Basics of SAT Solving Algorithms

Sol Swords

December 8, 2008

Outline

Outline

What is a SAT problem?

Given a propositional formula (Boolean variables with AND, OR, NOT), is there an assignment to the variables such that the formula evaluates to true?

- NP-complete problem with applications in AI, formal methods
- Input usually given as Conjunctive Normal Form formulas - linear reduction from general propositional formulas

Conjunctive Normal Form

SAT solvers usually take input in CNF: an AND of ORs of literals.

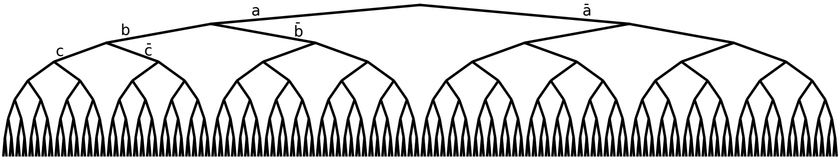
- *Atom* - a propositional variable: a, b, c
- *Literal* - an atom or its negation: $a, \bar{a}, b, \bar{b}$
- *Clause* - A disjunction of some literals: $a \vee \bar{b} \vee c$
- *CNF formula* - A conjunction of some clauses: $(a \vee \bar{b} \vee c) \wedge (\bar{c} \vee \bar{a})$

A formula is *satisfied* by a variable assignment if every clause has at least one literal which is true under that assignment.
A formula is *unsatisfied* by a variable assignment if some clause's literals are all false under that assignment.

Outline

The Bare Gist of DPLL-based SAT algorithms

- Perform a depth-first search through the space of possible variable assignments. Stop when a satisfying assignment is found or all possibilities have been tried.



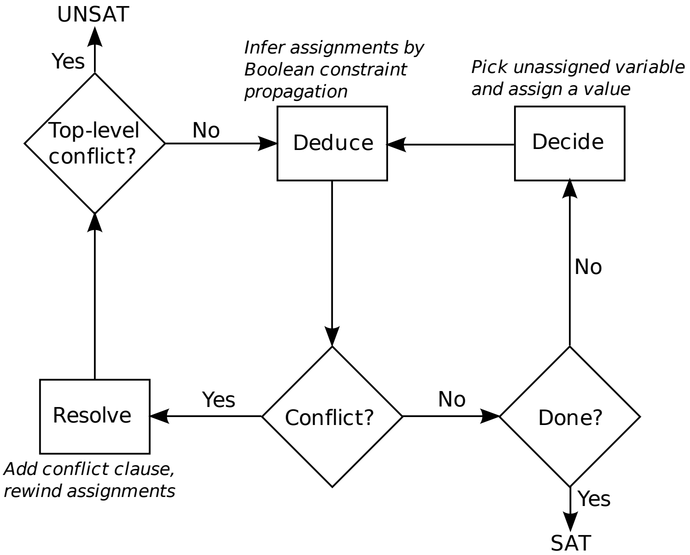
Many optimizations possible:

- Skip branches where no satisfying assignments can occur.
- Order the search to maximize the amount of the search space that can be skipped.

Slightly More Detailed Sketch

Repeat:

- *Decide*: Select some unassigned variable and assign it a value.
  - If all variables are assigned, return SAT.
- *Deduce*: Infer values of other variables that follow from that assignment and detect conflicts.
- *Resolve*: In case of conflict, record a new clause prohibiting that conflict; undo the assignments leading to the conflict.
  - If it's a top-level conflict (the conflict clause is empty), return UNSAT.


UNSAT
Yes
Top-level conflict?
No
Infer assignments by Boolean constraint propagation
Deduce
Pick unassigned variable and assign a value
Decide
No
Resolve
Yes
Conflict?
No
Done?
Add conflict clause, rewind assignments
Yes
SAT

Ways to make DPLL Faster

- *Decide*: Use a good heuristic to select among unassigned variables
  - activity heuristic based on how often a variable is involved in a conflict
- *Deduce*: Use a good trade-off between speed and completeness
  - Boolean constraint propagation with watched literals
  - Typically about 80% of SAT-solver runtime
- *Resolve*: Take advantage of information revealed by conflicts without over-growing the clause set
  - Learn one or more new clauses at each conflict
  - Backtrack to the "root cause" of the conflict
  - Delete conflict clauses based on an activity heuristic to keep the working set small

Outline

Boolean Constraint Propagation

Two simple rules:

- If all but one of a clause's literals are assigned FALSE and the remaining literal is unassigned, assign it TRUE.
- If all of a clause's literals are assigned FALSE, return UNSAT.

Naive algorithm: Inspect each clause and apply the rule; repeat until no new assignments are made.

Motivation for watched literal method

Ideal BCP: Each clause is inspected only after all but one literal is assigned false.

- Nothing is accomplished by inspecting a clause when it is satisfied or when multiple literals are unassigned.

Best known way to approximate this ideal:

- Associate each clause with two of its unassigned literals
- Only examine the clause when one of them is assigned false.

Watched Literal Algorithm

When a literal a is assigned true:

- For each clause k in the watch list of $\bar{a}$, do:
  - If all but one literal b is assigned false, assign b true (and recur);
  - If all literals are assigned false, exit (UNSAT);
  - If any literal is assigned true, continue;
  - Otherwise, add k to the watch list of one of its remaining unassigned literals and remove it from the watch list of $\bar{a}$.

Notes:

- Low overhead, large reduction in number of clause inspections relative to naive algorithms.
- Tricky to maintain all the right invariants so that backtracking doesn't break the watch lists.

Watched Literals Example

- Watched literals $a, \bar{b}$, all literals unassigned

$$\color{red}{a}\ \color{red}{\bar{b}}\ c\ d$$

Watched Literals Example

- Watched literals $a, \bar{b}$, all literals unassigned
- $\bar{c}$ assigned: don't inspect

$$a \ \bar{b} \ c \ d$$

Watched Literals Example

- Watched literals $a, \bar{b}$, all literals unassigned
- $\bar{c}$ assigned: don't inspect
- b assigned: inspect,

$$a \quad \bar{b} \quad c \quad d$$

Watched Literals Example

- Watched literals $a, \bar{b}$, all literals unassigned
- $\bar{c}$ assigned: don't inspect
- b assigned: inspect, choose new watched literal d

$$a \quad \bar{b} \quad c \quad d$$

Watched Literals Example

- Watched literals $a, \bar{b}$, all literals unassigned
- $\bar{c}$ assigned: don't inspect
- b assigned: inspect, choose new watched literal d
- $\bar{d}$ assigned: inspect,

$$a \ \bar{b} \ c \ d$$

Watched Literals Example

- Watched literals $a, \bar{b}$, all literals unassigned
- $\bar{c}$ assigned: don't inspect
- b assigned: inspect, choose new watched literal d
- $\bar{d}$ assigned: inspect, propagate a

$$a \ \bar{b} \ c \ d$$

Watched Literals Example

- Watched literals $a, \bar{b}$, all literals unassigned
- $\bar{c}$ assigned: don't inspect
- b assigned: inspect, choose new watched literal d
- $\bar{d}$ assigned: inspect, propagate a
- Backtrack to before b

$$a \quad \bar{b} \quad c \quad d$$

Watched Literals Example

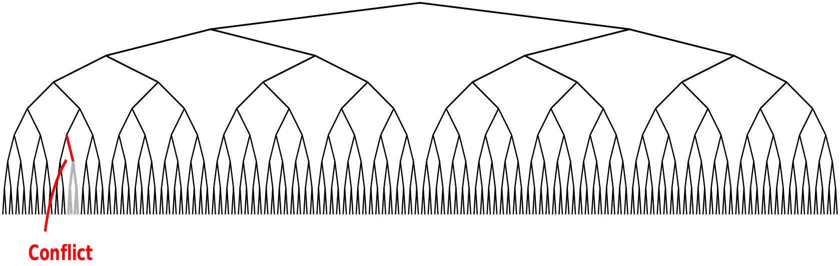
- Watched literals $a, \bar{b}$, all literals unassigned
- $\bar{c}$ assigned: don't inspect
- b assigned: inspect, choose new watched literal d
- $\bar{d}$ assigned: inspect, propagate a
- Backtrack to before b
- a assigned: don't inspect

$$a \ \bar{b} \ c \ d$$

Outline

Conflict Analysis

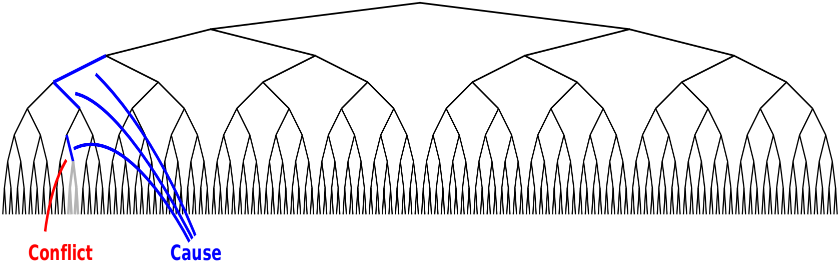
Conflicts can be exploited to reduce the space to be searched.



- Find a conflict (skip the subtree where it's rooted)

Conflict Analysis

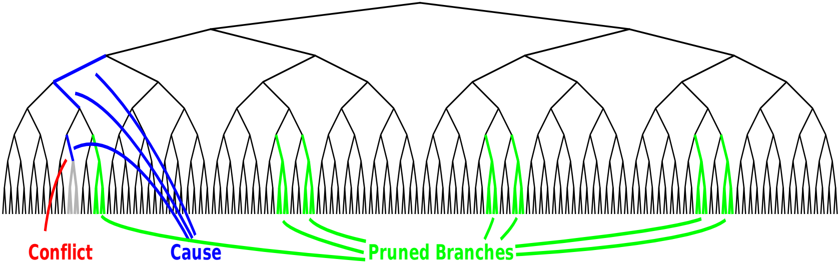
Conflicts can be exploited to reduce the space to be searched.



- Find a conflict (skip the subtree where it's rooted)
- Analyze the conflict to find a sufficient condition

Conflict Analysis

Conflicts can be exploited to reduce the space to be searched.



- Find a conflict (skip the subtree where it's rooted)
- Analyze the conflict to find a sufficient condition
- Skip future areas of search space where the condition holds

Conflict Clauses

- To prune branches where $a = \mathit{false}$, $b = \mathit{true}$, $c = \mathit{true}$, add conflict clause $a\ \bar{b}\ \bar{c}$.
- Pruning is implicit in BCP.
- Use learned clause to determine how far to backtrack.
  - Backtrack to earliest decision level in which exactly one variable is unassigned.
- How to calculate this clause?

Calculating a conflict clause

To analyze a clause c:

- For each false literal x in c, either add x to the conflict clause or analyze the clause c' which propagated $\bar{x}$ (heuristic decision.)

Picture: clause $a\ \bar{d}\ \bar{e}\ g$ causes a conflict, yielding conflict clause $a\ \bar{b}\ \bar{c}$.

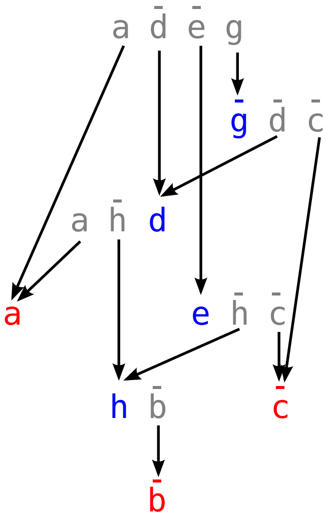
- Can construct a resolution proof of the new clause from this process:

$$a\ \bar{d}\ \bar{e}\ g,\ \ \bar{g}\ \bar{d}\ \bar{c} \Rightarrow a\ \bar{d}\ \bar{e}\ \bar{c}$$

$$a\ \bar{d}\ \bar{e}\ \bar{c},\ \ a\ \bar{h}\ d \Rightarrow a\ \bar{e}\ \bar{c}\ \bar{h}$$

$$a\ \bar{e}\ \bar{c}\ \bar{h},\ \ e\ \bar{h}\ \bar{c} \Rightarrow a\ \bar{c}\ \bar{h}$$

$$a\ \bar{c}\ \bar{h},\ \ h\ \bar{b} \Rightarrow a\ \bar{b}\ \bar{c}$$



Conflict clause heuristics

Issue: Include a literal in the conflict clause, or explore the clause that caused its assignment?

- No choice about decision literals
- Goal: Small, relevant conflict clauses
  - Possible to generate more than one clause from a conflict, but most solvers don't
- Typical choice is "First UIP" (Unique Implication Point) strategy:
  - Never explore causes of literals assigned due to previous decisions
  - Generate the smallest clause that includes exactly one literal from the current decision level

Outline

Ordering, Deletion, Restarting

Useful high-level strategic heuristics:

- Choose decision literals by activity heuristic: how often has a literal (recently) been involved conflicts?
  - Many tweaks possible
  - Choose randomly some small percentage of the time
- Periodically delete some conflict clauses to keep the working set small
  - Various heuristics: activity, size, number of currently-assigned literals
  - A clause is "locked" (may not be deleted) if it is the reason for a current assignment
- Periodically restart the search while keeping some learned clauses
  - Try to avoid "dead ends" where heuristics are pushing in the wrong direction
  - Most solvers increase limitations on backtracks and learned clauses at each restart

Outline

References

- Marques-Silva & Sakallah, *GRASP: A search algorithm for propositional satisfiability*
- Moskewicz et al, *Chaff: Engineering an efficient SAT solver*
- Een & Sorensson, *An extensible SAT-solver*
- Zhang et al, *Efficient conflict driven learning in a Boolean satisfiability solver*