Satisfiability Solving and Tools

[original created by Chun-Nan Chou and Ko-Lung Yuan]

Chiao Hsieh

Graduate Institute of Electronics Engineering
National Taiwan University

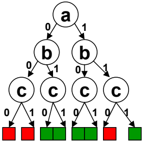
Spring 2015

Outline

Outline

Boolean Satisfiability Problem(SAT Problem)

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



- First proofed 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

- There are many ways for representing Boolean function like truth table, Boolean formula, BDD...etc.
- We use Boolean formula when solve SAT problems.
- Boolean variable
  - Boolean variable has two possible value: 0 and 1.
  - If a is a Boolean variable, a is also a Boolean formula.
- Boolean formula is constructed by several Boolean formulae with logic connective symbol $\vee$, $\wedge$, and negation. If g and h are Boolean formulae, then so are:
  - $(g \vee h)$
  - $(g \wedge h)$
  - $\bar{g}$

Satisfiable and Unsatisfiable

- Given a Boolean formula F
  - Unsatisfiable (UNSAT): All assignments let $F = 0$.
  - Satisfiable (SAT): there exits one assignment such that $F = 1$.
  - Ex1: $F = a$ is satisfiable when $a = 1$.
  - 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
- Popular SAT Solvers
  - MiniSat (2008 winner, the most popular one)
  - CryptoMiniSat (2011 winner)

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 (Conjunction Normal Form)

- Literal is a variable or its negation.
- CNF formula is 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: $(a \vee b \vee c)$, $(\bar{a} \vee \bar{b} \vee c)$ are clauses in this CNF formula.

Outline

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.
  - Efficient Boolean constraint propagation (BCP) algorithm (two watched literals)

Outline

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
  - Resolution rule
- By repeatedly applying these rules, eventually obtain:
  - a formula containing an empty clause indicates unsatisfiability
  - a formula with no clauses indicates satisfiability.
  - No rule can be used and no empty clause existing 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, $(a \lor l_1 \lor \cdots \lor l_m)$ and $(\bar{a} \lor k_1 \lor \cdots \lor k_n)$, resolution on a forms the new clause $(l_1 \lor \cdots \lor l_m \lor k_1 \lor \cdots \lor k_n)$.
- Example:
  $(a \lor b) \land (\bar{a} \lor c) \approx (b \lor c)$
  - 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`, $b \lor c$ 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.

Example 1

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

$\Updownarrow$ *Unit Propagation Rule*

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

Resolution Rule

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

$\Updownarrow$ *Unit Propagation Rule*

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

Resolution Rule

$(\ )$ *Unsatisfiable*

Potential memory explosion problem because of resolution rule

Example 2

- Solve $(a \vee b) \wedge (a \vee \bar{b}) \wedge (\bar{a} \vee c) \wedge (\bar{a} \vee \bar{c})$
- Wrong resolution:
  $(a \vee b) \wedge (a \vee \bar{b}) \wedge (\bar{a} \vee c) \wedge (\bar{a} \vee \bar{c})$ Use resolution rule
  $\approx (b \vee c) \wedge (\bar{b} \vee \bar{c})$ Use resolution rule
  $\approx (c \vee \bar{c})$ No rule can be used and no clause is empty!
  $\approx$ SAT $\rightarrow$ Wrong result!
- We have to resolve each clause in P with each clause in N.
- Correct resolution:
  - Choose a to do resolution
  - $P = \{(a \vee b), (a \vee \bar{b})\}$
  - $N = \{(\bar{a} \vee c), (\bar{a} \vee \bar{c})\}$
  - $R = \{(b \vee c), (b \vee \bar{c}), (\bar{b} \vee c), (\bar{b} \vee \bar{c})\}$
  - $(a \vee b) \wedge (a \vee \bar{b}) \wedge (\bar{a} \vee c) \wedge (\bar{a} \vee \bar{c})$
    $\approx (b \vee c) \wedge (b \vee \bar{c}) \wedge (\bar{b} \vee c) \wedge (\bar{b} \vee \bar{c})$ Replace P, N with R!
    $\approx$ ...

Outline

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.
- Main strategy
  - Decision Making
  - Unit Clause Rule
  - Implication
  - Conflict Detection
  - Backtracking

DPLL Algorithm

DPLL Pseudo Code

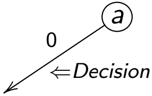
```
Function DPLL(Φ, A)

    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

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

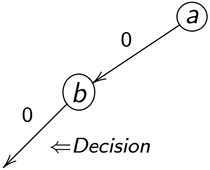
Basic DPLL Procedure - DFS

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



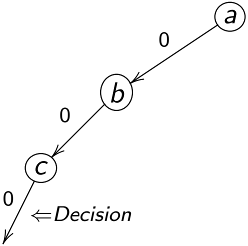
Basic DPLL Procedure - DFS

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



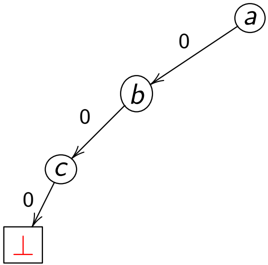
Basic DPLL Procedure - DFS

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

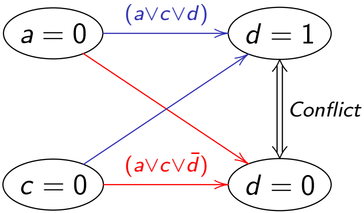


Basic DPLL Procedure - DFS

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

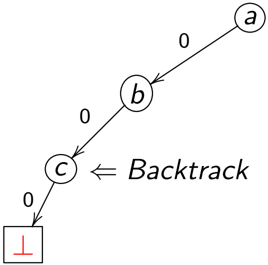


Implication Graph



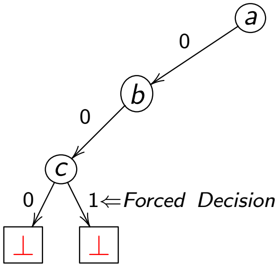
Basic DPLL Procedure - DFS

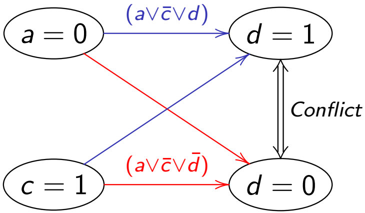
$(\bar{a} \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \bar{d})$
$(a \vee \bar{c} \vee d)$
$(a \vee \bar{c} \vee \bar{d})$
$(\bar{b} \vee \bar{c} \vee d)$
$(\bar{a} \vee b \vee \bar{c})$
$(\bar{a} \vee \bar{b} \vee c)$



Basic DPLL Procedure - DFS

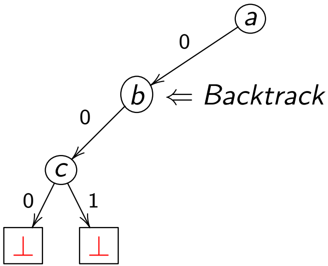
$(\bar{a} \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \bar{d})$
$(a \vee \bar{c} \vee d)$
$(a \vee \bar{c} \vee \bar{d})$
$(\bar{b} \vee \bar{c} \vee d)$
$(\bar{a} \vee b \vee \bar{c})$
$(\bar{a} \vee \bar{b} \vee c)$





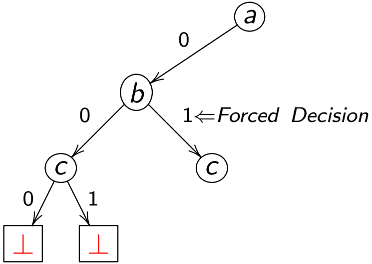
Basic DPLL Procedure - DFS

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



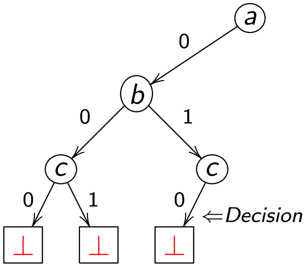
Basic DPLL Procedure - DFS

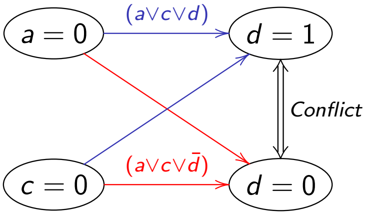
$(\bar{a} \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \bar{d})$
$(a \vee \bar{c} \vee d)$
$(a \vee \bar{c} \vee \bar{d})$
$(\bar{b} \vee \bar{c} \vee d)$
$(\bar{a} \vee b \vee \bar{c})$
$(\bar{a} \vee \bar{b} \vee c)$



Basic DPLL Procedure - DFS

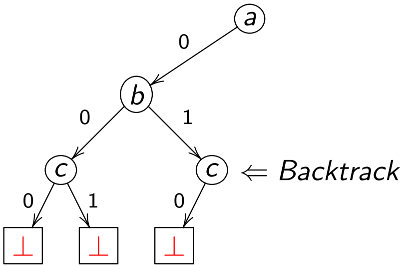
$(\bar{a} \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \bar{d})$
$(a \vee \bar{c} \vee d)$
$(a \vee \bar{c} \vee \bar{d})$
$(\bar{b} \vee \bar{c} \vee d)$
$(\bar{a} \vee b \vee \bar{c})$
$(\bar{a} \vee \bar{b} \vee c)$





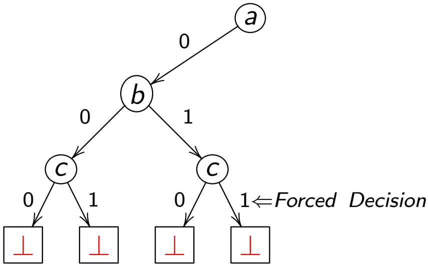
Basic DPLL Procedure - DFS

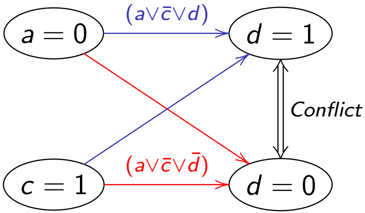
$(\bar{a} \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \bar{d})$
$(a \vee \bar{c} \vee d)$
$(a \vee \bar{c} \vee \bar{d})$
$(\bar{b} \vee \bar{c} \vee d)$
$(\bar{a} \vee b \vee \bar{c})$
$(\bar{a} \vee \bar{b} \vee c)$



Basic DPLL Procedure - DFS

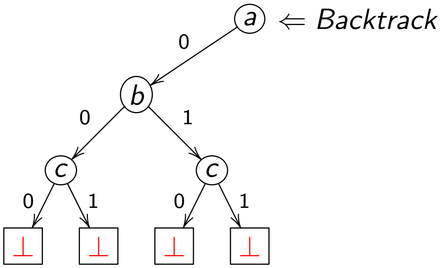
$(\bar{a} \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \bar{d})$
$(a \vee \bar{c} \vee d)$
$(a \vee \bar{c} \vee \bar{d})$
$(\bar{b} \vee \bar{c} \vee d)$
$(\bar{a} \vee b \vee \bar{c})$
$(\bar{a} \vee \bar{b} \vee c)$





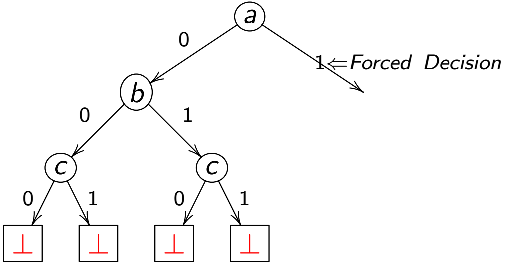
Basic DPLL Procedure - DFS

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



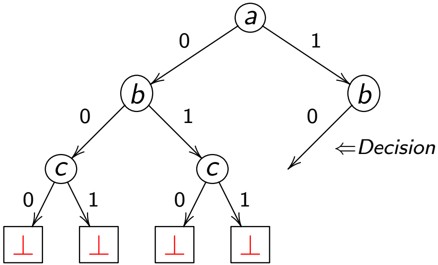
Basic DPLL Procedure - DFS

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



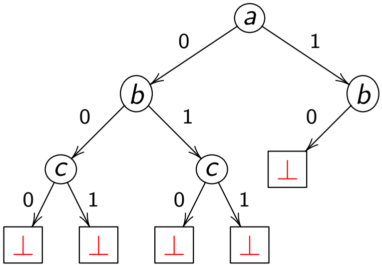
Basic DPLL Procedure - DFS

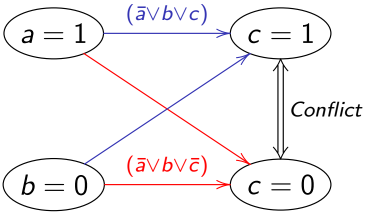
$(\bar{a} \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \bar{d})$
$(a \vee \bar{c} \vee d)$
$(a \vee \bar{c} \vee \bar{d})$
$(\bar{b} \vee \bar{c} \vee d)$
$(\bar{a} \vee b \vee \bar{c})$
$(\bar{a} \vee \bar{b} \vee c)$



Basic DPLL Procedure - DFS

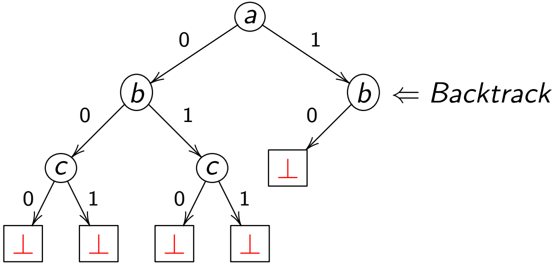
$(\bar{a} \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \bar{d})$
$(a \vee \bar{c} \vee d)$
$(a \vee \bar{c} \vee \bar{d})$
$(\bar{b} \vee \bar{c} \vee d)$
$(\bar{a} \vee b \vee \bar{c})$
$(\bar{a} \vee \bar{b} \vee c)$





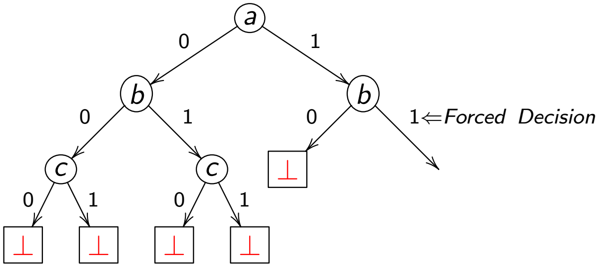
Basic DPLL Procedure - DFS

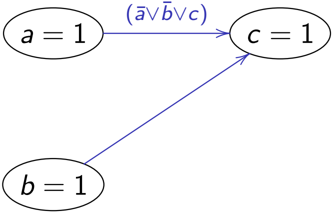
$(\bar{a} \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \bar{d})$
$(a \vee \bar{c} \vee d)$
$(a \vee \bar{c} \vee \bar{d})$
$(\bar{b} \vee \bar{c} \vee d)$
$(\bar{a} \vee b \vee \bar{c})$
$(\bar{a} \vee \bar{b} \vee c)$



Basic DPLL Procedure - DFS

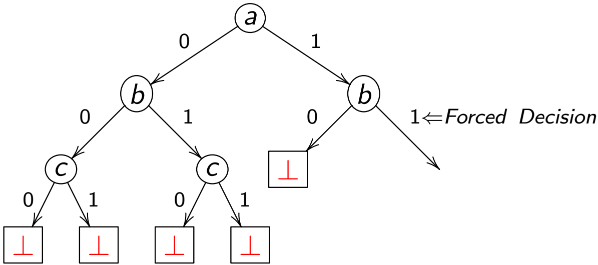
$(\bar{a} \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \bar{d})$
$(a \vee \bar{c} \vee d)$
$(a \vee \bar{c} \vee \bar{d})$
$(\bar{b} \vee \bar{c} \vee d)$
$(\bar{a} \vee b \vee \bar{c})$
$(\bar{a} \vee \bar{b} \vee c)$

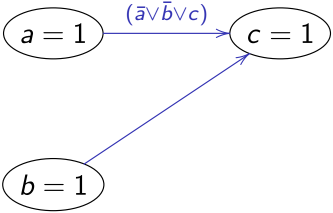




Basic DPLL Procedure - DFS

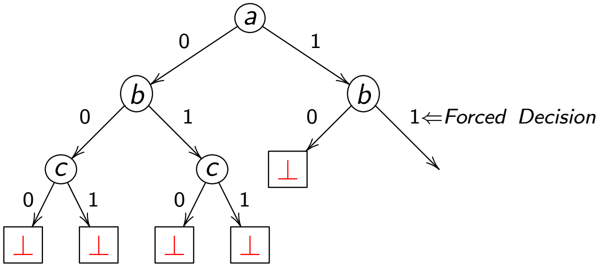
$(\bar{a} \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \bar{d})$
$(a \vee \bar{c} \vee d)$
$(a \vee \bar{c} \vee \bar{d})$
$(\bar{b} \vee \bar{c} \vee d)$
$(\bar{a} \vee b \vee \bar{c})$
$(\bar{a} \vee \bar{b} \vee c)$

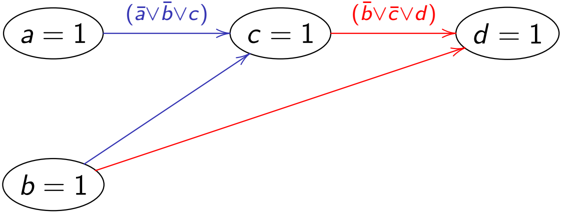




Basic DPLL Procedure - DFS

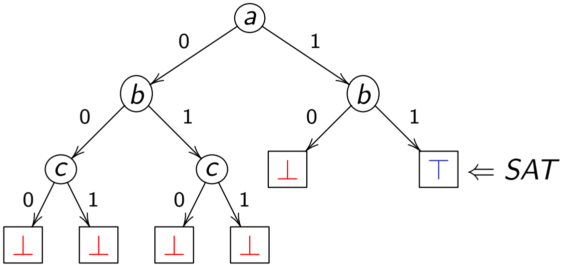
$(\bar{a} \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \bar{d})$
$(a \vee \bar{c} \vee d)$
$(a \vee \bar{c} \vee \bar{d})$
$(\bar{b} \vee \bar{c} \vee d)$
$(\bar{a} \vee b \vee \bar{c})$
$(\bar{a} \vee \bar{b} \vee c)$

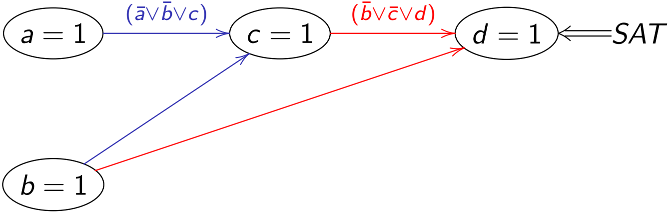




Basic DPLL Procedure - DFS

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





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.
  - The only unassigned literal, e.g. $\bar{c}$, is implied.

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

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

Satisfied Literal, *Unsatisfied Literal*,

Unassigned Literal

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)

Outline

GRASP

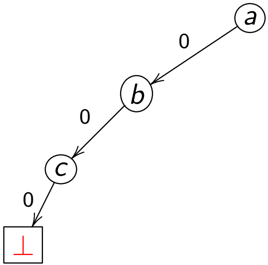
- Marques-Silva and Sakallah [SS96,SS99] (Univ. of Michigan)
  - J. P. Marques-Silva and K. A. Sakallah, "GRASP – A New Search Algorithm for Satisfiability", *Proc.ICCAD, 1996*.
  - J. P. Marques-Silva and Karem A. Sakallah, "GRASP: A Search Algorithm for Propositional Satisfiability", *IEEE Trans. Computers,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

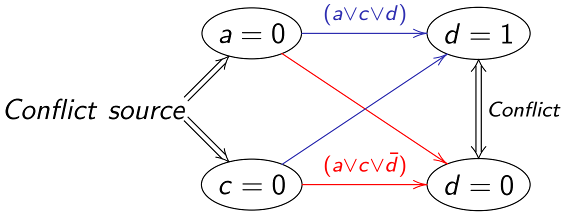
SAT Improvements

- 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

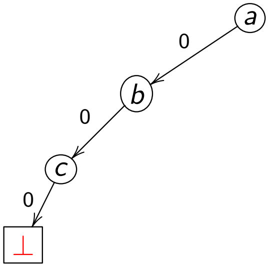
$(\bar{a} \lor b \lor c)$
$(a \lor c \lor d)$
$(a \lor c \lor \bar{d})$
$(a \lor \bar{c} \lor d)$
$(a \lor \bar{c} \lor \bar{d})$
$(\bar{b} \lor \bar{c} \lor d)$
$(\bar{a} \lor b \lor \bar{c})$
$(\bar{a} \lor \bar{b} \lor c)$

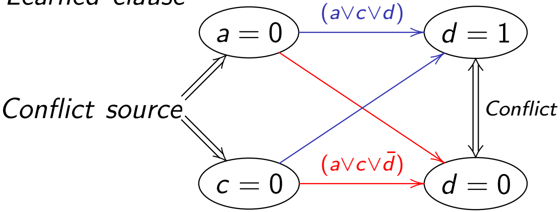




Conflict Driven Learning

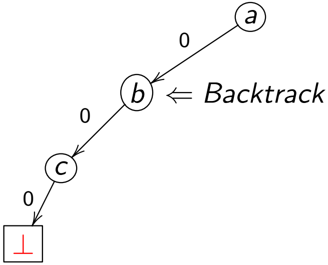
$(\bar{a} \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \bar{d})$
$(a \vee \bar{c} \vee d)$
$(a \vee \bar{c} \vee \bar{d})$
$(\bar{b} \vee \bar{c} \vee d)$
$(\bar{a} \vee b \vee \bar{c})$
$(\bar{a} \vee \bar{b} \vee c)$
$(a \vee c)$ *Learned clause*





Non-Chronological Backtracking

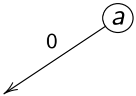
$(\bar{a} \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \bar{d})$
$(a \vee \bar{c} \vee d)$
$(a \vee \bar{c} \vee \bar{d})$
$(\bar{b} \vee \bar{c} \vee d)$
$(\bar{a} \vee b \vee \bar{c})$
$(\bar{a} \vee \bar{b} \vee c)$
$(a \vee c)$ *Learned clause*

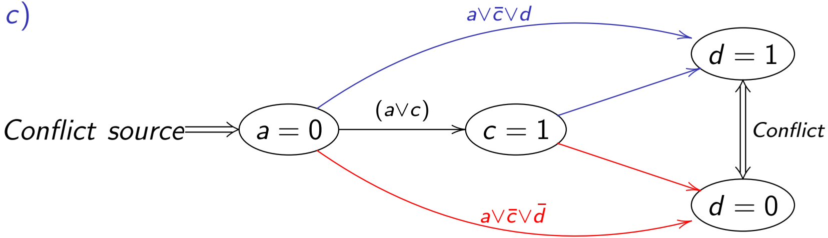


- 'a' is the next-to-the-last variable in the (current) learned clause.
  - c is the last (assigned) variable in this learned clause so a is called the next-to-the-last variable
  - Because of this learned clause, when a is assigned 0 then c will be implied and we don't have to make decision for c
- After doing non-chronological backtracking, we will not forgive the path $a = 0, b = 0 \ldots$ if needed.

Non-Chronological Backtracking

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





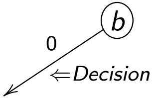
Non-Chronological Backtracking

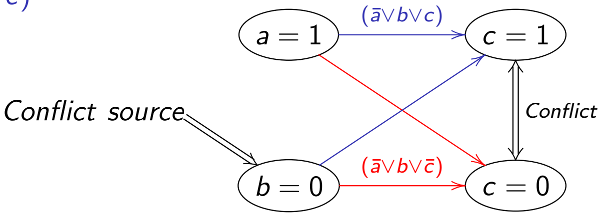
$(\bar{a} \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \bar{d})$
$(a \vee \bar{c} \vee d)$
$(a \vee \bar{c} \vee \bar{d})$
$(\bar{b} \vee \bar{c} \vee d)$
$(\bar{a} \vee b \vee \bar{c})$
$(\bar{a} \vee \bar{b} \vee c)$
$(a \vee c)$
(a) *Learned clause*

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

Non-Chronological Backtracking

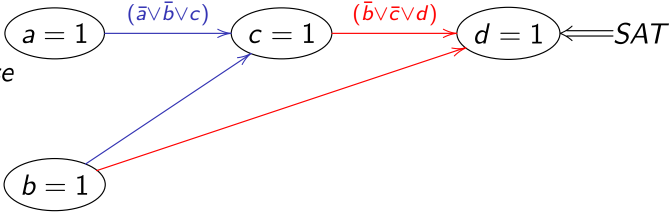
$(\bar{a} \lor b \lor c)$
$(a \lor c \lor d)$
$(a \lor c \lor \bar{d})$
$(a \lor \bar{c} \lor d)$
$(a \lor \bar{c} \lor \bar{d})$
$(\bar{b} \lor \bar{c} \lor d)$
$(\bar{a} \lor b \lor \bar{c})$
$(\bar{a} \lor \bar{b} \lor c)$
$(a \lor c)$
(a)





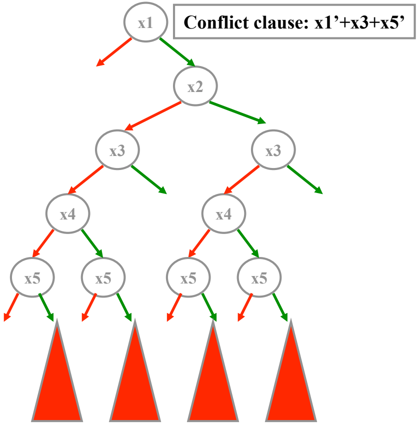
Non-Chronological Backtracking

$(\bar{a} \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \bar{d})$
$(a \vee \bar{c} \vee d)$
$(a \vee \bar{c} \vee \bar{d})$
$(\bar{b} \vee \bar{c} \vee d)$
$(\bar{a} \vee b \vee \bar{c})$
$(\bar{a} \vee \bar{b} \vee c)$
$(a \vee c)$
(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 verification, FPGA routing and many other applications
- Research direction changes towards more efficient implementations.

Outline

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, pruning search space (i.e. conflict driven learning) is important.

BCP Algorithm

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

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

  - For an N-literal clause, this can only occur after $N-1$ literals have been assigned to `False`.
  - So, (theoretically) we could completely ignore the first $N-2$ assignments to this clause.
  - Two watched Literals:
    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 one of the followings:
  - 1. Updating (there exists another unassigned literal)
  - 2. BCP (only one watched literal unassigned)
  - 3. Conflict handling (all literals are `False`)

BCP Algorithm

- Initially, pick any two literals in each clause as the watched literals.
  - Green: watched literals
- Clauses with only one literal are detected at the mean time.

$$v2 \vee v3 \vee v1 \vee v4 \vee v5$$
$$v1 \vee v2 \vee \overline{v3}$$
$$v1 \vee \overline{v2}$$
$$\overline{v1} \vee v4$$
$$\overline{v1} \longleftarrow \text{Detect unit clause}$$

BCP Algorithm

- We begin by processing the assignment $v1 = F$
  - Implied by the unit clause $\overline{v1}$

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

$$State : v1 = F$$

$$Pending :$$

BCP Algorithm

- Need not process clauses where watched literals are set to `True`.
  - Because those clauses are now satisfied.

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

$$State : v1 = F$$

$$Pending :$$

BCP Algorithm

- Need not process clauses where neither watched literal is assigned.
  - Because those clause are definitely not a unit clause.

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

$$State : v1 = F$$

$$Pending :$$

BCP Algorithm

- Only examine clauses where a watched literal is set to `False` due to the assignment.

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

$$State : v1 = F$$

$$Pending :$$

BCP Algorithm

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

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

$State : v1 = F$ $\qquad$ $State : v1 = F$

$Pending :$ $\qquad$ $Pending :$

BCP Algorithm

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

$$\Rightarrow \begin{array}{l} v2 \vee v3 \vee v1 \vee v4 \vee v5 \\ v1 \vee v2 \vee \overline{v3} \\ v1 \vee \overline{v2} \\ \overline{v1} \vee v4 \end{array} \qquad \begin{array}{l} v2 \vee v3 \vee v1 \vee v4 \vee v5 \\ v1 \vee v2 \vee \overline{v3} \\ v1 \vee \overline{v2} \\ \overline{v1} \vee v4 \end{array}$$

$$State : v1 = F \qquad State : v1 = F$$

$$Pending : \qquad \Longrightarrow \qquad Pending : (v2 = F)$$

BCP Algorithm

- Next, for $\overline{v2}$, only the first two clauses are examined.
  - For the first clause, replace $v2$ with $v4$ as a new watched literal.

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

$$State : v1 = F, v2 = F \qquad State : v1 = F, v2 = F$$

$$Pending : \quad \Longrightarrow \quad Pending : (v3 = F)$$

BCP Algorithm

- Next, for $\overline{v3}$, only the first clause is examined.
  - For the first clause, replace $v3$ with $v5$ as a new watched literal.
  - 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 assign $v4 = \texttt{True}$ and proceed.

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

$$State : v1 = F, v2 = F, \quad v3 = F \qquad\qquad State : v1 = F, v2 = F, \quad v3 = F$$

$$Pending : \qquad\qquad Pending :$$

BCP Algorithm

- Next, for $v4$, all clauses are satisfied.
- Depend on implementation, it may continue and assign value to $v5$.
- The instance is SAT, and we are done.

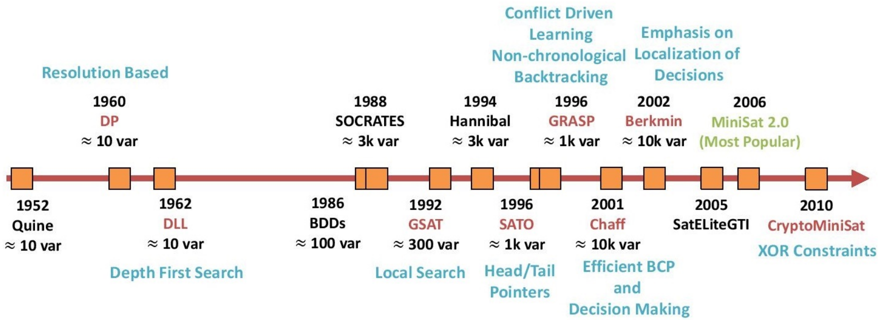
$$v2 \vee v3 \vee v1 \vee v4 \vee v5$$
$$v1 \vee v2 \vee \overline{v3}$$
$$v1 \vee \overline{v2}$$
$$\overline{v1} \vee v4$$

$$State : v1 = F, v2 = F,$$
$$v3 = F, v4 = T$$
$$Pending :$$

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 of non-watched literals
- During backtrack: Unwind Assignment Stack
  - No action is required at all to unassigned variables
  - But it is computation-intensive part in SATO (*SATO: an Efficient Propositional Prover. Hantao Zhang*. Department of Computer Science. The University of Iowa. Iowa City, IA 52242-1419, USA*)
- Overall minimize clause access

The Timeline of the SAT Solver



Outline

Outline

1 Fundamental Concepts

2 Core algorithms of satisfiability problems

3 Heuristics
- Decision heuristics
- Restart mechanism

4 SAT competitions

5 Applications

Make Decision

- Because 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.
- Some strategy:
  - Random
  - Dynamic Largest Individual Sum (DLIS)
  - Variable State Independent Decaying Sum (VSIDS)

RAND and DLIS

- Random
  - Simply select an unassigned variable and a value randomly for the next decision.
- Dynamic Largest Individual Sum (DLIS)
  - At each decision simply choose the assignment that satisfies the most unsatisfied clauses.
  - Simple and intuitive.
  - However, considerable work is required to maintain the statistics.
  - The total effort required is much more than the effort 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 (cont.)

- 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%

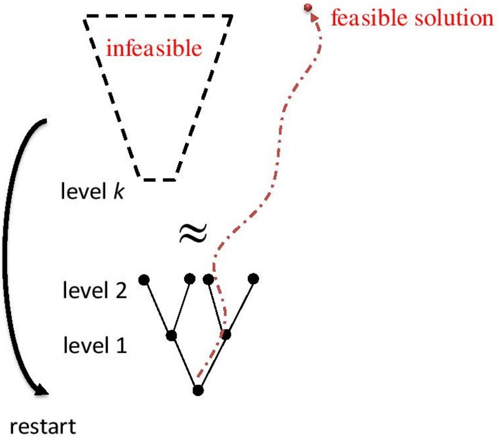
BerkMin

- 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 closest to the top of the stack.
- When making decision, choose the most active unassigned variable in the current top clause by using VSIDS.

Outline

Restart Motivation

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



Restart

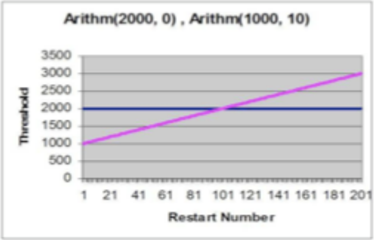
- Motivation: avoid spending too much time in “bad” branches.
  - no easy-to-find satisfying assignment
  - no opportunity for fast learning of strong clauses.
- All modern SAT solvers use a restart policy.
  - Following various criteria, the solver is forced to backtrack to level 0.
  - Abandon the 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.
  - Reduce variance - increase 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

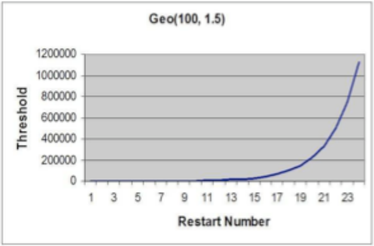
- Arithmetic (or fixed) series.
  - Parameters: x, y
  - t: threshold, when conflict number reaches the threshold, restart!
  - $Init(t) = x$
  - $Next(t) = t + y$



- Used in (solver name(x, y)):
  - Berkmin (550, 0)
  - Eureka (2000, 0)
  - zChaff 2004 (700, 0)
  - Siege (16000, 0)

Restart Strategies

- Geometric series.
  - Parameters: x, y
  - t: threshold, when conflict number reaches the threshold, restart!
  - $Init(t) = x$
  - $Next(t) = t * y$



- Used in (solver name(x, y)):
  - Minisat 2007 (100, 1.5)

Restart Strategies

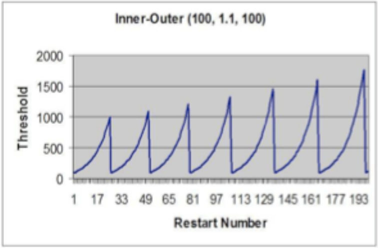
- Inner-Outer Geometric series.
  - Parameters: x, y, z
  - t: threshold, when conflict number reaches the threshold, restart!
  - $Init(t) = x$
  - if $(t * y < z)$

    $Next(t) = t * y$

    else

    $Next(t) = x$

    $Next(z) = z * y$



- Used in (solver name(x, y, z)):
  - Picosat (100, 1.1, 1000)

Other Issues

- Incremental SAT
  - Take apart the clause database.
  - Solve the first part and record the learned information.
  - If it is UNSAT, then stop.
  - If it is SAT, then add the next part to solve.
  - And so on...
- Refutation proof, i.e., proof of UNSAT (Ex.Resolution Proof)
- Parallel computation
- Memory management
- etc...

Outline

SAT competitions

- From March to June
- The international SAT Competitions (Starting from 2002)
  http://www.satcompetition.org/
  - Three main categories of benchmarks:
    Application(Industrial), Hard Combinatorial(Crafted), Random
  - Three Evaluation in each category:
    SAT, UNSAT, ALL(SAT + UNSAT)
  - Separate sequential and parallel since 2011
- SAT-Race (**2015**, 2010, 2008, 2006)
  http://baldur.iti.kit.edu/sat-race-2015/
- SAT Challenge 2012
  http://baldur.iti.kit.edu/SAT-Challenge-2012/

Famous SAT Solvers

- MiniSat, http://minisat.se/
  - Silver in 2005, Gold in 2006 and 2008
  - Well-known for its compact and simple implementation
  - Originally only 600 lines in total
    but contains most algorithms mentioned in the slide!!
  - A category since 2009 called Minisat Hack
- SATzilla, http://www.cs.ubc.ca/labs/beta/Projects/SATzilla/
  - Gold in 2007, 2009, and 2012
  - Evaluate the problem instance first
  - Select an appropriate solver to solve

Famous SAT Solvers

- ppfolio, http://www.cril.univ-artois.fr/˜roussel/ppfolio/
  - Win a total of 16 medals in 2011
  - Assign cores to the five solvers in use.
- Winners of recent years
  - glucose, http://www.labri.fr/perso/lsimon/glucose/
  - Lingeling, http://fmv.jku.at/lingeling

Outline

The Usage of the MiniSat

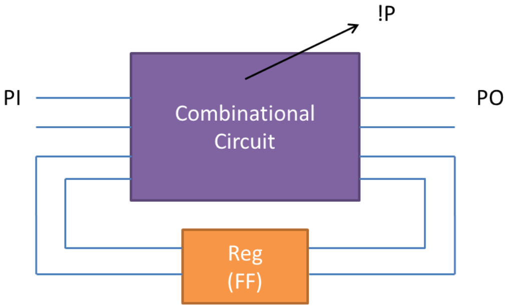
- MiniSat Page: http://minisat.se/
- The newest version: 2.2.0
- Use MiniSat to find a solution of $F = (x_0 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee x_2)$.
  - Go to MiniSat Page to download it.
  - Tar the .gz file   tar -zxvf minisat-2.2.0.tar.gz
  - Change to directory "core"   cd core
  - Modify path   export MROOT=../
  - Make and compile in directory "core"   make
  - Build DIMACS CNF file for problem you want to solve
    http://www.satcompetition.org/2009/format-benchmarks2009.html
  - Run the minisat to solve problem   ./minisat CnfFileName

DIMACS CNF Format

- It is a standard format for the input files (CNF files) of SAT solvers.
  - Use c to write comments
  - Start with p cnf VarialbeNumber ClauseNumber
  - Write the clause with integer(with/without "-") for representing the literals
  - Use "0" to mark the end of a clause
- Example: $(x_0 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee x_2)$
  c this is a simple DIMACS cnf, use 1, 2, 3 for x0, x1, x2 respectively
  p cnf 3 2
  1 2 3 0
  -2 3 0

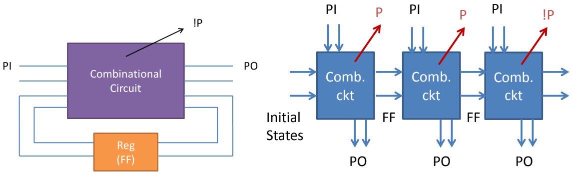
Example 1: Bounded Model Checking

- We want to check property AG(p) for a given sequential circuit. See whether it has bugs!



Timeframe Expansion Model

- Iterative timeframe expansion model: sequential SAT becomes a combinational problem.

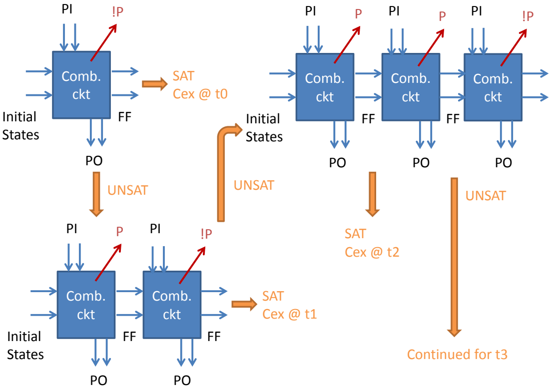


BMC Algorithm

- Let C be the set of constraints on the combinational circuit
- For an iterative model that unfolds the circuit for n times, let C_i correspond to the i-th iteration of the circuit constraint$(0 \leqq i \leqq k-1)$
- Let I_0 be the initial state value
- Let P be the property to prove
- Following is the BMC algorithm:
- BMC(P)
  - Let k=1
  - loop:
    - if (SAT($I_0 \wedge C_0 \wedge ... \wedge C_{k-1} \wedge !P_{k-1}$))
    - return Find a counter-example at time (k-1)
    - k=k+1
    - go to loop

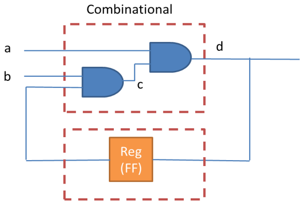
BMC Algorithm

- In other words ...



How to Write CNF for C_i

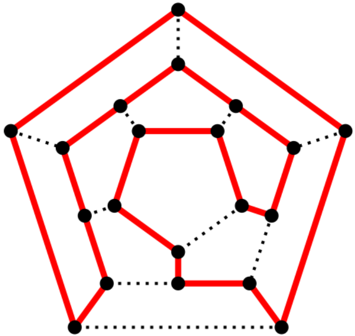
- Here is an example:



- We use a_i, b_i, c_i, d_i to represent the signals for timeframe i
- We use s_{outi} to represent FF_{out} for timeframe i
- $C_i = (c_i = b_i \wedge s_{outi}) \wedge (d_i = a_i \wedge c_i) \wedge (s_{outi} = d_{i-1})$ for $i > 0$...(1)
- $C_0 = (c_0 = b_0 \wedge I_0) \wedge (d_0 = a_0 \wedge c_0)$...(2)
- $(m = n) \equiv (m' \vee n) \wedge (n' \vee m)$...(3)
- We can use (3) to rewrite (1) and (2) for CNF

Example 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.



(Wiki: http://en.wikipedia.org/wiki/File:Hamiltonian_path.svg)

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.
  - Let each Boolean variables $x_{i,j}$ represent that i_{th} node is 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 occupies only 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 occupies only 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 holds only one 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 at least a node:

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

  - 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 is no connection between the i_{th} node and the j_{th} node:

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

Demo

- Given following graph, check if there is a Hamiltonian Cycle