Satisfiability Solving and Tools

[original created by Chun-Nan Chou]

Chih-Pin Tai

Dept. of Information Management
National Taiwan University

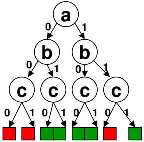
May 13, 2009

Outline

- Fundamental concepts
- Algorithms for satisfiability problems
- Decision heuristics
- Restart
- SAT competitions
- A satisfiability example using MiniSat

Boolean Satisfiability (SAT)

- Given a Boolean formula (propositional logic formula), find a variable assignment such that the formula evaluates to 1, or prove that no such assignment exists.
  - $F = (a \vee b) \wedge (\bar{a} \vee \bar{b} \vee c)$
- For n variables, there are 2^n possible truth assignments to be checked.



- First established NP-Complete problem.
  - S. A. Cook, The complexity of theorem proving procedures, *Proceedings, Third Annual ACM Symp. on the Theory of Computing, 1971*.

Boolean Formula

- If a is a Boolean variable, a is also a Boolean formula.
- If g and h are Boolean formulas, then so are:
  - $(g) \vee (h)$
  - $(g) \wedge (h)$
  - $\bar{g}$
- For example:
  - Variables a and b belong to $\{0,1\}$.
  - a is a Boolean formula.
  - $\bar{a}$, $a \vee b$, $a \wedge b$ are Boolean formulas.

Satisfiable and Unsatisfiable

- Given a Boolean formula F
  - Unsatisfiable: for all assignemts such that $F = 0$.
  - Satisfiable: there exits one assignment such that $F = 1$.
  - Ex1: $F = a$ is satisfiable.
  - Ex2: $F = a \wedge b \wedge (\bar{a} \vee \bar{b})$ is unsatisfiable.

Boolean Satisfiability Solvers

- Boolean SAT solvers have been very successful recent years in the verification area.
  - Cooperate with BDDs
  - Applications: equivalence checking and model checking
  - Applicable even for million-gate designs in EDA
- Most popular ones
  - MiniSat (2008 winner)
  - http://www.satcompetition.org/

Types of Boolean Satisfiability Solvers

- Conjunctive Normal Form (CNF) Based
  - A Boolean formula is represented as a CNF (i.e. Product of Sum).
  - For example:
    $(a \vee b \vee c) \wedge (\bar{a} \vee \bar{b} \vee c) \wedge (\bar{a} \vee b \vee \bar{c})$
  - To be satisfied, all the clauses should be '1'.
- Circuit-Based
  - A Boolean formula is represented as a circuit netlist.
  - The SAT algorithm is directly operated on the netlist.

CNF

- A conjunction of clauses, where a clause is a disjunction of literals.
- For example, a CNF formula: $(a \vee b \vee c) \wedge (\bar{a} \vee \bar{b} \vee c)$
  - Variable: a, b, c in this CNF formula.
  - Literals: a, b, c are literals in $(a \vee b \vee c)$.
  - Literals: $\bar{a}, \bar{b}, c$ are literals in $(\bar{a} \vee \bar{b} \vee c)$.
  - Clauses: $(a \vee b \vee c)$, $(\bar{a} \vee \bar{b} \vee c)$ in this CNF formula.

Outline

- Fundamental concepts
- Algorithms for satisfiability problems
- Decision heuristics
- Restart
- SAT competitions
- A satisfiability example using MiniSat

CNF-Based SAT Algorithms

- Davis-Putnam (DP), 1960.
  - Explicit resolution based
  - May explode in memory
- Davis-Putnam-Logemann-Loveland (DPLL), 1962.
  - Search based
  - Most successful, basis for almost all modern SAT solvers
- GRASP, 1996
  - Conflict driven learning and non-chronological backtracking
- zChaff, 2001.
  - Boolean constraint propagation (BCP) Algorithm

Davis-Putnam Algorithm

- M. Davis, H. Putnam, "A computing procedure for quantification theory", *J. of ACM, 1960*. (New York Univ.)
- Three satisfiability-preserving ($\approx$) transformations in DP:
  - Unit propagation rule
  - Pure literal rule
  - Resolutoin rule
- By repeatedly applying these rules, eventually obtain:
  - a formula containing an empty clause indicates unsatisfiability or
  - a formula with no clauses indicates satisfiability.

Unit Propagation Rule

- Suppose (a) is a unit clause, i.e. a clause contains only one literal.
  - Remove any instances of $\bar{a}$ from the formula.
  - Remove all clauses containing a.
- Example:
  - $(a) \wedge (\bar{a} \vee b \vee c) \wedge (a \vee \bar{b} \vee c) \wedge (\bar{a} \vee \bar{c} \vee d)$
    $\approx (b \vee c) \wedge (\bar{c} \vee d)$
  - $(a) \wedge (a \vee b) \approx$ *satisfiable*
  - $(a) \wedge (\bar{a}) \approx (\ )$ *unsatisfiable*

Pure Literal Rule

- If a literal appears only positively or only negatively, delete all clauses containing that literal.
- Example:
  $(\bar{a} \vee b \vee c) \wedge (\bar{a} \vee \bar{b} \vee c) \wedge (\bar{b} \vee c \vee d) \wedge (\bar{a} \vee \bar{c} \vee \bar{d})$
  $\approx (\bar{b} \vee c \vee d)$

Resolution Rule

- For a single pair of clauses, $(a \vee l_1 \vee \cdots \vee l_m)$ and $(\bar{a} \vee k_1 \vee \cdots \vee k_n)$, resolution on a forms the new clause $(l_1 \vee \cdots \vee l_m \vee k_1 \vee \cdots \vee k_n)$.
- Example:
  $(a \vee b) \wedge (\bar{a} \vee c)$
  $\approx (b \vee c)$
- If a is true, then for the formula to be true, c must be true.
- If a is false, then for the formula to be true, b must be true.
- So regardless of a, for the formula to be true, $b \vee c$ must be true.

Resolution Rule (cont.)

- Choose a propositional variable p which occurs positively in at least one clause and negatively in at least one other clause.
- Let P be the set of all clauses in which p occurs positively.
- Let N be the set of all clauses in which p occurs negatively.
- Replace the clauses in P and N with those obtained by resolving each clause in P with each clause in N.

An Example

$$(a \vee b) \wedge (a \vee \bar{b}) \wedge (\bar{a} \vee c) \wedge (c \vee d) \wedge (\bar{a} \vee \bar{c}) \wedge (d)$$

$$\Updownarrow \; \textit{Unit Propagation Rule}$$

$$(a \vee b) \wedge (a \vee \bar{b}) \wedge (\bar{a} \vee c) \wedge (\bar{a} \vee \bar{c})$$

$$\downarrow \; \textit{Resolution Rule}$$

$$(a) \wedge (\bar{a} \vee c) \wedge (\bar{a} \vee \bar{c})$$

$$\Updownarrow \; \textit{Unit Progation Rule}$$

$$(c) \wedge (\bar{c})$$

$$\downarrow \; \textit{Resolution Rule}$$

$$(\;) \; \textit{Unsatisfiable}$$

Potential memory explosion problem!

DPLL Algorithm

- M. Davis, G. Logemann and D. Loveland, "A Machine Program for Theorem-Proving", *Communications of ACM, 1962*. (New York Univ.)
- The basic framework for many modern SAT solvers.
  - Decision Making
  - Unit Clause rule
  - Implication
  - Conflict Detection
  - Backtrack

DPLL Algorithm

DPLL Pseudo Code

```
Function DPLL(Φ, A)

    A ← Unit − Propagation(Φ, A);

    if A is inconsistent then
        return UNSAT;
    if A assigns a value to every variable then
        return SAT;

    v ← a variable not assigned a value by A;

    if DPLL(Φ, A ∪ { v = false }) = SAT
        return SAT;
    else
        return DPLL(Φ, A ∪ { v = true });
```

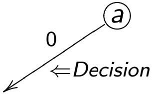
Basic DPLL Procedure - DFS

$(\bar{a} \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \bar{d})$
$(a \vee \bar{c} \vee d)$
$(a \vee \bar{c} \vee \bar{d})$
$(\bar{b} \vee \bar{c} \vee d)$
$(\bar{a} \vee b \vee \bar{c})$
$(\bar{a} \vee \bar{b} \vee c)$

a

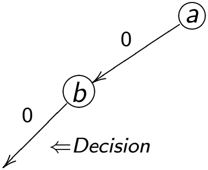
Basic DPLL Procedure - DFS

$(\bar{a} \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \bar{d})$
$(a \vee \bar{c} \vee d)$
$(a \vee \bar{c} \vee \bar{d})$
$(\bar{b} \vee \bar{c} \vee d)$
$(\bar{a} \vee b \vee \bar{c})$
$(\bar{a} \vee \bar{b} \vee c)$



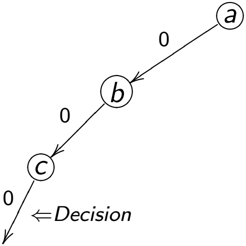
Basic DPLL Procedure - DFS

$(\bar{a} \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \bar{d})$
$(a \vee \bar{c} \vee d)$
$(a \vee \bar{c} \vee \bar{d})$
$(\bar{b} \vee \bar{c} \vee d)$
$(\bar{a} \vee b \vee \bar{c})$
$(\bar{a} \vee \bar{b} \vee c)$



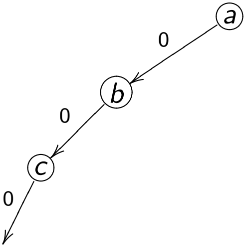
Basic DPLL Procedure - DFS

$(\bar{a} \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \bar{d})$
$(a \vee \bar{c} \vee d)$
$(a \vee \bar{c} \vee \bar{d})$
$(\bar{b} \vee \bar{c} \vee d)$
$(\bar{a} \vee b \vee \bar{c})$
$(\bar{a} \vee \bar{b} \vee c)$

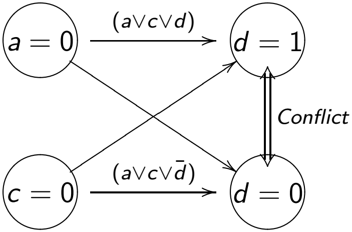


Basic DPLL Procedure - DFS

$(\bar{a} \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \bar{d})$
$(a \vee \bar{c} \vee d)$
$(a \vee \bar{c} \vee \bar{d})$
$(\bar{b} \vee \bar{c} \vee d)$
$(\bar{a} \vee b \vee \bar{c})$
$(\bar{a} \vee \bar{b} \vee c)$

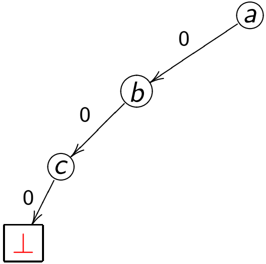


Implication Graph

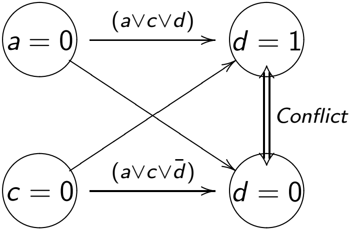


Basic DPLL Procedure - DFS

$(\bar{a} \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \bar{d})$
$(a \vee \bar{c} \vee d)$
$(a \vee \bar{c} \vee \bar{d})$
$(\bar{b} \vee \bar{c} \vee d)$
$(\bar{a} \vee b \vee \bar{c})$
$(\bar{a} \vee \bar{b} \vee c)$

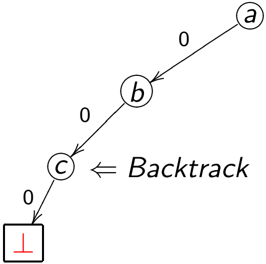


Implication Graph



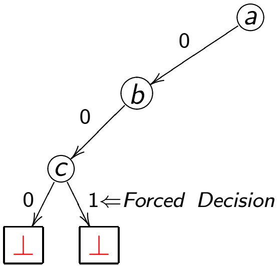
Basic DPLL Procedure - DFS

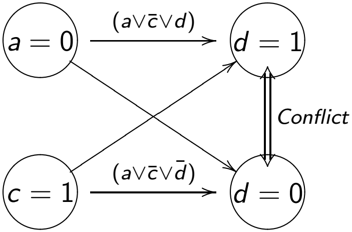
$(\bar{a} \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \bar{d})$
$(a \vee \bar{c} \vee d)$
$(a \vee \bar{c} \vee \bar{d})$
$(\bar{b} \vee \bar{c} \vee d)$
$(\bar{a} \vee b \vee \bar{c})$
$(\bar{a} \vee \bar{b} \vee c)$



Basic DPLL Procedure - DFS

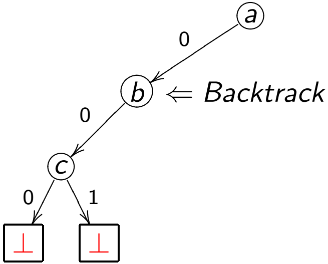
$(\bar{a} \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \bar{d})$
$(a \vee \bar{c} \vee d)$
$(a \vee \bar{c} \vee \bar{d})$
$(\bar{b} \vee \bar{c} \vee d)$
$(\bar{a} \vee b \vee \bar{c})$
$(\bar{a} \vee \bar{b} \vee c)$





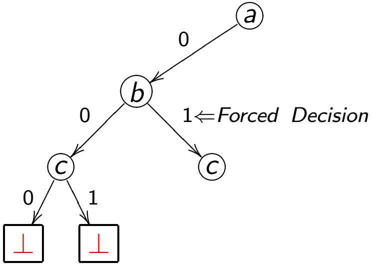
Basic DPLL Procedure - DFS

$(\bar{a} \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \bar{d})$
$(a \vee \bar{c} \vee d)$
$(a \vee \bar{c} \vee \bar{d})$
$(\bar{b} \vee \bar{c} \vee d)$
$(\bar{a} \vee b \vee \bar{c})$
$(\bar{a} \vee \bar{b} \vee c)$



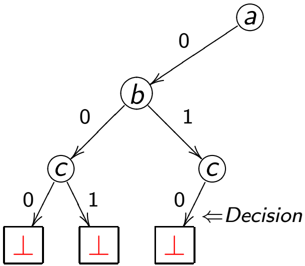
Basic DPLL Procedure - DFS

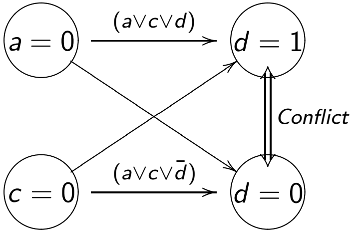
$(\bar{a} \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \bar{d})$
$(a \vee \bar{c} \vee d)$
$(a \vee \bar{c} \vee \bar{d})$
$(\bar{b} \vee \bar{c} \vee d)$
$(\bar{a} \vee b \vee \bar{c})$
$(\bar{a} \vee \bar{b} \vee c)$



Basic DPLL Procedure - DFS

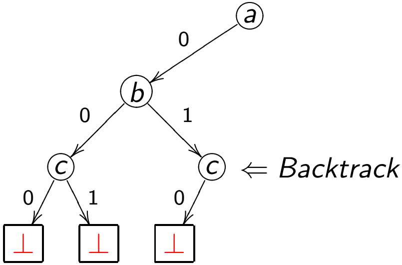
$(\bar{a} \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \bar{d})$
$(a \vee \bar{c} \vee d)$
$(a \vee \bar{c} \vee \bar{d})$
$(\bar{b} \vee \bar{c} \vee d)$
$(\bar{a} \vee b \vee \bar{c})$
$(\bar{a} \vee \bar{b} \vee c)$





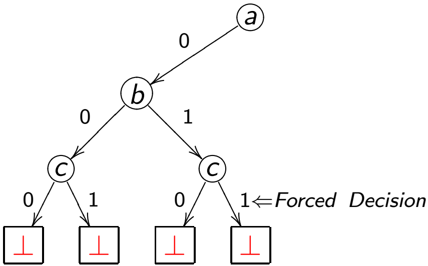
Basic DPLL Procedure - DFS

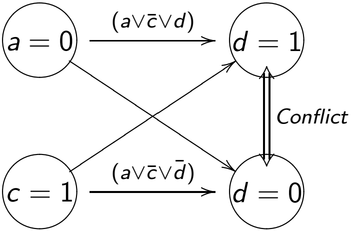
$(\bar{a} \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \bar{d})$
$(a \vee \bar{c} \vee d)$
$(a \vee \bar{c} \vee \bar{d})$
$(\bar{b} \vee \bar{c} \vee d)$
$(\bar{a} \vee b \vee \bar{c})$
$(\bar{a} \vee \bar{b} \vee c)$



Basic DPLL Procedure - DFS

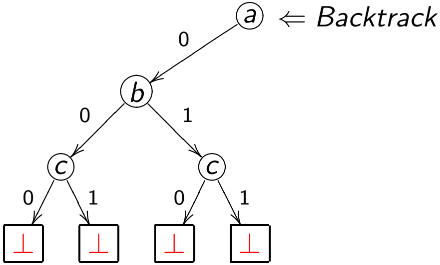
$(\bar{a} \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \bar{d})$
$(a \vee \bar{c} \vee d)$
$(a \vee \bar{c} \vee \bar{d})$
$(\bar{b} \vee \bar{c} \vee d)$
$(\bar{a} \vee b \vee \bar{c})$
$(\bar{a} \vee \bar{b} \vee c)$





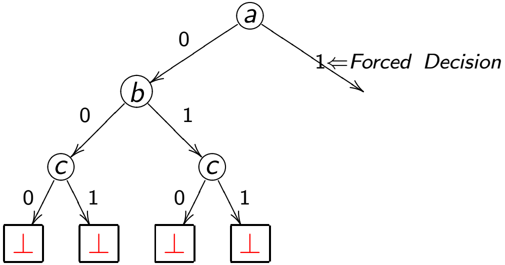
Basic DPLL Procedure - DFS

$(\bar{a} \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \bar{d})$
$(a \vee \bar{c} \vee d)$
$(a \vee \bar{c} \vee \bar{d})$
$(\bar{b} \vee \bar{c} \vee d)$
$(\bar{a} \vee b \vee \bar{c})$
$(\bar{a} \vee \bar{b} \vee c)$



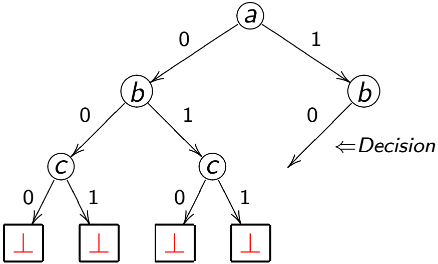
Basic DPLL Procedure - DFS

$(\bar{a} \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \bar{d})$
$(a \vee \bar{c} \vee d)$
$(a \vee \bar{c} \vee \bar{d})$
$(\bar{b} \vee \bar{c} \vee d)$
$(\bar{a} \vee b \vee \bar{c})$
$(\bar{a} \vee \bar{b} \vee c)$



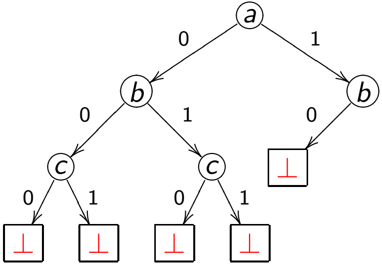
Basic DPLL Procedure - DFS

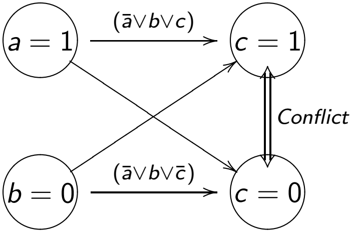
$(\bar{a} \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \bar{d})$
$(a \vee \bar{c} \vee d)$
$(a \vee \bar{c} \vee \bar{d})$
$(\bar{b} \vee \bar{c} \vee d)$
$(\bar{a} \vee b \vee \bar{c})$
$(\bar{a} \vee \bar{b} \vee c)$



Basic DPLL Procedure - DFS

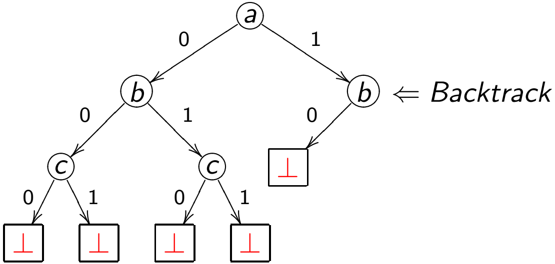
$(\bar{a} \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \bar{d})$
$(a \vee \bar{c} \vee d)$
$(a \vee \bar{c} \vee \bar{d})$
$(\bar{b} \vee \bar{c} \vee d)$
$(\bar{a} \vee b \vee \bar{c})$
$(\bar{a} \vee \bar{b} \vee c)$





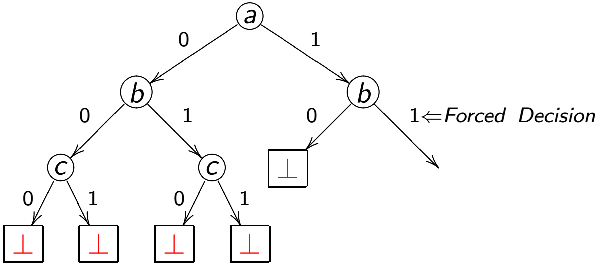
Basic DPLL Procedure - DFS

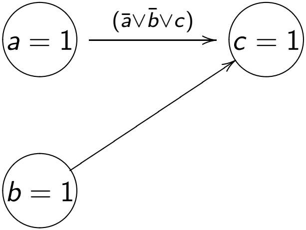
$(\bar{a} \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \bar{d})$
$(a \vee \bar{c} \vee d)$
$(a \vee \bar{c} \vee \bar{d})$
$(\bar{b} \vee \bar{c} \vee d)$
$(\bar{a} \vee b \vee \bar{c})$
$(\bar{a} \vee \bar{b} \vee c)$



Basic DPLL Procedure - DFS

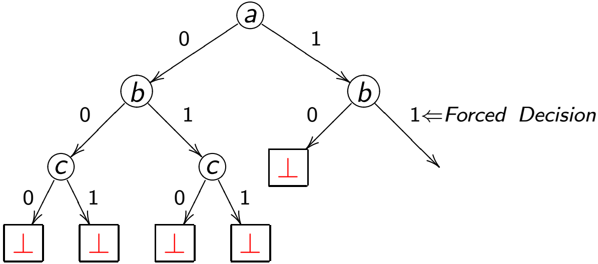
$(\bar{a} \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \bar{d})$
$(a \vee \bar{c} \vee d)$
$(a \vee \bar{c} \vee \bar{d})$
$(\bar{b} \vee \bar{c} \vee d)$
$(\bar{a} \vee b \vee \bar{c})$
$(\bar{a} \vee \bar{b} \vee c)$

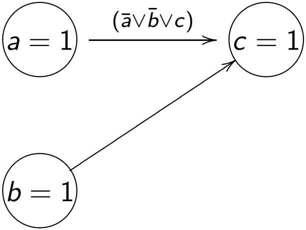




Basic DPLL Procedure - DFS

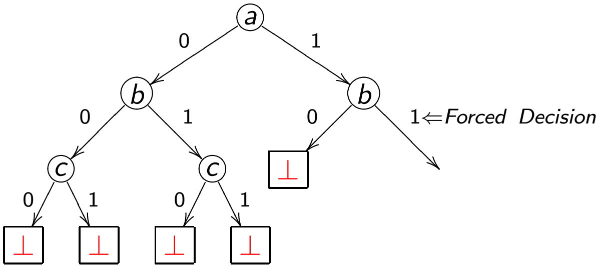
$(\bar{a} \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \bar{d})$
$(a \vee \bar{c} \vee d)$
$(a \vee \bar{c} \vee \bar{d})$
$(\bar{b} \vee \bar{c} \vee d)$
$(\bar{a} \vee b \vee \bar{c})$
$(\bar{a} \vee \bar{b} \vee c)$

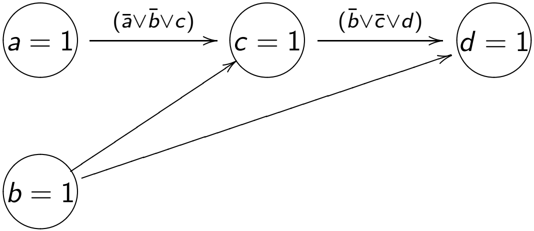




Basic DPLL Procedure - DFS

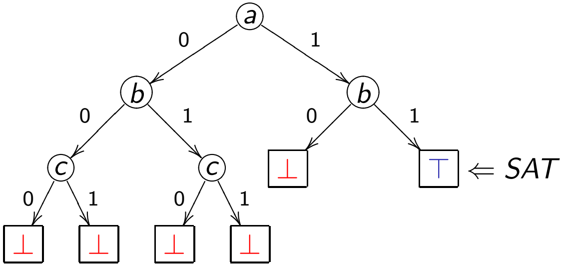
$(\bar{a} \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \bar{d})$
$(a \vee \bar{c} \vee d)$
$(a \vee \bar{c} \vee \bar{d})$
$(\bar{b} \vee \bar{c} \vee d)$
$(\bar{a} \vee b \vee \bar{c})$
$(\bar{a} \vee \bar{b} \vee c)$

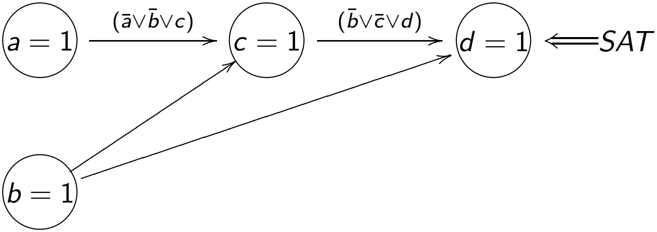




Basic DPLL Procedure - DFS

$(\bar{a} \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \bar{d})$
$(a \vee \bar{c} \vee d)$
$(a \vee \bar{c} \vee \bar{d})$
$(\bar{b} \vee \bar{c} \vee d)$
$(\bar{a} \vee b \vee \bar{c})$
$(\bar{a} \vee \bar{b} \vee c)$





Implications and Unit Clause Rule

- Implication
  - A variable is forced to be True or False based on previous assignments.
- Unit clause rule
  - A rule for elimination of one-literal clauses
  - An unsatisfied clause is a unit clause if it has exactly one unassigned literal.

$$(a \vee \bar{b} \vee c) \wedge (b \vee \bar{c}) \wedge (\bar{a} \vee \bar{c})$$

$$a = T, b = T, c \ \textit{is unassigned}$$

$$\textit{Satisfied Literal}, \ \textit{Unsatisfied Literal},$$

$$\textit{Unassigned Literal}$$

  - The unassigned literal is implied because of the unit clause.

Boolean Constraint Propagation

- Boolean Constraint Propagation (BCP)
  - Iteratively apply the unit clause rule until there is no unit clause available.
  - a.k.a. Unit Propagation
- Workhorse of DPLL based algorithms.

Features of DPLL

- Eliminate the exponential memory requirements of DP
- Exponential time is still a problem
- Limited practical applicability - largest use seen in automatic theorem proving
- Very limited size of problems are allowed
  - 32K word memory
  - Problem size limited by total size of clauses (about 1300 clauses)

GRASP

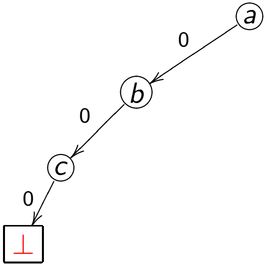
- Marques-Silva and Sakallah [SS96,SS99] (Univ. of Michigan)
  - J. P. Marques-Silva and K. A. Sakallah, "GRASP – A New Search Algorithm for Satisfiability", *Proc.ICCAD, 1996*.
  - J. P. Marques-Silva and Karem A. Sakallah, "GRASP: A Search Algorithm for Propositional Satisfiability", *IEEE Trans. Computers, 1999*.
- Incorporate conflict driven learning and non-chronological backtracking
- Practical SAT problem instances can be solved in reasonable time

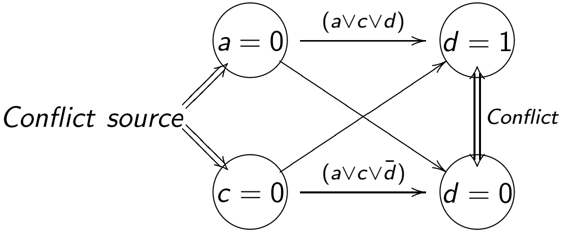
SAT Improvements

- Conflict driven learning
  - Once we encounter a conflict, figure out the cause(s) of this conflict and prevent to see this conflict again.
  - Add learned clause (conflict clause) which is the negative proposition of the conflict source.
- Non-chronological backtracking
  - After getting a learned clause from the conflict analysis, we backtrack to the "next-to-the-last" variable in the learned clause.
  - Instead of backtracking one decision at a time.

Conflict Driven Learning

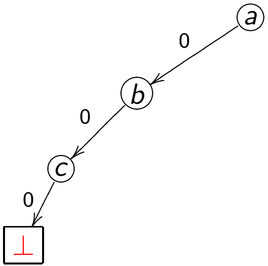
$(\bar{a} \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \bar{d})$
$(a \vee \bar{c} \vee d)$
$(a \vee \bar{c} \vee \bar{d})$
$(\bar{b} \vee \bar{c} \vee d)$
$(\bar{a} \vee b \vee \bar{c})$
$(\bar{a} \vee \bar{b} \vee c)$

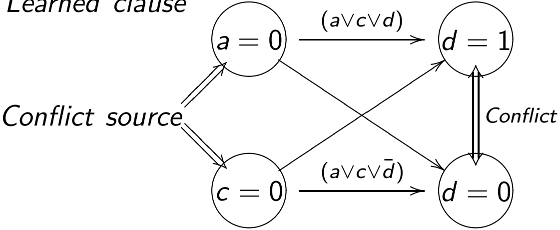




Conflict Driven Learning

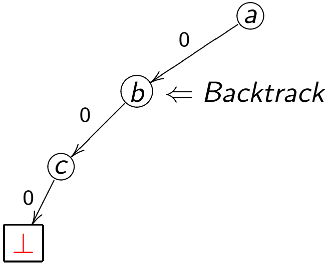
$(\bar{a} \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \bar{d})$
$(a \vee \bar{c} \vee d)$
$(a \vee \bar{c} \vee \bar{d})$
$(\bar{b} \vee \bar{c} \vee d)$
$(\bar{a} \vee b \vee \bar{c})$
$(\bar{a} \vee \bar{b} \vee c)$
$(a \vee c)$ *Learned clause*





Non-Chronological Backtracking

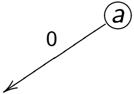
$(\bar{a} \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \bar{d})$
$(a \vee \bar{c} \vee d)$
$(a \vee \bar{c} \vee \bar{d})$
$(\bar{b} \vee \bar{c} \vee d)$
$(\bar{a} \vee b \vee \bar{c})$
$(\bar{a} \vee \bar{b} \vee c)$
$(a \vee c)$ *Learned clause*

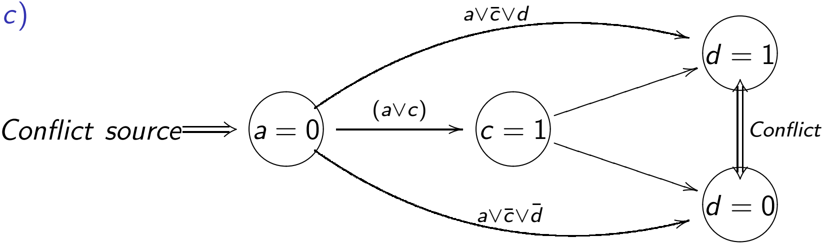


- 'a' is the next-to-the-last variable in the learned clause.
- Backtrack c=0 and b=0.

Non-Chronological Backtracking

$(\bar{a} \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \bar{d})$
$(a \vee \bar{c} \vee d)$
$(a \vee \bar{c} \vee \bar{d})$
$(\bar{b} \vee \bar{c} \vee d)$
$(\bar{a} \vee b \vee \bar{c})$
$(\bar{a} \vee \bar{b} \vee c)$
$(a \vee c)$





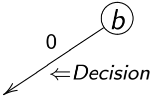
Non-Chronological Backtracking

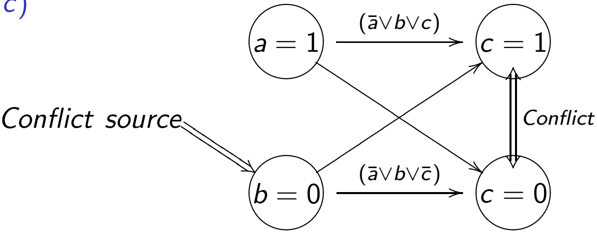
$(\bar{a} \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \bar{d})$
$(a \vee \bar{c} \vee d)$
$(a \vee \bar{c} \vee \bar{d})$
$(\bar{b} \vee \bar{c} \vee d)$
$(\bar{a} \vee b \vee \bar{c})$
$(\bar{a} \vee \bar{b} \vee c)$
$(a \vee c)$
(a) *Learned clause*

- Since there is only one variable in the learned clause, no one is the next-to-thelast variable.
- Backtrack all decisions

Non-Chronological Backtracking

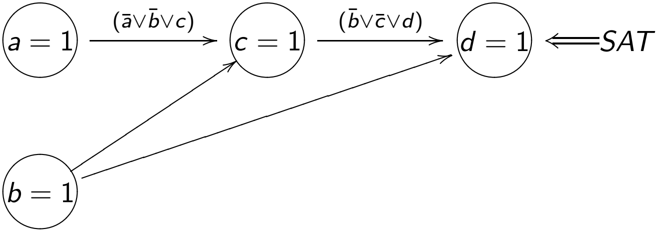
$(\bar{a} \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \bar{d})$
$(a \vee \bar{c} \vee d)$
$(a \vee \bar{c} \vee \bar{d})$
$(\bar{b} \vee \bar{c} \vee d)$
$(\bar{a} \vee b \vee \bar{c})$
$(\bar{a} \vee \bar{b} \vee c)$
$(a \vee c)$
(a)





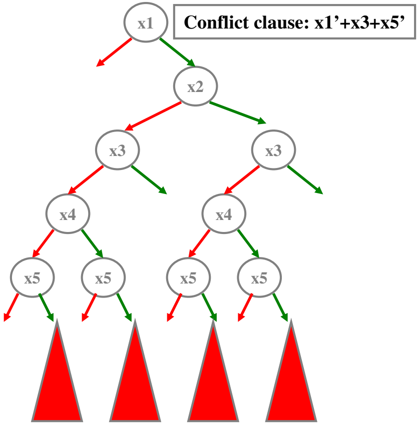
Non-Chronological Backtracking

$(\bar{a} \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \bar{d})$
$(a \vee \bar{c} \vee d)$
$(a \vee \bar{c} \vee \bar{d})$
$(\bar{b} \vee \bar{c} \vee d)$
$(\bar{a} \vee b \vee \bar{c})$
$(\bar{a} \vee \bar{b} \vee c)$
$(a \vee c)$
(a)
(b) *Learned clause*



What's the big deal?

- Significantly prune the search space because learned clause is useful forever!
- Useful in generating future conflict clauses.



Search Completeness

- With conflict driven learning, SAT search is still guaranteed to be complete.
- SAT search becomes a decision stack instead of a binary decision tree.
- When encountering a conflict, the conflict analysis does the following tasks:
  - Learned clause
  - Indicate where to backtrack
  - Learned implication

SAT Becomes Practical

- Conflict driven learning greatly increases the capacity of SAT solvers (several thousand variables) for structured problems.
- Realistic applications became plausible.
  - Usually thousands and even millions of variables
  - Typical EDA applications can make use of SAT including circuit verfication, FPGA routing and many other applications
- Research direction changes towards more efficient implementations.

zChaff

- M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, S. Malik,"Chaff: Engineering an Efficient SAT Solver" *Proc. DAC 2001*. (UC Berkeley, MIT and Princeton Univ.)
- Make the core operations fast.
  - After profiling, the most time-consuming parts are Boolean Constraint Propagation (BCP) and Decision.
- As always, good search space pruning (i.e. conflict driven learning) is important.

BCP Algorithm

- When can BCP occur ?
  - All literals in a clause but one are assigned to False.

$$\textit{The implied cases of } (v1 \vee v2 \vee v3):$$
$$(0 \vee 0 \vee v3) \textit{ or } (0 \vee v2 \vee 0) \textit{ or } (v1 \vee 0 \vee 0)$$

  - For an N-literal clause, this can only occur after $N-1$ of the literals have been assigned to False.
  - So, (theoretically) we could completely ignore the first $N-2$ assignments to this clause.
  - In reality, we pick two literals in each clause to "watch" and thus can ignore any assignments to the other literals in the clause.

BCP Algorithm

- Heuristically start with watching two unassigned literals in each clause.
- When one of the two watched literals is assigned True, this clause becomes True.
- When one of the two watched literals is assigned False, we send the clause into an Update-Watch queue to do :
  - 1.updating (another unassigned literal exists)
  - 2.BCP(only one watched literal unassigned)
  - 3.conflict (all literals are False)

BCP Algorithm

- Let's illustrate this with an example:
  - Green: watched literal
- Initially, we identify any two literals in each clause as the watched ones.
- Clauses of size one are a special case.

$$v2 \vee v3 \vee v1 \vee v4 \vee v5$$

$$v1 \vee v2 \vee \overline{v3}$$

$$v1 \vee \overline{v2}$$

$$\overline{v1} \vee v4$$

$$\overline{v1} \longleftarrow \textit{Detect unit clause}$$

BCP Algorithm

- We begin by processing the assignemt $v1 = F$ (which is implied by the size one clause)

$$v2 \lor v3 \lor v1 \lor v4 \lor v5$$
$$v1 \lor v2 \lor \overline{v3}$$
$$v1 \lor \overline{v2}$$
$$\overline{v1} \lor v4$$

$$State : (v1 = F)$$
$$Pending :$$

BCP Algorithm

- Examine each clause where the assignment being processed has set a watched literal to F.

$$
\begin{array}{ll}
 & v2 \vee v3 \vee v1 \vee v4 \vee v5 \\
\Rightarrow & v1 \vee v2 \vee \overline{v3} \\
\Rightarrow & v1 \vee \overline{v2} \\
 & \overline{v1} \vee v4
\end{array}
$$

$$State : (v1 = F)$$
$$Pending :$$

BCP Algorithm

- We need not process clauses where a watched literal has been set to T, because the clause is now satisfied and so can not become unit.

$$v2 \vee v3 \vee v1 \vee v4 \vee v5$$
$$v1 \vee v2 \vee \overline{v3}$$
$$v1 \vee \overline{v2}$$
$$\Rightarrow \quad \overline{v1} \vee v4$$

$$State : (v1 = F)$$
$$Pending :$$

BCP Algorithm

- We certainly need not process any clauses where neither watched literal changes state (in this example, where $v1$ is not watched).

$$\Rightarrow \quad v2 \lor v3 \lor v1 \lor v4 \lor v5$$
$$v1 \lor v2 \lor \overline{v3}$$
$$v1 \lor \overline{v2}$$
$$\overline{v1} \lor v4$$

$$State : (v1 = F)$$
$$Pending :$$

BCP Algorithm

- Now let's actually process the second and third clauses:

$$v2 \vee v3 \vee v1 \vee v4 \vee v5$$
$$v1 \vee v2 \vee \overline{v3}$$
$$v1 \vee \overline{v2}$$
$$\overline{v1} \vee v4$$

$$State : (v1 = F)$$
$$Pending :$$

BCP Algorithm

- For the second clause, we replace $v1$ with $\overline{v3}$ as a new watched literal because $\overline{v3}$ is not assigned to F.

$$
\begin{array}{l}
v2 \vee v3 \vee v1 \vee v4 \vee v5 \\
v1 \vee v2 \vee \overline{v3} \\
v1 \vee \overline{v2} \\
\overline{v1} \vee v4
\end{array}
\quad \Longrightarrow \quad
\begin{array}{l}
v2 \vee v3 \vee v1 \vee v4 \vee v5 \\
v1 \vee v2 \vee \overline{v3} \\
v1 \vee \overline{v2} \\
\overline{v1} \vee v4
\end{array}
$$

$State : (v1 = F)$
$Pending :$

$State : (v1 = F)$
$Pending :$

BCP Algorithm

- The third clause is unit. We record the new implication of $\overline{v2}$, and add it to the queue of assignments to process.

$$\begin{array}{l} v2 \lor v3 \lor v1 \lor v4 \lor v5 \\ v1 \lor v2 \lor \overline{v3} \\ v1 \lor \overline{v2} \\ \overline{v1} \lor v4 \\ \\ State : (v1 = F) \\ Pending : \end{array} \Longrightarrow \begin{array}{l} v2 \lor v3 \lor v1 \lor v4 \lor v5 \\ v1 \lor v2 \lor \overline{v3} \\ v1 \lor \overline{v2} \\ \overline{v1} \lor v4 \\ \\ State : (v1 = F) \\ Pending : (v2 = F) \end{array}$$

BCP Algorithm

- Next, we process $\overline{v2}$. We only examine the first two clauses.
  - For the first clause, we replace $v2$ with $v4$ as a new watched literal since $v4$ is not assigned to F.
  - The second clause is unit. We record the new implication of $\overline{v3}$, and add it to the queue of assignments to process.

$v2 \vee v3 \vee v1 \vee v4 \vee v5$		$v2 \vee v3 \vee v1 \vee v4 \vee v5$
$v1 \vee v2 \vee \overline{v3}$	$\Longrightarrow$	$v1 \vee v2 \vee \overline{v3}$
$v1 \vee \overline{v2}$		$v1 \vee \overline{v2}$
$\overline{v1} \vee v4$		$\overline{v1} \vee v4$
$State : (v1 = F,\ v2 = F)$		$State : (v1 = F,\ v2 = F)$
$Pending :$		$Pending : (v3 = F)$

BCP Algorithm

- Next, we process $\overline{v3}$. We only examine the first clause.
  - For the first clause, we replace $v3$ with $v5$ as a new watched literal since $v5$ is not assigned to F.
  - Since there are no pending assignments, and no conflict, BCP terminates and we make a decision. Both $v4$ and $v5$ are unassigned. Let's say we decide to assign $v4 = T$ and proceed.

$$v2 \vee v3 \vee v1 \vee v4 \vee v5$$
$$v1 \vee v2 \vee \overline{v3}$$
$$v1 \vee \overline{v2}$$
$$\overline{v1} \vee v4$$

$$\Longrightarrow$$

$$v2 \vee v3 \vee v1 \vee v4 \vee v5$$
$$v1 \vee v2 \vee \overline{v3}$$
$$v1 \vee \overline{v2}$$
$$\overline{v1} \vee v4$$

$State : (v1 = F,\ v2 = F,\ v3 = F)$
$Pending :$

$State : (v1 = F,\ v2 = F,\ v3 =$
$Pending :$

BCP Algorithm

- Next, we process $v4$. We do nothing at all.
  - Since there are no pending assignments, and no conflict, BCP terminates and we make a decision. Only $v5$ is unassigned. Let's say we decide to assign $v5 = F$ and proceed.

$$v2 \vee v3 \vee v1 \vee v4 \vee v5$$
$$v1 \vee v2 \vee \overline{v3}$$
$$v1 \vee \overline{v2}$$
$$\overline{v1} \vee v4$$

$$\Longrightarrow$$

$$v2 \vee v3 \vee v1 \vee v4 \vee v5$$
$$v1 \vee v2 \vee \overline{v3}$$
$$v1 \vee \overline{v2}$$
$$\overline{v1} \vee v4$$

$State : (v1 = F,\ v2 = F,\ v3 = F,\ v4 = F)$

$State : (v1 = F,\ v2 = F,\ v3 =$
$v4 = F)$

BCP Algorithm

- Next, we process $v5 = F$. We examine the first clause.
  - The first clause is already satisfied by $v4$ so we ignore it.
  - Since there are no pending assignments, and no conflict, BCP terminates and we make a decision. No variables are unassigned, so the instance is SAT, and we are done.

$$v2 \vee v3 \vee v1 \vee v4 \vee v5$$
$$v1 \vee v2 \vee \overline{v3}$$
$$v1 \vee \overline{v2}$$
$$\overline{v1} \vee v4$$

$$\Longrightarrow$$

$$v2 \vee v3 \vee v1 \vee v4 \vee v5$$
$$v1 \vee v2 \vee \overline{v3}$$
$$v1 \vee \overline{v2}$$
$$\overline{v1} \vee v4$$

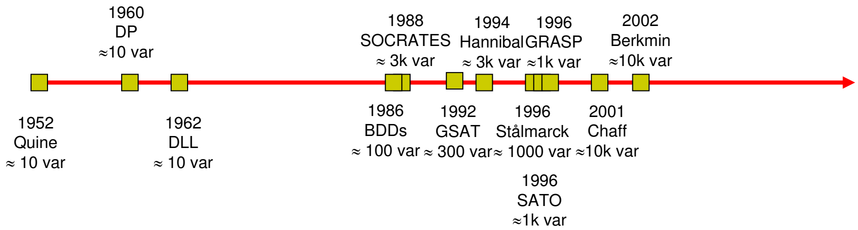
$State : (v1 = F,\ v2 = F,\ v3 = F,$
$v4 = F,\ v5 = F)$

$State : (v1 = F,\ v2 = F,\ v3 =$
$v4 = F,\ v5 = F)$

BCP Algorithm Summary

- During forward progress: Decisions and Implications
  - Only need to examine clauses where watched literal is set to F
  - Can ignore any assignments of literals to T
  - Can ignore any assignments to non-watched literals
- During backtrack: Unwind Assignment Stack
  - No action is required at all to unassign variables
  - But it is compute-intensive part in SATO
- Overall minimize clause access

The Timeline of the SAT Solver



Outline

- Fundamental concepts
- Algorithms for satisfiability problems
- Decision heuristics
- Restart
- SAT competitions
- A satisfiability example using MiniSat

Make Decision

- Beause we want to prove that the target Boolean formula is satisfiable or not, we should start with guessing the state (true or false) of a variable until the proof is done.
  - Random
  - Dynamic largest individual sum (DLIS)
  - Variable State Independent Decaying Sum (VSIDS)
  - BerkMin

RAND and DLIS

- Random
  - Simply select the next decision randomly from among the unassigned variables and its value.
- Dynamic largest individual sum (DLIS)
  - Simple and intuitive: At each decision simply choose the assignment that satisfies the most unsatisfied clauses.
  - However, considerable work is required to maintain the statistics necessary for this heuristic.
  - The total effort required for this and similar decision heuristics is much more than for the BCP algorithm in zChaff.

VSIDS

- Variable State Independent Decaying Sum (VSIDS)
  - Each variable in each polarity has a counter which is initialized to zero.
  - When a new clause is added to the database, the counter associated with each literal in this clause is incremented.
  - The (unassigned) variable and polarity with the highest counter is chosen at each decision.
  - Ties are broken randomly by default configuration.
  - Periodically, all the counters are divided by a constant.

VSIDS

- VSIDS attempts to satisfy the conflict clauses but particularly attempts to satisfy recent learned clauses.
- Difficult problems generate many conflicts (and therefore many conflict clauses), the conflict clauses dominate the problem in terms of literal count.
- Since it is independent of the variable state, it has very low overhead.
- The average rum time overhead in zChaff:
  - BCP: about 80%
  - Decision: about 10%
  - Conflict analysis: about 10%

BerkMin

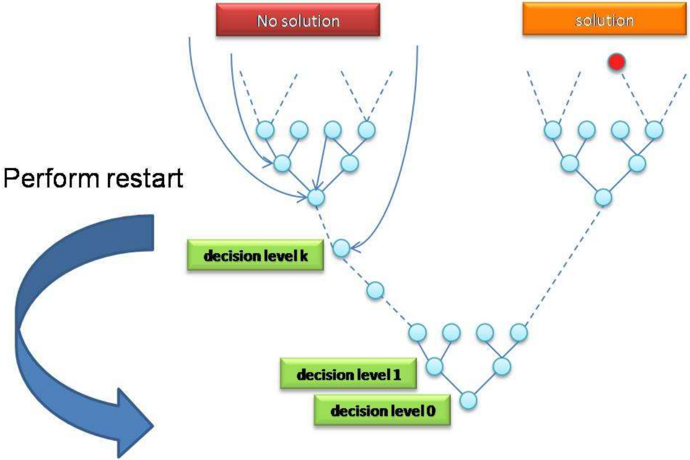
- E. Goldberg, and Y. Novikov, "BerkMin: A Fast and Robust Sat-Solver", *Proc. DATE 2002*. (Cadence Berkeley Labs and Academy of Sciences in Belarus)
- BerkMin tries to satisfy the most recent clause.
- The clause database is organized as a stack.
- The clauses of the original Boolean formula are located at the bottom of the stack and each new conflict clause is added to the top of the stack.
- The current top clause is the an unsatisfied clause which is the closet to the top of the stack.
- When making decision, choose the most active unassigned variable in the current top clause by using VSIDS.

Outline

- Fundamental concepts
- Algorithms for satisfiability problems
- Decision heuristics
- Restart
- SAT competitions
- A satisfiability example using MiniSat

Restart Motivation

- Best time to restart: when algorithm spends too much time under a wrong branch



Restart

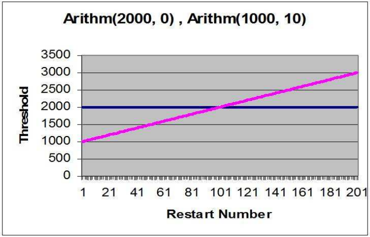
- Motivation: avoid spending too much time in “bad” branches.
  - no easy-to-find satisfying assignment
  - no opportunity for fast learning of strong clauses.
- All modern SAT solvers use a restart policy.
  - Following various criteria, the solver is forced to backtrack to level 0.
  - Abandon the current search tree and reconstruct a new one.
  - The clauses learned prior to the restart are still there after the restart and can help pruning the search space.
- Restarts have crucial impact on performance.
  - Helps reduce variance - adds to robustness in the solver.

The basic measure for restarts

- All existing techniques use the number of conflicts learned as of the previous restart.
- The difference is only in the method of calculating the threshold.

Restarts strategies

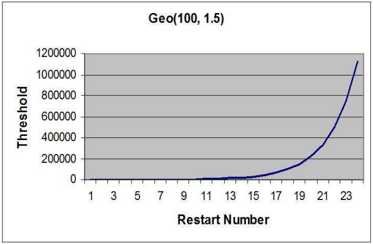
- Arithmetic (or fixed) series.
  - Parameters: x, y
  - $Init(t) = x$
  - $Next(t) = t + y$



- Used in:
  - Berkmin (550, 0)
  - Eureka (2000, 0)
  - Zchaff 2004 (700, 0)
  - Siege (16000, 0)

Restart Strategies

- Geometric series.
  - Parameters: x, y
  - $Init(t) = x$
  - $Next(t) = t * y$



- Used in
  - Minisat 2007 (100, 1.5)

Restart Strategies

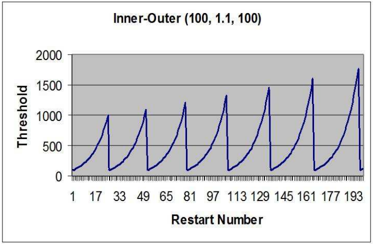
- Inner-Outer Geometric series.
  - Parameters: x, y, z
  - $Init(t) = x$
  - if $(t * y < z)$

    $Next(t) = t * y$

    else

    $Next(t) = x$

    $Next(z) = z * y$



- Used in
  - Picosat (100, 1.1, 1000)

Outline

- Fundamental concepts
- Algorithms for satisfiability problems
- Decision heuristics
- Restart
- SAT competitions
- A satisfiability example using MiniSat

SAT competitions

- The international SAT Competitions
  http://www.satcompetition.org/
- SAT Race
  http://baldur.iti.uka.de/sat-race-2008/

SAT 2007 competition

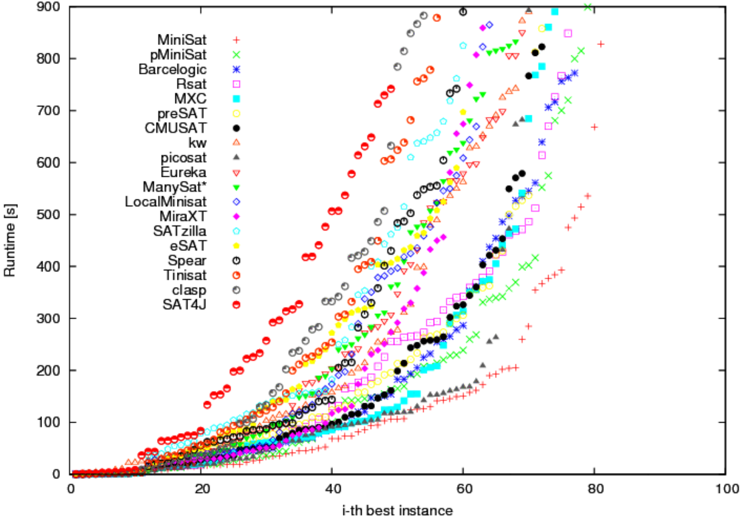
- Industrial
  - *SAT* + *UNSAT*: Rsat > Picosat > Minisat
  - *SAT*: Picosat > Rsat > Minisat
  - *UNSAT*: Rsat > Minisat > TiniSatELite
- Handmade
  - *SAT* + *UNSAT*: SATzilla CRAFTED > Minisat > MXC
  - *SAT*: March KS > SATzilla CRAFTED > Minisat
  - *UNSAT*: SATzilla > TTS > Minisat
- Random
  - *SAT* + *UNSAT*: SATzilla RANDOM > March KS > KCNFS 2004
  - *SAT*: gnovelty+ > adaptg2wsat0 > adaptg2wsat+
  - *UNSAT*: March KS > KCNFS 2004 > SATzilla RANDOM

SAT-Race 2008

- The Race
  - The Race itself will take place during or shortly before the SAT'08 conference.
  - Each solver will have to process 100 SAT instances.
  - Per SAT instance and solver a run-time limit of 15 minutes will be imposed.
- Execution Environment
  - Operating System: Scientific Linux 2.6.18, both 32-bit and 64-bit executables supported.
  - Processor(s): 2x Dual-Core Intel Xeon 5150, 2.66 GHz.
  - Memory: 8 GB (7 GB memory limit for solver processes enforced).
  - Cache: 4 MB L2 (shared).
  - Compilers: GCC 4.1.1, javac 1.5.0_11.

SAT-Race 2008

- Results: MiniSat 2.1 ¿ pMiniSat ¿ Barcelogic



Outline

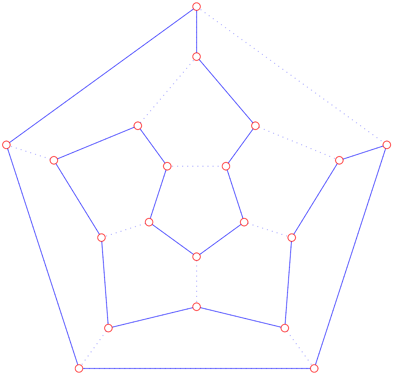
- Fundamental concepts
- Algorithms for satisfiability problems
- Decision heuristics
- Restart
- SAT competitions
- A satisfiability example using MiniSat

The usage of the MiniSat

- Use MiniSat to find a solution to $F = (x_0 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee x_2)$.

Hamiltonian Cycle

- Hamiltonian cycle, also called a Hamiltonian circuit, is a graph cycle (i.e., closed loop) through a graph that visits each node exactly once.



Encoding

- Encode the Hamiltonian cycle problem into SAT problem by the following way:
  - Assume that there is a path of length n which is the number of nodes.
  - And each Boolean variables $x_{i,j}$ represent the i_{th} node in the j_{th} position of this path.
  - So there are n^2 Boolean variables in SAT problem by this encoding method.

Add Constraint Clauses

- First constraints: Each node only exist one position of this path.
- Second constraints: Each position of this path contains only one node.
- Third constraints: Two consecutive nodes are connected by an edge.

First Constraints

- Each node only exist one position of this path
  - Each node is in the path:

$$(x_{i,0} \vee x_{i,1} \vee \cdots \vee x_{i,n-1}),\ \textit{where } 0 \leq i \leq n-1$$

  - Each node has only position (one hot):

$$\begin{aligned}
&(\overline{x_{i,0}} \vee \overline{x_{i,1}}) \wedge (\overline{x_{i,0}} \vee \overline{x_{i,2}}) \wedge \ldots \\
&(\overline{x_{i,0}} \vee \overline{x_{i,n-1}}) \wedge (\overline{x_{i,1}} \vee \overline{x_{i,2}}) \wedge \ldots \\
&(\overline{x_{i,j}} \vee \overline{x_{i,k}}) \wedge \ldots \\
&\textit{where } 0 \leq i \leq n-1,\ 0 \leq j \leq n-2,\ j+1 \leq k \leq n+1
\end{aligned}$$

Second Constraints

- Each position of this path contains only one node
  - Each position contains nodes:

$$(x_{0,i} \vee x_{1,i} \vee \cdots \vee x_{n-1,i}), \ \textit{where } 0 \leq i \leq n-1$$

  - Each position contains only one node (one hot):

$$\begin{aligned}
&(\overline{x_{0,i}} \vee \overline{x_{1,i}}) \wedge (\overline{x_{0,i}} \vee \overline{x_{2,i}}) \wedge \ldots \\
&(\overline{x_{0,i}} \vee \overline{x_{n-1,i}}) \wedge (\overline{x_{1,i}} \vee \overline{x_{2,i}}) \wedge \ldots \\
&(\overline{x_{j,i}} \vee \overline{x_{k,i}}) \wedge \ldots \\
&\textit{where } 0 \leq i \leq n-1, \ 0 \leq j \leq n-2, \ j+1 \leq k \leq n+1
\end{aligned}$$

Third Constraints

- Two consecutive nodes are connected by an edge
  - There is an edge between the i_{th} node and the j_{th} node:

$$Don't\ add\ constraint\ clauses\ into\ solver.$$

  - There are no edge between the i_{th} node and the j_{th} node:

$$(\overline{x_{i,0}} \vee \overline{x_{j,1}}) \wedge (\overline{x_{i,1}} \vee \overline{x_{j,2}}) \wedge \ldots$$
$$(\overline{x_{i,n-2}} \vee \overline{x_{j,n-1}})$$
$$where\ 0 \leq i \leq n-1,\ 0 \leq j \leq n-1, and\ i \neq j$$