Satisfiability Solving and Tools

Originally created by Chun-Nan Chou and Ko-Lung Yuan
Revised by Chiao Hsieh

Shao-Wei Chu

Graduate Institute of Electronics Engineering
National Taiwan University

Fall 2019

Outline

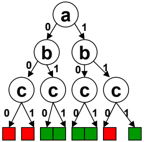
1. Fundamental Concepts
2. Core algorithms of satisfiability problems
3. Heuristics
4. SAT competitions
5. Applications

Outline

1. Fundamental Concepts
2. Core algorithms of satisfiability problems
3. Heuristics
4. SAT competitions
5. Applications

Boolean Satisfiability Problem(SAT Problem)

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



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

Bolean Reasoning

- The central idea in Boolean reasoning, first given by Boole, is to reduce a given system of logical equations, and then to carry out the desired reasoning on that equation.
- e.g. Model checking $A \models f : L(A) \cap L(B_{\neg f}) = \emptyset$
- Fundamental tradeoff
  - canonical data structure (e.g. truth table, ROBDD)
    - data structure uniquely represents function
    - decision procedure is trivial(pointer comparison, DFS)
    - size of data structure is in general exponential

    item non-canonical data structure (e.g. AIG, CNF)
    - size of data structure is in general linear
    - systematic search for for satisfying assignment
    - decision may take an exponential amount of time

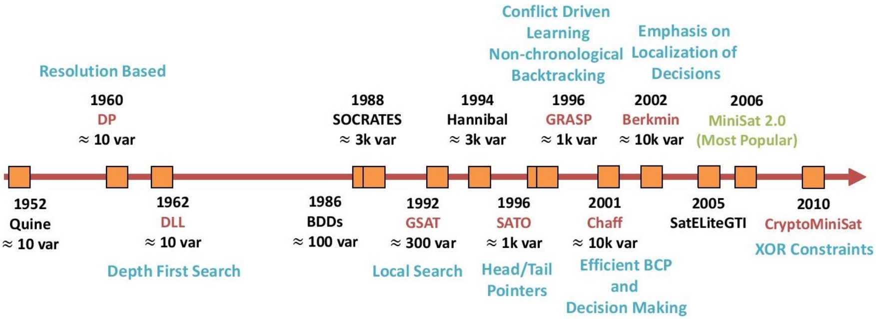
Boolean Satisfiability Solvers

- Boolean SAT solvers have been very successful recent years in the verification area, due to various nice heuristics
  - Support up to 10k variables, much more scalable than 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)
  - glucose

Conjunctive Normal Form (CNF)

- A Boolean formula is represented as a CNF (i.e. Product of Sum).
- Linear structure for lots of variables and easy to add extra constraints
- Literal is a variable or its negation.
- CNF formula is a conjunction of clauses, where a clause is a disjunction of literals.
- For example:
  $(a \vee b \vee c) \wedge (\bar{a} \vee \bar{b} \vee c)$
  - Variable: a, b, c in this CNF formula.
  - 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.

The Timeline of the SAT Solver



1952
Quine
≈ 10 var

Resolution Based
1960
DP
≈ 10 var

1962
DLL
≈ 10 var
Depth First Search

1986
BDDs
≈ 100 var

1988
SOCRATES
≈ 3k var

1992
GSAT
≈ 300 var
Local Search

1994
Hannibal
≈ 3k var

1996
SATO
≈ 1k var
Head/Tail Pointers

Conflict Driven Learning
Non-chronological Backtracking
1996
GRASP
≈ 1k var

2001
Chaff
≈ 10k var
Efficient BCP and Decision Making

Emphasis on Localization of Decisions
2002
Berkmin
≈ 10k var

2005
SatELiteGTI

2006
MiniSat 2.0
(Most Popular)

2010
CryptoMiniSat
XOR Constraints

Outline

1. Fundamental Concepts
2. Core algorithms of satisfiability problems
   - Davis-Putnam Algorithm
   - DPLL Algorithm
   - GRASP Algorithm
   - zChaff Algorithm
3. Heuristics
4. SAT competitions
5. Applications

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

1. Fundamental Concepts

2. Core algorithms of satisfiability problems
   - Davis-Putnam Algorithm
   - DPLL Algorithm
   - GRASP Algorithm
   - zChaff Algorithm

3. Heuristics

4. SAT competitions

5. Applications

Davis-Putnam Algorithm

- M.Davis, H.Putnam, "A computing procedure for quantification theory" J. of ACM, 1960
- By repeating three satisfiability-preserving rules:
  - Unit propagation rule
  - Pure literal rule
  - Resolution rule
- eventually obtain:
  - $\bot \in F$ indicates UNSAT
  - $F = \top$ (a formula with no clauses indicates SAT)

DP Algorithm

DP Pseudo Code

```
Function DP($F$, $A$)
    forever
        if $\perp \in F$ then
        return $UNSAT$;
        if $F = \top$ then
        return $SAT$;

        $A \leftarrow Unit - Propagation(F, A)$;
        $A \leftarrow Pure - Literal(F, A)$;
        $A \leftarrow Resolution(F, A)$;
```

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

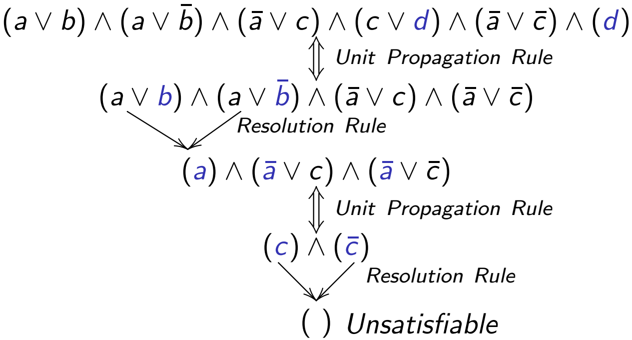
Resolution Rule

- 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.

Example 1



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

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

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

Resolution Rule

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

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

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

Resolution Rule

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

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 → 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!
    $\approx$ ...
- Potential memory explosion problem (n → $n^2/4$)

Outline

1. Fundamental Concepts

2. Core algorithms of satisfiability problems
    - Davis-Putnam Algorithm
    - DPLL Algorithm
    - GRASP Algorithm
    - zChaff Algorithm

3. Heuristics

4. SAT competitions

5. Applications

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

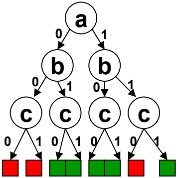
DPLL Algorithm

DPLL Pseudo Code

```
Function DPLL($F$, $A$)

    $A \leftarrow Unit - Propagation(F, A)$;
    if $A$ is inconsistent then
        return $UNSAT$;
    if $A$ assigns a value to every variable then
        return $SAT$;

    $v \leftarrow$ a variable not assigned a value by $A$;
    if DPLL($F$, $A \cup$ { $v =$ False }) $= SAT$
        return $SAT$;
    else
        return DPLL($F$, $A \cup$ { $v =$ True });
```



Boolean Constraint Propagation(a.k.a. Unit Propagation)

- Iteratively apply the unit clause rule until there is no unit clause available.
- 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.
- Workhorse of DPLL based algorithms.

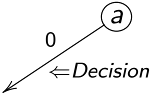
Basic DPLL Procedure - DFS

Caution: The graph on the right is drawn for the purpose of lecture. It is not seen in the algorithm implementation.

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

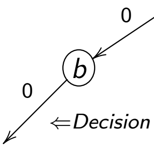
Basic DPLL Procedure - DFS

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



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

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

$(a \vee c \vee d)$

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

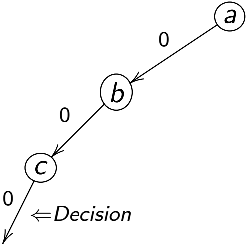
$(a \vee \bar{c} \vee d)$

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

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

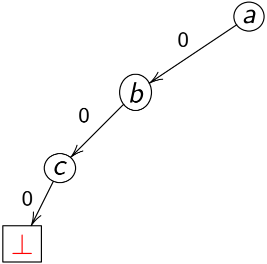
$(\bar{a} \vee b \vee \bar{c})$

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

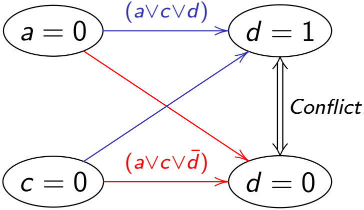


Basic DPLL Procedure - DFS

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

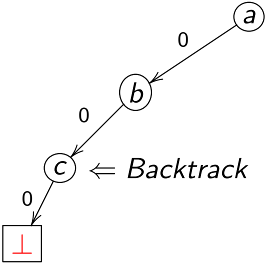


Implication Graph



Basic DPLL Procedure - DFS

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



$\Leftarrow$ *Backtrack*

Basic DPLL Procedure - DFS

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

$(a \vee c \vee d)$

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

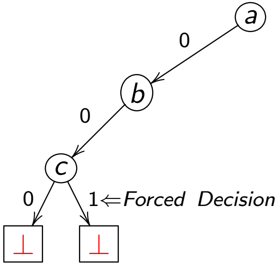
$(a \vee \bar{c} \vee d)$

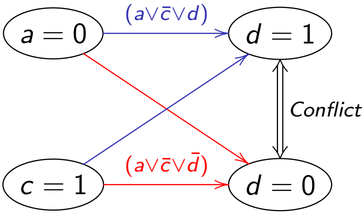
$(a \vee \bar{c} \vee \bar{d})$

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

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

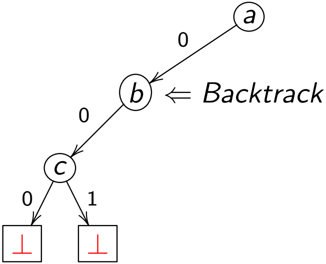
$(\bar{a} \vee \bar{b} \vee c)$





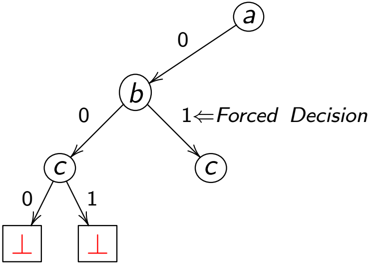
Basic DPLL Procedure - DFS

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



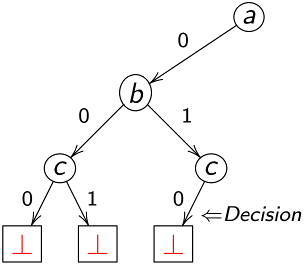
Basic DPLL Procedure - DFS

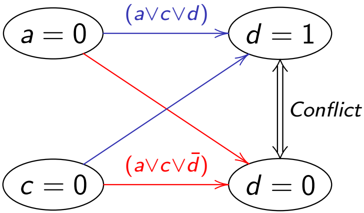
$(\bar{a} \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \bar{d})$
$(a \vee \bar{c} \vee d)$
$(a \vee \bar{c} \vee \bar{d})$
$(\bar{b} \vee \bar{c} \vee d)$
$(\bar{a} \vee b \vee \bar{c})$
$(\bar{a} \vee \bar{b} \vee c)$



Basic DPLL Procedure - DFS

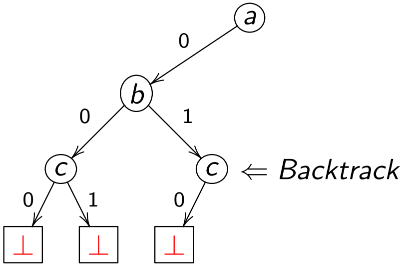
$(\bar{a} \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \bar{d})$
$(a \vee \bar{c} \vee d)$
$(a \vee \bar{c} \vee \bar{d})$
$(\bar{b} \vee \bar{c} \vee d)$
$(\bar{a} \vee b \vee \bar{c})$
$(\bar{a} \vee \bar{b} \vee c)$





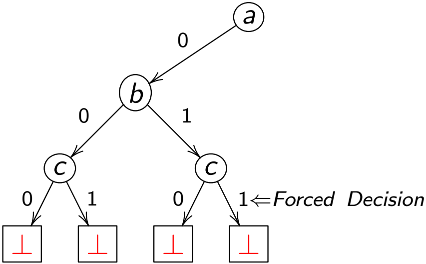
Basic DPLL Procedure - DFS

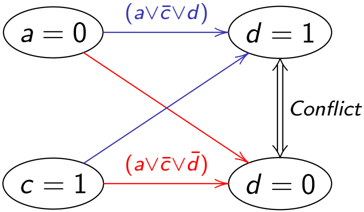
$(\bar{a} \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \bar{d})$
$(a \vee \bar{c} \vee d)$
$(a \vee \bar{c} \vee \bar{d})$
$(\bar{b} \vee \bar{c} \vee d)$
$(\bar{a} \vee b \vee \bar{c})$
$(\bar{a} \vee \bar{b} \vee c)$



Basic DPLL Procedure - DFS

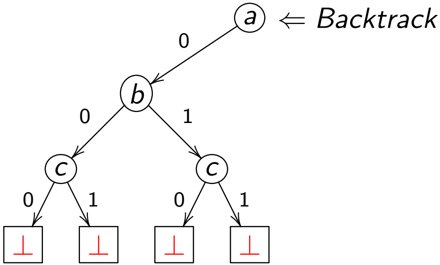
$(\bar{a} \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \bar{d})$
$(a \vee \bar{c} \vee d)$
$(a \vee \bar{c} \vee \bar{d})$
$(\bar{b} \vee \bar{c} \vee d)$
$(\bar{a} \vee b \vee \bar{c})$
$(\bar{a} \vee \bar{b} \vee c)$





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

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

$(a \vee c \vee d)$

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

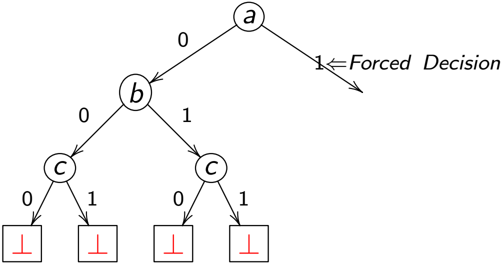
$(a \vee \bar{c} \vee d)$

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

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

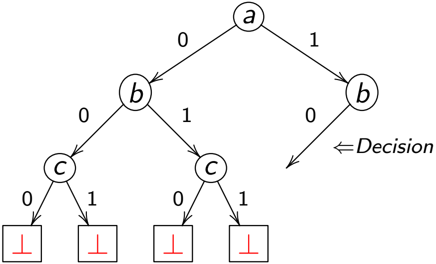
$(\bar{a} \vee b \vee \bar{c})$

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



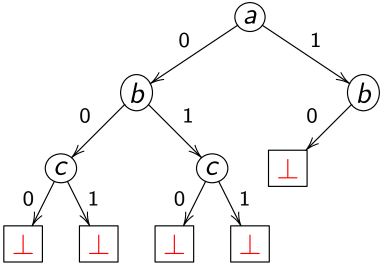
Basic DPLL Procedure - DFS

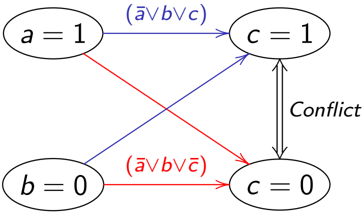
$(\bar{a} \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \bar{d})$
$(a \vee \bar{c} \vee d)$
$(a \vee \bar{c} \vee \bar{d})$
$(\bar{b} \vee \bar{c} \vee d)$
$(\bar{a} \vee b \vee \bar{c})$
$(\bar{a} \vee \bar{b} \vee c)$



Basic DPLL Procedure - DFS

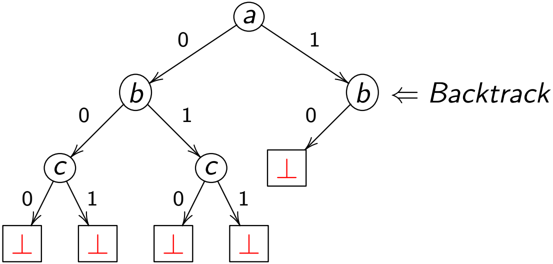
$(\bar{a} \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \bar{d})$
$(a \vee \bar{c} \vee d)$
$(a \vee \bar{c} \vee \bar{d})$
$(\bar{b} \vee \bar{c} \vee d)$
$(\bar{a} \vee b \vee \bar{c})$
$(\bar{a} \vee \bar{b} \vee c)$





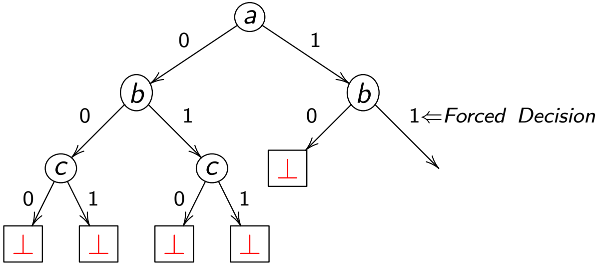
Basic DPLL Procedure - DFS

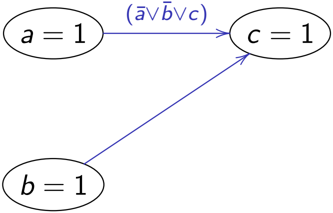
$(\bar{a} \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \bar{d})$
$(a \vee \bar{c} \vee d)$
$(a \vee \bar{c} \vee \bar{d})$
$(\bar{b} \vee \bar{c} \vee d)$
$(\bar{a} \vee b \vee \bar{c})$
$(\bar{a} \vee \bar{b} \vee c)$



Basic DPLL Procedure - DFS

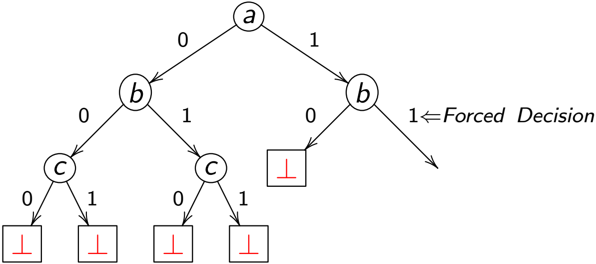
$(\bar{a} \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \bar{d})$
$(a \vee \bar{c} \vee d)$
$(a \vee \bar{c} \vee \bar{d})$
$(\bar{b} \vee \bar{c} \vee d)$
$(\bar{a} \vee b \vee \bar{c})$
$(\bar{a} \vee \bar{b} \vee c)$

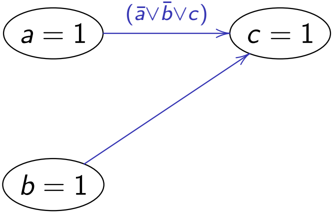




Basic DPLL Procedure - DFS

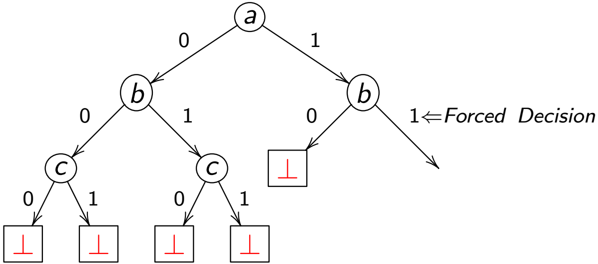
$(\bar{a} \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \bar{d})$
$(a \vee \bar{c} \vee d)$
$(a \vee \bar{c} \vee \bar{d})$
$(\bar{b} \vee \bar{c} \vee d)$
$(\bar{a} \vee b \vee \bar{c})$
$(\bar{a} \vee \bar{b} \vee c)$

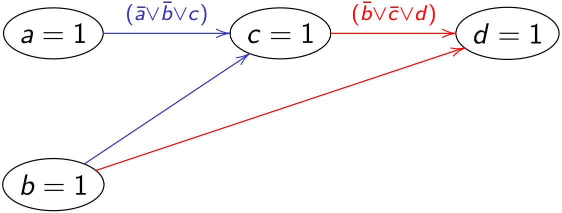




Basic DPLL Procedure - DFS

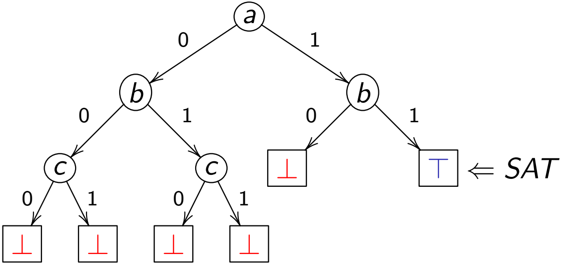
$(\bar{a} \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \bar{d})$
$(a \vee \bar{c} \vee d)$
$(a \vee \bar{c} \vee \bar{d})$
$(\bar{b} \vee \bar{c} \vee d)$
$(\bar{a} \vee b \vee \bar{c})$
$(\bar{a} \vee \bar{b} \vee c)$

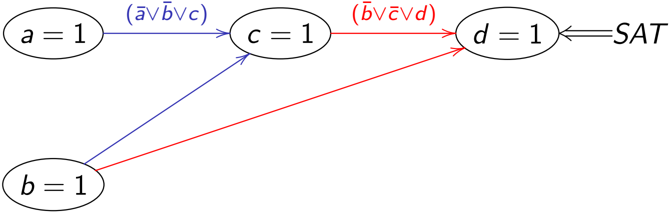




Basic DPLL Procedure - DFS

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





Features of DPLL

- Eliminate the potential memory explosion of DP
- Exponential time is still a problem
- Very limited size of problems are allowed
  - 32K word memory
  - Problem size limited by total size of clauses (about 1300 clauses)

Outline

GRASP

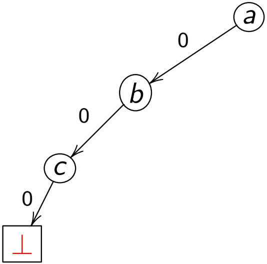
- Marques-Silva and Sakallah [SS96,SS99] (Univ. of Michigan)
  - J. P. Marques-Silva and K. A. Sakallah, "GRASP – A New Search Algorithm for Satisfiability", *Proc.ICCAD, 1996*.
  - J. P. Marques-Silva and Karem A. Sakallah, "GRASP: A Search Algorithm for Propositional Satisfiability", *IEEE Trans. Computers,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

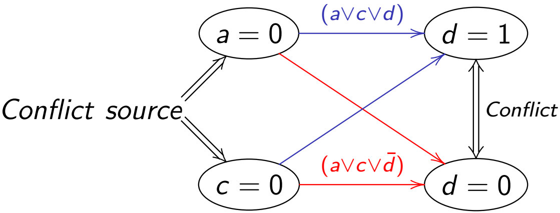
SAT Improvements

- 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

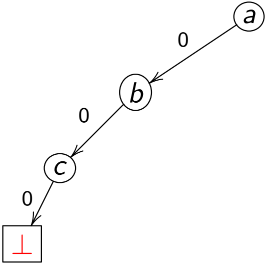
$(\bar{a} \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \bar{d})$
$(a \vee \bar{c} \vee d)$
$(a \vee \bar{c} \vee \bar{d})$
$(\bar{b} \vee \bar{c} \vee d)$
$(\bar{a} \vee b \vee \bar{c})$
$(\bar{a} \vee \bar{b} \vee c)$

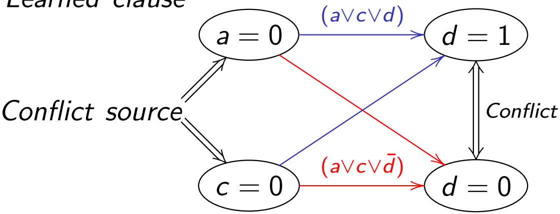




Conflict Driven Learning

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





Non-Chronological Backtracking

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

$(a \vee c \vee d)$

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

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

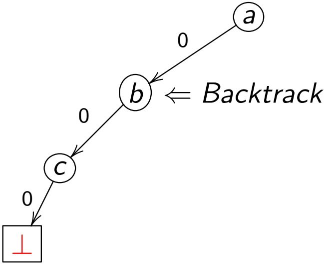
$(a \vee \bar{c} \vee \bar{d})$

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

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

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

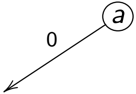
$(a \vee c)$ *Learned clause*

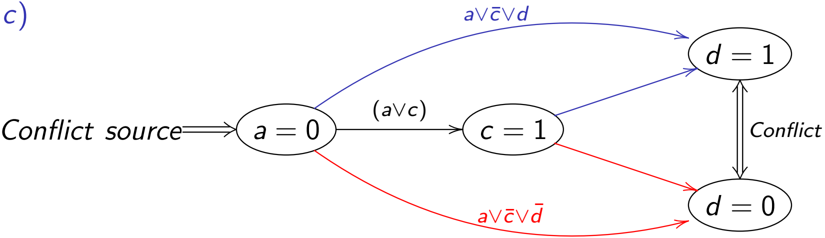


- ‘a’ is the next-to-the-last variable in the (current) learned clause.
  - c is the last (assigned) variable in this learned clause so a is called the next-to-the-last variable
  - Because of this learned clause, when a is assigned 0 then c will be implied and we don’t have to make decision for c
- 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

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

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

$(a \vee c \vee d)$

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

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

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

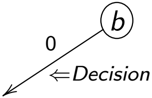
$(\bar{b} \vee \bar{c} \vee d)$

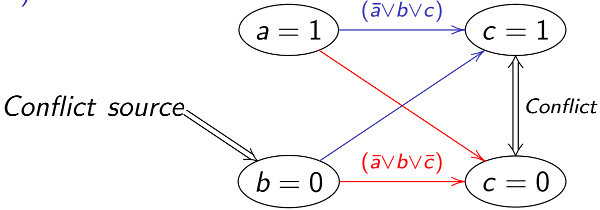
$(\bar{a} \vee b \vee \bar{c})$

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

$(a \vee c)$

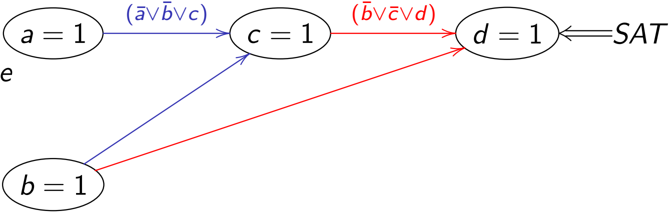
(a)





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*



More on Implication Graph

- How to determine the conflict source?

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

$(a \lor c \lor d)$

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

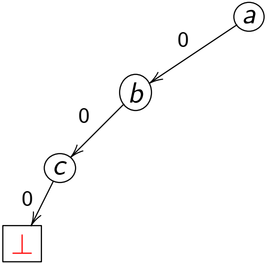
$(a \lor \bar{c} \lor d)$

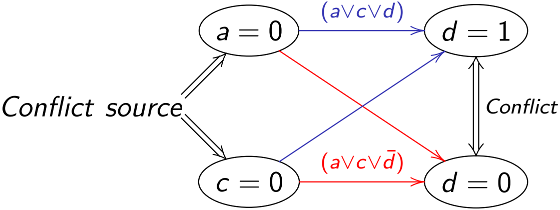
$(a \lor \bar{c} \lor \bar{d})$

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

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

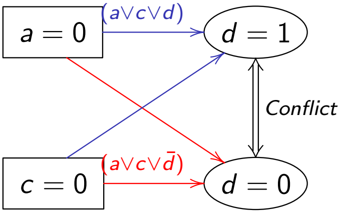
$(\bar{a} \lor \bar{b} \lor c)$





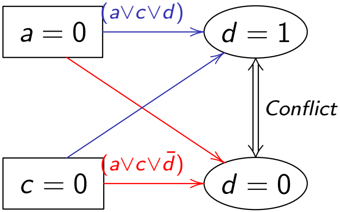
More on Implication Graph (cont.)

- How to determine the conflict source?
- We need to find a Cut on the Implication Graph, such that every path from the decision nodes to the conflict nodes must pass through it.
  - Decision nodes are the variables assigned to value in each decision process
  - Implication nodes are the variables assigned to value by implication
  - Conflict nodes are where the conflict shows up



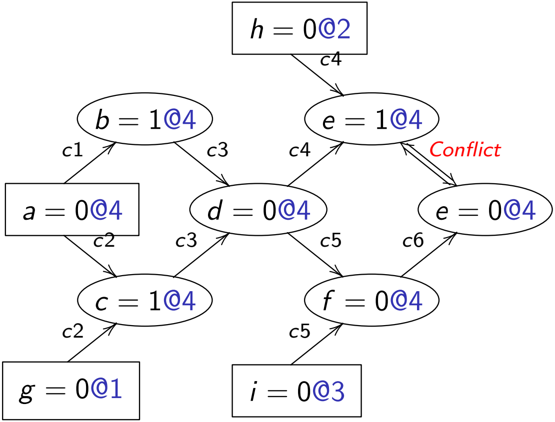
More on Implication Graph(cont.)

- Take all decision nodes to form a cut (technique used by *Rel_Sat*, [R. Bayardo R. Shrag, 1997])
- In this case, the cut consists of a=0 c=0.



Unique Implication Point(UIP)

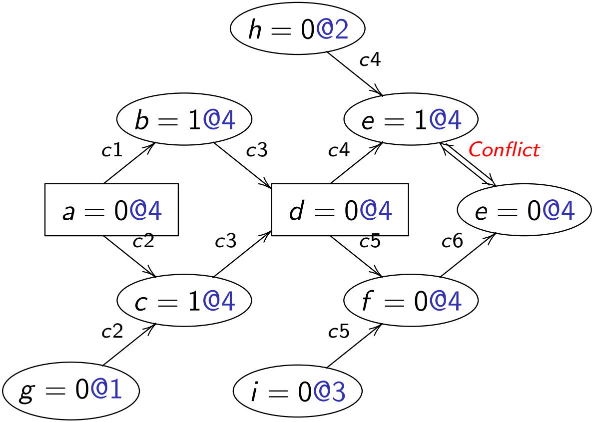
- Taking all decision nodes to form a cut may result in less valuable learnt clause.
- How about learning a clause that is close related to the conflict?



$$(a \vee b)$$
$$(a \vee c \vee g)$$
$$(\bar{b} \vee \bar{c} \vee d)$$
$$(\bar{d} \vee e \vee h)$$
$$(\bar{d} \vee f \vee i)$$
$$(\bar{e} \vee \bar{f})$$
$$(\bar{d} \vee h \vee i)$$

Unique Implication Point(UIP)

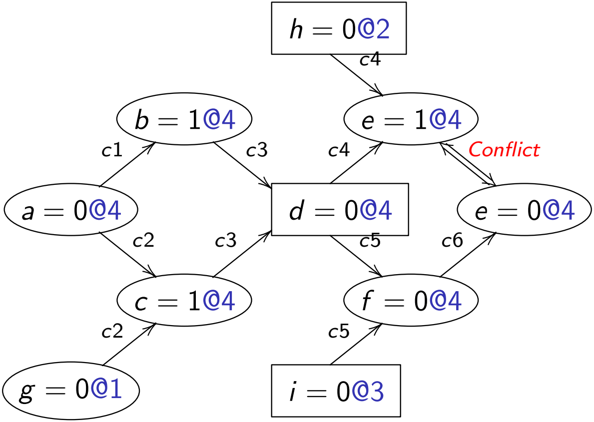
- A UIP is any node at the current decision level (@4) such that any path from decision variable (a=0@4) to the conflict nodes must pass through it.



$$(a \vee b)$$
$$(a \vee c \vee g)$$
$$(\bar{b} \vee \bar{c} \vee d)$$
$$(\bar{d} \vee e \vee h)$$
$$(\bar{d} \vee f \vee i)$$
$$(\bar{e} \vee \bar{f})$$
$$(\bar{d} \vee h \vee i)$$

Unique Implication Point(UIP)

- First-UIP Learning Scheme: used by MiniSAT and zChaff
- Set the cut right before the first UIP is encountered on the path leading from the conflict nodes.



$(a \vee b)$

$(a \vee c \vee g)$

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

$(\bar{d} \vee e \vee h)$

$(\bar{d} \vee f \vee i)$

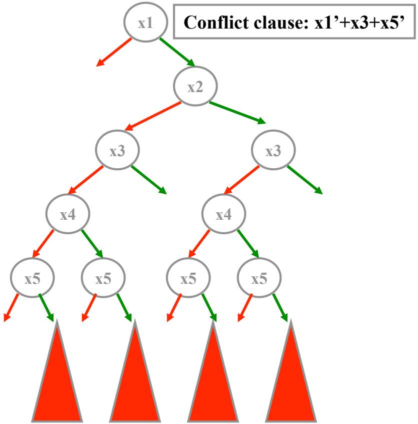
$(\bar{e} \vee \bar{f})$

$(\bar{d} \vee h \vee i)$

$(h \vee d \vee i)$

What's the big deal?

- We can know learn the related clause to each conflict we encountered.
- Significantly prune the search space because learned clause is useful forever!



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

1. Fundamental Concepts

2. Core algorithms of satisfiability problems
   - Davis-Putnam Algorithm
   - DPLL Algorithm
   - GRASP Algorithm
   - zChaff Algorithm

3. Heuristics

4. SAT competitions

5. Applications

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 (Unit propagation, implication) 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.
  - This is not a pruning technique, but saves time while performing BCP.

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.

$v2 \lor v3 \lor v1 \lor v4 \lor v5$

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

$v1 \lor \overline{v2}$

$\overline{v1} \lor v4$

$\overline{v1} \longleftarrow$ 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.

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

$$State : v1 = F$$

$$Pending :$$

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.

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

$State : v1 = F$

$Pending :$

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`.

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

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

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

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.

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

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

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

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.

$$
\begin{array}{llcl}
\Rightarrow & v2 \vee v3 \vee v1 \vee v4 \vee v5 & \Longrightarrow & v2 \vee v3 \vee v1 \vee v4 \vee v5 \\
\Rightarrow & v1 \vee v2 \vee \overline{v3} & & v1 \vee v2 \vee \overline{v3} \\
 & v1 \vee \overline{v2} & & v1 \vee \overline{v2} \\
 & \overline{v1} \vee v4 & & \overline{v1} \vee v4 \\
 & & & \\
 & State : v1 = F, v2 = F & & State : v1 = F, v2 = F \\
 & & & \\
 & Pending : & \Longrightarrow & Pending : (v3 = F)
\end{array}
$$

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 $v4 =$ `True` and proceed.

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

$$\begin{array}{ll} State: v1 = F, v2 = F, & \quad State: v1 = F, v2 = F, \\ \qquad v3 = F & \quad \qquad v3 = F \\ Pending: & \quad Pending: \end{array}$$

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.

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

$$\begin{aligned} State &: v1 = F, v2 = F, \\ &\quad v3 = F, v4 = T \\ Pending &: \end{aligned}$$

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

Outline

1. Fundamental Concepts

2. Core algorithms of satisfiability problems

3. Heuristics
   - Decision heuristics
   - Restart mechanism

4. SAT competitions

5. Applications

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

1 Fundamental Concepts

2 Core algorithms of satisfiability problems

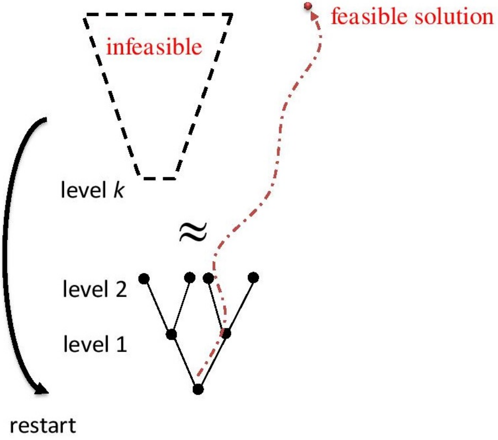
3 Heuristics
  - Decision heuristics
  - Restart mechanism

4 SAT competitions

5 Applications

Restart Motivation

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



Restart

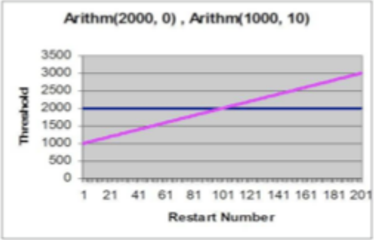
- Motivation: avoid spending too much time in “bad” branches.
  - no easy-to-find satisfying assignment
  - no opportunity for fast learning of strong clauses.
- All modern SAT solvers use a restart policy.
  - Following various criteria, the solver is forced to backtrack to level 0.
  - Abandon the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

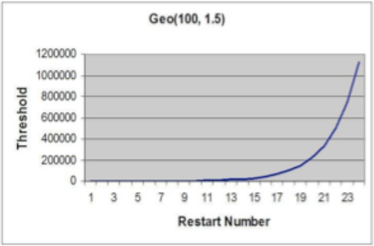
- All existing techniques use the number of conflicts learned as of the previous restart.
- The difference is only in the method of calculating the threshold.

Restarts strategies

- Arithmetic (or fixed) series
  - Used in Berkmin, Eureka, zChaff, Siege

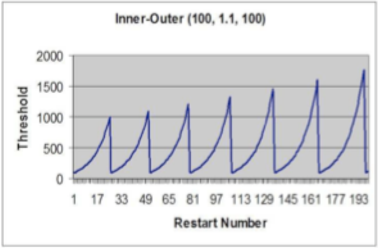


- Geometric series
  - Used in Minisat 2007



Restarts strategies

- Inner-Outer Geometric series
  - Used in Picosat



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...
  - Add Constraints according to the previous results
    - Since relevant learnt clauses are preserved, we speed up the later exploration.

Other Issues

- Reducing Learnt Clause
  - Large CNF can result in large amount of learnt clauses, and some of them may not be useful until later
  - They slow down BCP
- Remove learnt clauses periodically
  - Keep a certain number of learnt clauses
  - Minisat removes half of the learnt clauses if the number of clauses reaches threshold (which grows geometrically)
  - Glucose keeps short learnt clauses forever, but removes long ones if if the number of clauses reaches threshold (which grows arithmetically)

Other Issues

- Refutation proof, i.e., proof of UNSAT (Ex.Resolution Proof)
- Parallel computation
- Memory management
- etc...

Outline

1. Fundamental Concepts
2. Core algorithms of satisfiability problems
3. Heuristics
4. **SAT competitions**
5. Applications

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 of C code 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 (Static Build)

- 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 ResultFileName`

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
  ```
- What if we want yet another solution?
  Add block clauses and solve again!
  Problem: starting from scratch, file I/O

The Usage of the MiniSat (C++ API)

- MiniSat Page: http://minisat.se/
- MiniSat fork with CMake Integration: https://github.com/master-keying/minisat/
- MiniSat provides elegant c++ API

The Usage of the MiniSat (C++ API)

- Usage
  - Unzip the package `unzip minisat-master.zip`
  - Build minisat to target directory (If you would like a static build of MiniSat, do not build under the recent directory because of naming alias) `cmake -S . -B path`
  - Make and compile minisat build `make`
  - Write your code(details in later slides) `vim main.cpp`
  - Provide CMakeLists.txt(details in later slides vim CMakeLists.txt
  - Make and compile your program `cmake; make; ./demo`

API Usage

- Solver.h class Solver
  - newVar(bool polarity, bool dvar)
  - addClause(vec$<$ *Lit* $>$ ps)
  - addClause(Lit p)
  - addEmptyClause()
  - simplify()
  - solve()
- Your program
  - Construct Solver Solver s;
  - Add all constraints(clauses) s.addClause(ps)
  - solve s.solve();

CMake Guide

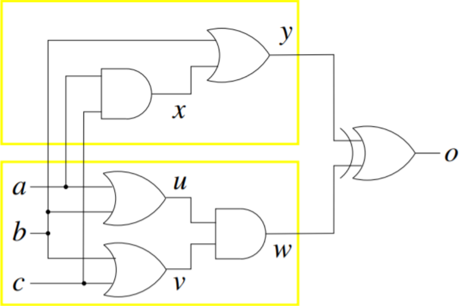
- Trivial CMakeList.txt
  - Identify CMake version
    `cmake_minimum_required(VERSION 3.5)`
  - Project name
    `project (projectname LANGUAGES CXX)`
  - Include all codes
    `add_executable(exename main.cpp)`
  - Add subdirectory
    `add_subdirectory(dir)`
  - Link to libminisat.a
    `target_link_libraries(exename MiniSat::libminisat)`
- Of course, you can build on your own, but the previous five lines is enough for now.

Demo 1 - Simple case

- Solve $(A \vee B \vee C) \wedge (\bar{A} \vee B \vee C) \wedge (A \vee \bar{B} \vee C) \wedge (A \vee B \vee \bar{C})$
- What if we need more than one solution? Add code!

Demo 2 - Equivalence Checking

- Check if the two circuits in the yellow boxes are equivalent?



$$\begin{aligned} & {\color{blue}o} \wedge \\ (x &\leftrightarrow a \wedge c) \wedge \\ (y &\leftrightarrow b \vee x) \wedge \\ (u &\leftrightarrow a \vee b) \wedge \\ (v &\leftrightarrow b \vee c) \wedge \\ (w &\leftrightarrow u \wedge v) \wedge \\ (o &\leftrightarrow y \oplus w) \end{aligned}$$

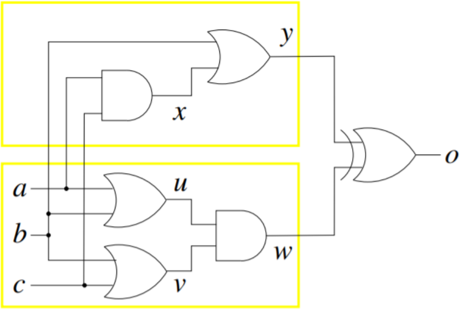
How to write CNF for circuits?

- Tseytin transformation
- $A \rightarrow B$ $(\bar{A} \vee B)$
- $A \leftrightarrow B$ $(\bar{A} \vee B) \wedge (A \vee \bar{B})$

Type	Operation	CNF Sub-expression
AND	$C = A \cdot B$	$(\bar{A} \vee \bar{B} \vee C) \wedge (A \vee \bar{C}) \wedge (B \vee \bar{C})$
NAND	$C = \overline{A \cdot B}$	$(\bar{A} \vee \bar{B} \vee \bar{C}) \wedge (A \vee C) \wedge (B \vee C)$
OR	$C = A + B$	$(A \vee B \vee \bar{C}) \wedge (\bar{A} \vee C) \wedge (\bar{B} \vee C)$
NOR	$C = \overline{A + B}$	$(A \vee B \vee C) \wedge (\bar{A} \vee \bar{C}) \wedge (\bar{B} \vee \bar{C})$
NOT	$C = \bar{A}$	$(\bar{A} \vee \bar{C}) \wedge (A \vee C)$
XOR	$C = A \oplus B$	$(\bar{A} \vee \bar{B} \vee \bar{C}) \wedge (A \vee B \vee \bar{C}) \wedge (A \vee \bar{B} \vee C) \wedge (\bar{A} \vee B \vee C)$
XNOR	$C = \overline{A \oplus B}$	$(\bar{A} \vee \bar{B} \vee C) \wedge (A \vee B \vee C) \wedge (A \vee \bar{B} \vee \bar{C}) \wedge (\bar{A} \vee B \vee \bar{C})$

Equivalence Checking

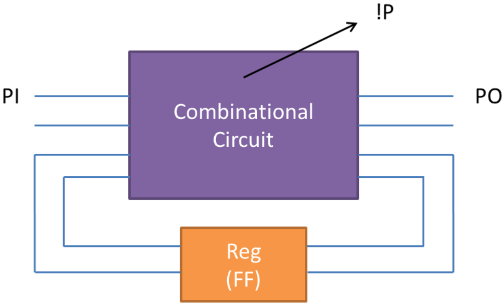
- Miter: Link the output of the two circuits with an XOR
- If the circuits are equivalent, signal O should always be False.
- By asserting O as a unit clause, the CNF formula should be UNSAT if the circuits are equivalent.



$$\begin{array}{r}
o \;\wedge \\
(x \leftrightarrow a \wedge c) \wedge \\
(y \leftrightarrow b \vee x) \wedge \\
(u \leftrightarrow a \vee b) \wedge \\
(v \leftrightarrow b \vee c) \wedge \\
(w \leftrightarrow u \wedge v) \wedge \\
(o \leftrightarrow y \oplus w)
\end{array}$$

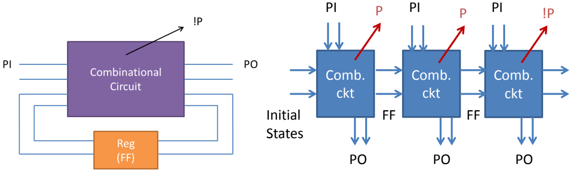
Optional: Bounded Model Checking

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



Optional: Timeframe Expansion Model

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

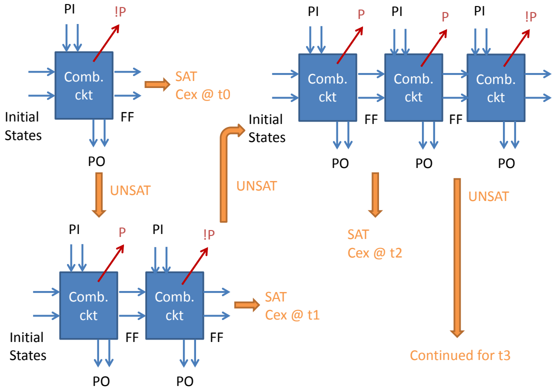


Optional: BMC Algorithm

- 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

Optional: BMC Algorithm

- In other words ...



Solving Various Problems

- We now know how to use SAT.
- For many NP problems, SAT is a powerful tool. All you need is to develop proper SAT formulation, i.e. encoding your constraints into CNF formula
- Small Toy: online CNF generators