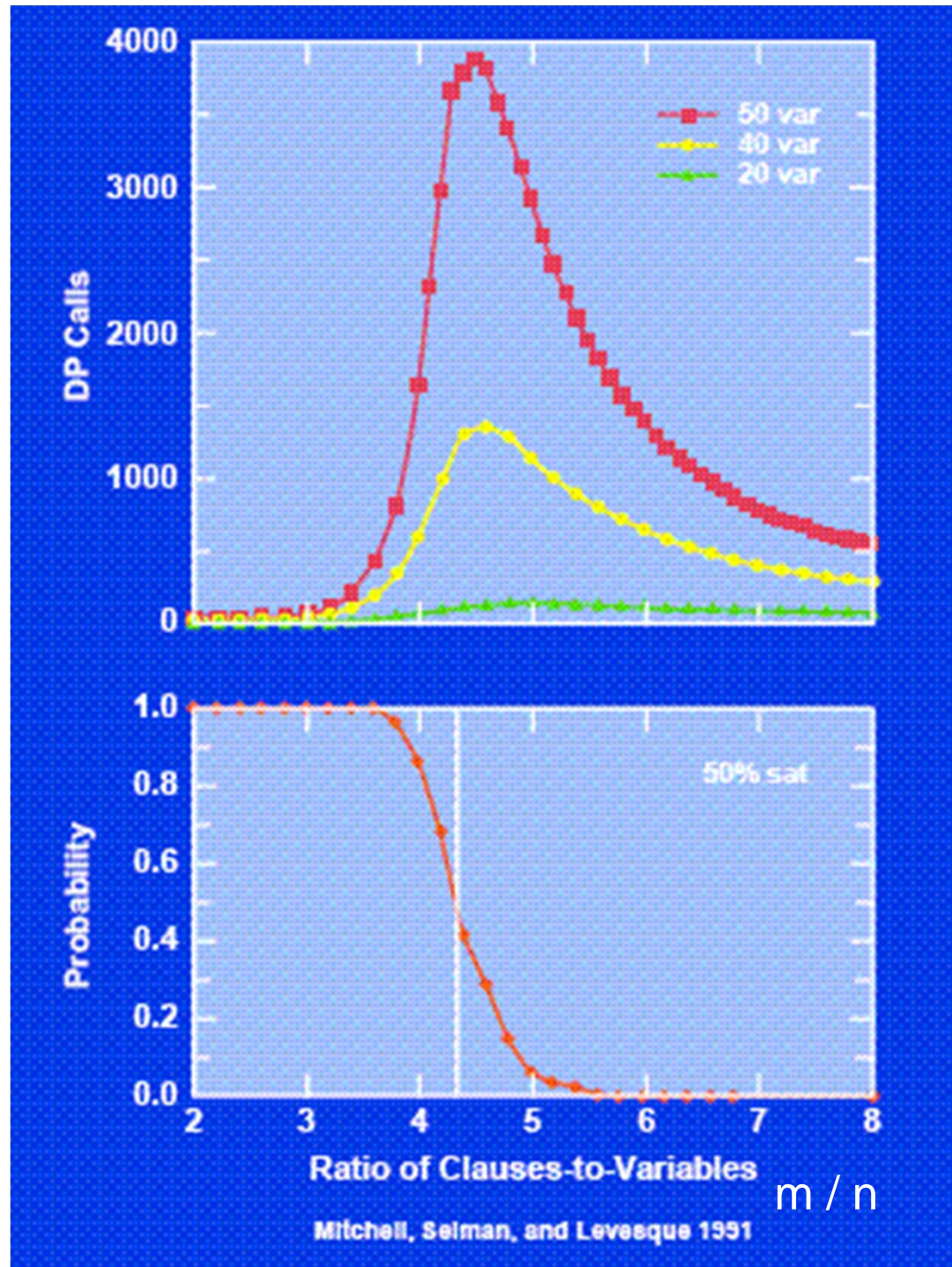
- Consider a fixed-length clause model
 - k-SAT means that each clause contains exactly k literals
- Let SAT problem comprise m clauses and n variables
 - Randomly generate the problem for fixed k and varying m and n
- Question: How does the problem hardness vary with m/n ?

3-SAT Hardness



As n increases
hardness
transition
grows sharper

Transition at $m/n \sim 4.3$



Threshold Conjecture

- For every k , there exists a c^* such that
 - For $m/n < c^*$, as $n \rightarrow \infty$, problem is satisfiable with probability 1
 - For $m/n > c^*$, as $n \rightarrow \infty$, problem is unsatisfiable with probability 1
- Conjecture proved true for $k=2$ and $c^*=1$
- For $k=3$, current status is that c^* is in the range 3.42 – 4.51

The $(2+p)$ -SAT Model

- We know:
 - 2-SAT is in P
 - 3-SAT is in NP
- Problems are (typically) a mix of binary and ternary clauses
 - Let $p \in \{0,1\}$
 - Let problem comprise $(1-p)$ fraction of binary clauses and p of ternary
 - So-called $(2+p)$ -SAT problem

Experimentation with random (2+p)-SAT

- When $p < \sim 0.41$
 - Problem behaves like 2-SAT
 - Linear scaling
- When $p > \sim 0.42$
 - Problem behaves like 3-SAT
 - Exponential scaling
- Intriguing observations, but don't help us predict behavior of problems in practice

Backbones and Backdoors

- Backbone [Parkes; Monasson et al.]
 - Subset of literals that must be true in every satisfying assignment (if one exists)
 - Empirically related to hardness of problems
- Backdoor [Williams, Gomes, Selman]
 - Subset of variables such that once you've given those a suitable assignment (if one exists), the rest of the problem is poly-time solvable
 - Also empirically related to hardness
- But no easy way to find such backbones / backdoors! ☹️

A Classification of SAT Algorithms

- Davis-Putnam (DP)
 - Based on **resolution**
- Davis-Logemann-Loveland (DLL/DPLL)
 - Search-based
 - Basis for current most successful solvers (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'

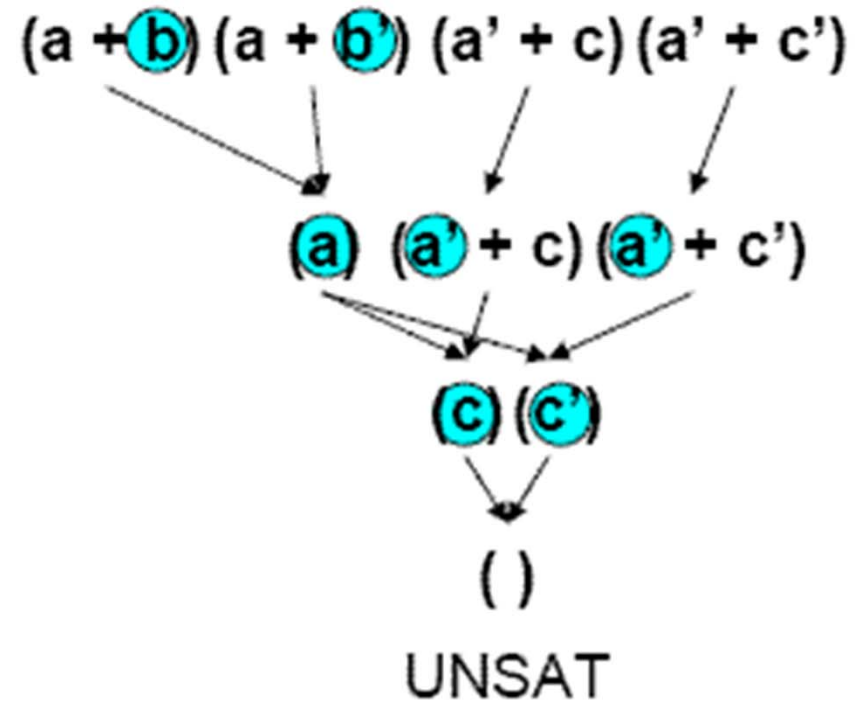
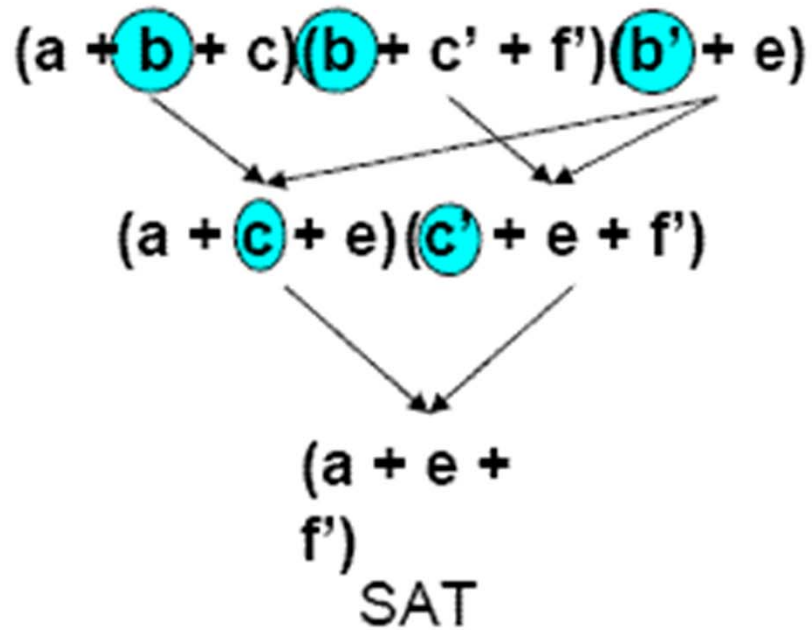
$$(a + b) (a' + c) = (a + b)(a' + c)(b + c)$$

- 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)**

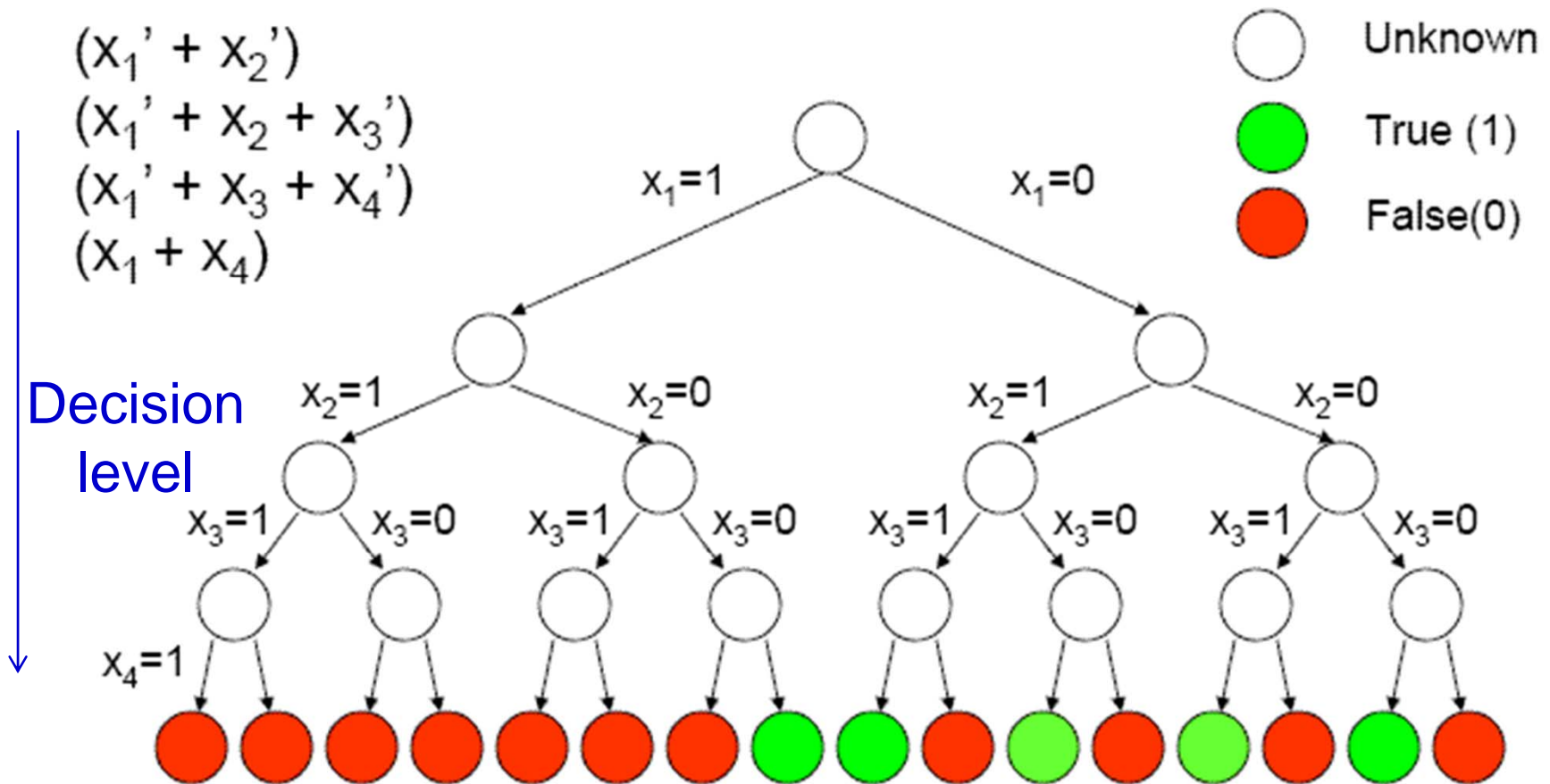
A Classification of SAT Algorithms

- Davis-Putnam (DP)
 - Based on **resolution**
- Davis-Logemann-Loveland (DLL/DPLL)
 - Search-based
 - Basis for current most successful solvers (**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 Algorithm: General Ideas

- Iteratively set variables until
 - you find a satisfying assignment (done!)
 - you reach a conflict (backtrack and try different value)
- Two main rules:
 - *Unit Literal Rule*: If an unsatisfied clause has all but 1 literal set to 0, the remaining literal must be set to 1
 $(a + b + c) (d' + e) (a + c' + d)$
 - *Conflict Rule*: If all literals in a clause have been set to 0, the formula is unsatisfiable along the current assignment path

Search Tree



DLL Example 1

DLL Algorithm Pseudo-code

```
DLL_iterative()
{
  Preprocess  status = preprocess();
              if (status!=UNKNOWN)
                return status;
              while(1) {
  Branch      decide_next_branch();
              while (true)
  {
  Propagate   status = deduce();
  implications of that
  branch and deal with conflicts
              if (status == CONFLICT)
  {
                blevel = analyze_conflict();
                if (blevel < 0)
                  return UNSATISFIABLE;
                else
                  backtrack(blevel);
              }
              else if (status == SATISFIABLE)
                return SATISFIABLE;
              else break;
  }
  }
}
```

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

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?

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” 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(); ← Pre-processing
    if (status!=UNKNOWN)
        return status;
    while(1) {
        decide_next_branch(); ← Branching
        while (true)
        {
            status = deduce(); ← Unit propagation
            if (status == CONFLICT)
            {
                blevel = analyze_conflict(); ← Conflict Analysis
                if (blevel < 0)
                    return UNSATISFIABLE;
                else
                    backtrack(blevel); ← & Backtracking
            }
            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 $\text{cnt}(l)$ = number of occurrences of literal l in unsatisfied clauses
 - Set the l with highest $\text{cnt}(l)$
 - 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 $\text{cnt}(l)$
 - 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)

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 \rightarrow 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 \rightarrow 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
- Clause deletion
 - Conflict clauses take up memory
 - What's the worst-case blow-up?
 - Delete periodically based on some heuristic (“age”, length, etc.)
- Preprocessing/“Inprocessing” and Rewriting techniques give a lot of performance improvements in recent solvers