C440: Software Reliability

SAT Solving: Basics and DPLL

Cristian Cadar

Based on slides by Ișıl Dillig and Hristina Palikareva

SAT Problem

input a Boolean formula φ (usually in CNF)

decide whether or not there exists an assignment of φ's variables under which φ evaluates to true

- SAT Solver is an algorithm (and tool) for solving the SAT problem.
- SAT is NP-complete [Cook'70]
  - SAT algorithms worst-case exponential in time
- Modern SAT solvers employ various heuristics
  - Perform exceptionally well in practice
  - Have had deep impact on fields such as testing and verification

DPLL

- Most modern SAT solvers based on the DPLL framework
  - Due to Davis, Putnam, Loveland, Logemann, 1962
- DPLL operates on formulas in normal form, which we will review shortly

Boolean Formulas and CNF

Boolean Formulas

- Defined inductively by the following grammar:

$$\varphi ::= \text{true} \mid \text{false} \mid x \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \rightarrow \varphi_2 \mid \varphi_1 \leftrightarrow \varphi_2$$

- x is a variable
- Formulas of the form x and $\neg x$ are literals
- Example $p \leftrightarrow (q \rightarrow \neg r)$
  - Variables appearing in the formula?
  - Literals appearing in the formula?

Normal Forms

- A normal form of a formula F is another formula F' such that F is equivalent to F', but F' obeys certain syntactic restrictions.
- There are three kinds of normal forms that are interesting in propositional logic:
  - Negation Normal Form (NNF)
  - Disjunctive Normal Form (DNF)
  - Conjunctive Normal Form (CNF)

Negation Normal Form (NNF)

Negation Normal Form requires two syntactic restrictions:

- The only logical connectives are $\neg, \wedge, \vee$ (i.e., no $\rightarrow$, $\leftrightarrow$)
- Negations are only applied to literals
- Is formula $p \vee (\neg q \wedge (r \vee \neg s))$ in NNF?
- What about $p \vee (\neg q \wedge \neg(\neg r \wedge s))$?
- What about $p \vee (\neg q \wedge (\neg\neg r \vee \neg s))$?

Conversion to NNF I

- To make sure the only logical connectives are $\neg, \wedge, \vee$, need to eliminate $\rightarrow$ and $\leftrightarrow$
- How do we express $F_1 \rightarrow F_2$ using $\vee, \wedge, \neg$?
- How do we express $F_1 \leftrightarrow F_2$ using only $\neg, \wedge$. $\vee$?

Conversion to NNF II

- Also need to ensure negations appear only in literals: push negations in
- Use DeMorgan's laws to distribute $\neg$ over $\wedge$ and $\vee$:

$$\neg(F_1 \wedge F_2) \Leftrightarrow \neg F_1 \vee \neg F_2$$

$$\neg(F_1 \vee F_2) \Leftrightarrow \neg F_1 \wedge \neg F_2$$

- We also disallow double negations:

$$\neg\neg F \Leftrightarrow F$$

NNF Example

Convert $F: \neg(p \rightarrow (p \wedge q))$ to NNF

Disjunctive Normal Form (DNF)

- A formula in disjunctive normal form is a disjunction of conjunction of literals.

$$\bigvee_i \bigwedge_j \ell_{i,j} \quad \text{for literals } \ell_{i,j}$$

- i.e., $\vee$ can never appear inside $\wedge$ or $\neg$
- Called disjunctive normal form because disjuncts are at the outer level
- Each inner conjunction is called a clause
- Question: If a formula is in DNF, is it also in NNF?

Conversion to DNF

- To convert formula to DNF, first convert it to NNF.
- Then, distribute $\wedge$ over $\vee$:

$$(F_1 \vee F_2) \wedge F_3 \quad \Leftrightarrow \quad (F_1 \wedge F_3) \vee (F_2 \wedge F_3)$$
$$F_1 \wedge (F_2 \vee F_3) \quad \Leftrightarrow \quad (F_1 \wedge F_2) \vee (F_1 \wedge F_3)$$

Example

Convert $F : \; (q_1 \; \vee \; \neg\neg q_2) \; \wedge \; (\neg r_1 \; \rightarrow \; r_2)$ into DNF

DNF and Satisfiability

- Claim: If formula is in DNF, trivial to determine satisfiability. How?
-
-
- Idea: To determine satisfiability, convert formula to DNF and just do a syntactic check.

DNF and Blow-up in formula size

- This idea is completely impractical. Why?
- Consider formula: $(F_1 \vee F_2) \wedge (F_3 \vee F_4)$
- In DNF:

$$(F_1 \wedge F_3) \vee (F_1 \wedge F_4) \vee (F_2 \wedge F_3) \vee (F_2 \wedge F_4)$$

- Every time we distribute, formula size doubles!
- Moral: DNF conversion causes exponential blow-up in size!
- Checking satisfiability by converting to DNF is almost as bad as truth tables!

Conjunctive Normal Form (CNF)

- A formula in conjuctive normal form is a conjunction of disjunction of literals.

$$\bigwedge_i \bigvee_j \ell_{i,j} \quad \text{for literals } \ell_{i,j}$$

- i.e., $\wedge$ not allowed inside $\vee, \neg$.
- Called conjunctive normal form because conjucts are at the outer level
- Each inner disjunction is called a clause
- Is formula in CNF also in NNF?

Conversion to CNF

- To convert formula to CNF, first convert it to NNF.
- Then, distribute $\lor$ over $\land$:

$$
\begin{array}{rcl}
(F_1 \land F_2) \lor F_3 & \Leftrightarrow & (F_1 \lor F_3) \land (F_2 \lor F_3) \\
F_1 \lor (F_2 \land F_3) & \Leftrightarrow & (F_1 \lor F_2) \land (F_1 \lor F_3)
\end{array}
$$

CNF Conversion Example

Convert $F : \ (p \leftrightarrow (q \rightarrow r))$ into CNF

DNF vs. CNF

- Fact: Unlike DNF, it is not trivial to determine satisfiability of formula in CNF.
- Does CNF conversion cause exponential blow-up in size?
- News: But almost all SAT solvers first convert formula to CNF before solving!

Why CNF?

- Question: If it is just as expensive to convert formula to CNF as to DNF, why do solvers convert to CNF although it is much easier to determine satisfiability in DNF?

-

Equisatisfiability

- Two formulas F and F' are equisatisfiable iff:

  F is satisfiable if and only if F' is satisfiable

- If two formulas are equisatisfiable, are they equivalent?
- Example:
-
- Equisatisfiability is a much weaker notion than equivalence.
- But useful if all we want to do is determine satisfiability.

The Plan

- To determine satisfiability of F, convert formula to equisatisfiable formula F' in CNF
- Use an algorithm (DPLL) to decide satisfiability of F'
- Since F' is equisatisfiable to F, F is satifiable iff algorithm decides F' is satisfiable
- Big question: How do we convert formula to equisatisfiable formula without causing exponential blow-up in size?

Tseitin's Transformation

Tseitin's transformation converts formula F to equisatisfiable formula F' in CNF with only a linear increase in size.

Tseitin's Transformation I

- Step 1: Introduce a new variable p_G for every subformula G of F (unless G is already a single variable).
- For instance, if $F = G_1 \wedge G_2$, introduce two variables p_{G_1} and p_{G_2} representing G_1 and G_2 respectively.
- p_{G_1} is said to be representative of G_1 and p_{G_2} is representative of G_2.

Tseitin's Transformation II

- Step 2: Consider each subformula

$$G : G_1 \circ G_2 \quad (\circ \text{ arbitrary boolean connective})$$

- Stipulate representative of G is equivalent to representative of $G_1 \circ G_2$

$$p_G \leftrightarrow p_{G_1} \circ p_{G_2}$$

- Step 3: Convert $p_G \leftrightarrow p_{G_1} \circ p_{G_2}$ to equivalent CNF (by converting to NNF and distributing $\vee$'s over $\wedge$'s).

- Observe: Since $p_G \leftrightarrow p_{G_1} \circ p_{G_2}$ contains at most three propositional variables and exactly two connectives, size of this formula in CNF is bound by a constant.

Tseitin's Transformation II

- Given original formula F, let p_F be its representative and let S_F be the set of all subformulas of F (including F itself).
- Then, introduce the formula

$$p_F \wedge \bigwedge_{G=(G_1 \circ G_2) \in S_F} CNF(p_g \leftrightarrow p_{g_1} \circ p_{g_2})$$

- Claim: This formula is equisatisfiable to F.
- The proof is by structural induction
- Formula is also in CNF because conjunction of CNF formulas is in CNF.

Tseitin's Transformation and Size

- Using this transformation, we converted F to an equisatisfiable CNF formula F'.
- What about the size of F'?

$$p_F \wedge \bigwedge_{G=(G_1 \circ G_2) \in S_F} CNF(p_g \leftrightarrow p_{g_1} \circ p_{g_2})$$

- $|S_F|$ is bound by the number of connectives in F.
- Each formula $CNF(p_g \leftrightarrow p_{g_1} \circ p_{g_2})$ has constant size.
- Thus, trasformation causes only linear increase in formula size.

Tseitin's Transformation Example

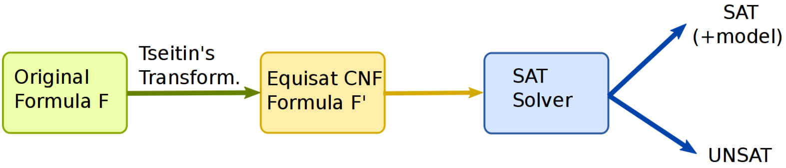
Convert $F : (p \vee q) \rightarrow (p \wedge \neg r)$ to equisatisfiable CNF formula.

1.

2.

3.

SAT Solvers



- A model is a (total or partial) assignment of variables to $\top$ or $\bot$ that makes the formula $\top$
- How do you map the assignment to F' to an assignment to F?
  - Simply drop assignments to new representative variables

DPLL: Historical Perspective

- Almost all SAT solvers today are based on an algorithm called DPLL (Davis-Putnam-Logemann-Loveland)
- 1962: the original algorithm known as DP (Davis-Putnam) $\Rightarrow$ "simple" procedure for automated theorem proving

- Davis and Putnam hired two programmers, George Logemann and David Loveland, to implement their ideas on the IBM 704.
- Not all of their ideas worked out as planned $\Rightarrow$ refined algorithm to what is known today as DPLL

DPLL insight

- There are two distinct ways to approach the boolean satisfiability problem:
- Search
  - Find satisfying assignment by searching through all possible assignments $\Rightarrow$ most basic incarnation: truth table!
- Deduction
  - Deduce new facts from set of known facts $\Rightarrow$ application of proof rules, semantic argument method
- DPLL combines search and deduction in a very effective way!

Deduction in DPLL

- Deductive principle underlying DPLL is propositional resolution
- Resolution can only be applied to formulas in CNF
- SAT solvers convert formulas to CNF to be able to perform resolution

Propositional Resolution

- Consider two clauses in CNF:

$$C_1 : \quad (l_1 \vee \ldots p \ldots \vee l_k) \qquad C_2 : \quad (l'_1 \vee \ldots \neg p \ldots \vee l'_n)$$

- From these, we can deduce a new clause C_3, called resolvent:

$$C_3 : \quad (l_1 \vee \ldots \vee l_k \vee l'_1 \vee \ldots\ldots \vee l'_n)$$

- Correctness:
  - Suppose p is assigned $\top$: Since C_2 must be satisfied and since $\neg p$ is $\bot$, $(l'_1 \vee \ldots\ldots \vee l'_n)$ must be true.
  - Suppose p is assigned $\bot$: Since C_1 must be satisfied and since p is $\bot$, $(l_1 \vee \ldots\ldots \vee l_k)$ must be true.
  - Thus, C_3 must be true.

Unit Resolution

- DPLL uses a restricted form of resolution, known as unit resolution.
- Unit resolution is propositional resolution, but one of the clauses must be a unit clause (i.e., contains only one literal)
- $C_1 : p \qquad C_2 : \quad (l_1 \vee \ldots \neg p \ldots \vee l_n)$
- Resolvent: $(l_1 \vee \ldots \vee l_n)$
- Performing unit resolution on C_1 and C_2 is same as replacing p with true in the original clauses.
- In DPLL, all possible applications of unit resolution called Boolean Constraint Propagation (BCP).

Boolean Constraint Propagation (BCP) Example

- Apply BCP to CNF formula:

$$(p) \wedge (\neg p \vee q) \wedge (r \vee \neg q \vee s)$$

- Resolvent of first and second clause:
- New formula:
- Apply unit resolution again:
- No more unit resolution possible, so this is the result of BCP.

Basic DPLL

```
bool DPLL(ϕ)
{
  1.  ϕ' = BCP(ϕ)
  2.  if(ϕ' = ⊤) then return SAT;
  3.  else if(ϕ' = ⊥) then return UNSAT;
  4.  p = choose_var(ϕ');
  5.  if(DPLL(ϕ'[p ↦ ⊤])) then return SAT;
  6.  else return (DPLL(ϕ'[p ↦ ⊥]));
}
```

- Recursive procedure; input is formula in CNF
- Formula is $\top$ if no more clauses left
- Formula becomes $\bot$ if we derive $\bot$ due to unit resolution

An Optimization: Pure Literal Propagation

- If variable p occurs only positively in the formula (i.e., no $\neg p$), p must be set to $\top$
- Similarly, if p occurs only negatively (i.e., only appears as $\neg p$), p must be set to $\bot$
- This is known as Pure Literal Propagation (PLP).

DPLL with Pure Literal Propagation

```
bool DPLL(φ)
{
   1. φ' = BCP(φ)
   2. φ'' = PLP(φ')
   3. if(φ'' = ⊤) then return SAT;
   4. else if(φ'' = ⊥) then return UNSAT;
   5. p = choose_var(φ'');
   6. if(DPLL(φ''[p ↦ ⊤])) then return SAT;
   7. else return (DPLL(φ''[p ↦ ⊥]));
}
```

Example

$$F : (\neg p \lor q \lor r) \land (\neg q \lor r) \land (\neg q \lor \neg r) \land (p \lor \neg q \lor \neg r)$$

- No BCP possible because no unit clause
- No PLP possible because there are no pure literals
- Choose variable q to branch on:

$$F[q \mapsto \top] : (r) \land (\neg r) \land (p \lor \neg r)$$

- Unit resolution using (r) and $(\neg r)$ deduces $\bot \Rightarrow$ backtrack

Example Cont.

$$F: \ (\neg p \ \vee \ q \ \vee \ r) \ \wedge \ (\neg q \ \vee \ r) \ \wedge \ (\neg q \ \vee \ \neg r) \ \wedge \ (p \ \vee \ \neg q \ \vee \ \neg r)$$

- Now, try $q = \bot$

$$F[q \mapsto \bot]: \ (\neg p \vee r)$$

- By PLP, set p to $\bot$ and r to $\top$
- $F[q \mapsto \bot, p \mapsto \bot, r \mapsto \top] : \top$
- Thus, F is satisfiable and the assignment $[q \mapsto \bot, p \mapsto \bot, r \mapsto \top]$ is a model of F.

Modern SAT Solvers

- Most solvers based on DPLL, but extend it in three important ways:
  1. Non-chronological backtracking
  2. Learning from past “mistakes”
  3. Heuristics for choosing variables and assignments
- Referred to as CDCL: conflict-driven clause learning

Non-Chronological Backtracking

- Recall basic DPLL: First try assigning p to $\top$; if doesn't work, backtrack to most recent decision level and try $p = \bot$
- Called chronological backtracking but often sub-optimal
- Suppose made assignments $p_1, p_2, \ldots p_{100}$ but discovered p_4 was a bad choice
- Backtracking to decision level associated with p_{100} is stupid...
- In non-chronological backtracking, can go back to earlier decision levels

Learning

- Learning = acquisition of new clauses to prevent similar bad assignments
- For instance, suppose we discover $p_5 = \top, p_{32} = \bot, p_{100} = \top$ is inconsistent, i.e.,

$$\phi \Rightarrow \neg(p_5 \land \neg p_{32} \land p_{100})$$

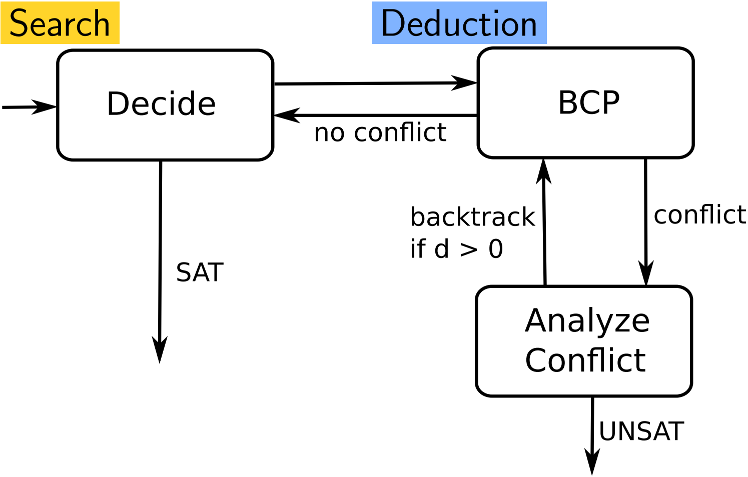
$$\phi \Rightarrow (\neg p_5 \lor p_{32} \lor \neg p_{100})$$

- Can add this clause without changing satisfiability
- Such clauses called conflict clauses $\Rightarrow$ SAT solver has database of conflict clauses

Decision Heuristics

- Basic DPLL chooses variables in random order
- But making assignment to certain variables can make formula much easier to solve!
- Modern solvers use more sophisticated heuristics
- This is something of a black art, but one of the most important elements in SAT solving ...

Architecture of DPLL-Based SAT Solvers



The Plan

- We will talk about BCP and AnalyzeConflict first (related)
- Then: common decision heuristics used in the Decide step
- Finally: Implementation tricks to make all this fast

BCP in SAT Solvers

- Recall: BCP is all possible applications of unit resolution
- SAT solvers remember deductions performed in the BCP process $\Rightarrow$ recorded as implication graph
- First some terminology . . .

Some Terminology and Conventions

- Decision variable: variable assigned in the Decide step
- The decision level of a decision variable is the level (order) in which it was assigned
- The decision level of a variable assigned due to BCP is the decision level of the last assigned decision variable
- Important note: Think of assignments as literals: Assignment $p = \top$ is literal p; assignment $p = \bot$ as literal $\neg p$
- Also: An assignment corresponds to a new unit clause added to our set of clauses

Decision Level Example

$$(\neg x_1 \lor x_2) \land (\neg x_3 \lor \neg x_4)$$

- Decide assigns $x_1 = \top \Rightarrow x_1$ decision var at level 1
- BCP yields:
- Decision level of x_2?
- Decide next assigns $x_4 = \top$. BCP deduces:
- x_4 decision variable with decision level:
- x_3's decision level:

Implication Graph

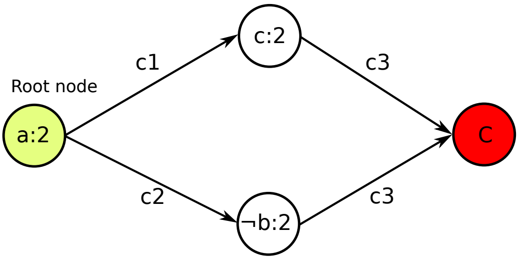
- An implication graph is a labeled directed acyclic graph
- Nodes: literals in the current partial assignment
- Node labels: Indicate assignment and decision level.
- Example: Node labeled $\neg x : 3$ (alternative notation $\neg x@3$) means variable x was assigned to $\perp$ at decision level 3
- Edges from $l_1, \ldots l_k$ to l labeled with c: Assignments $l_1, \ldots, l_k$ caused assignment l due to clause c during BCP
- A special node C is called the conflict node.
- Edge to conflict node labeled with c: current partial assignment contradicts clause c.

Implication Graph Example

- Consider the following set of clauses:

$$c_1 : (\neg a \vee c) \quad c_2 : (\neg a \vee \neg b) \quad c_3 : (\neg c \vee b)$$

- Assume *Decide* assigned $a = \top$ at decision level 2
- BCP yields:
- Assignment contradicts c_3!



Another Example

- Consider the following clauses:

  $c_1 : (\neg a \vee c) \;\; c_2 : (\neg c \vee \neg a \vee b) \;\; c_3 : (\neg c \vee d) \;\; c_4 : (\neg d \vee \neg b)$

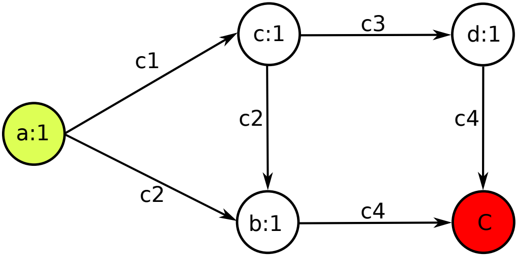
- Suppose *Decide* assigned $a = \top$ at decision level 1
- Using clause c_1, BCP yields:
- Using clause c_2, BCP yields:
- Using clause c_3, BCP yields:
- Assignment $b = \top, d = \top$ contradicts:
- Resulting implication graph?

Example cont.

- Consider the following clauses:

$$c_1 : (\neg a \vee c) \;\; c_2 : (\neg c \vee \neg a \vee b) \;\; c_3 : (\neg c \vee d) \;\; c_4 : (\neg d \vee \neg b)$$

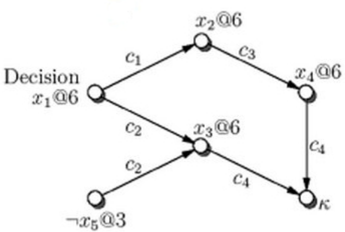
- Suppose *Decide* assigned $a = \top$ at decision level 1
- Resulting implication graph:



Implication Graph Properties

- Root nodes in the implication graph correspond to what kind of variables?
- Edges and internal nodes arise due to BCP
- If literal l has incoming edge labeled c, what do we know about c?
- If literal l has outgoing edge labeled c, what do we know about c?

Example



Based on this implication graph and ignoring variables decided in prior levels:

- What is c_4?
- What is c_3?
- What is c_1?
- What is c_2?

Analyzing Conflicts

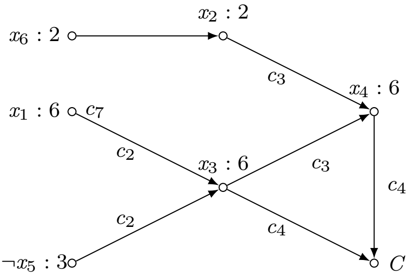
- We will use the implication graph to analyze conflicts
- AnalyzeConflict has two goals:

1. Learn new conflict clauses
2. Figure out what level to backtrack to

Conflict Clauses

- A conflict clause is a clause implied by the original formula
- Goal of conflict clauses: Prevent bad partial assignments by deriving contradiction as quickly as possible
- Question: To achieve this goal, are small or large conflict clauses better?
- Answer: Small ones because the smaller the clause, the quicker BCP forces variable assignments, and the quicker we derive contradictions!
- The implication graph is very useful for deriving small clauses implied by the original formula!

Conflicts and Learning



The roots of the graph $x_6 : 2$, $x_1 : 6$ and $\neg x_5 : 3$ constitute a sufficient condition for creating the conflict.

DPLL generates a conflict clause $c_9 = (\neg x_1 \lor x_5 \lor \neg x_6)$ and adds it to the clause database – the process is called learning

- c_9 logically implied by original formula
- addition sound, prunes the search space

Choosing Conflict Clauses

- One way to derive conflict clause: Conjoin all literals associated with root nodes reaching conflict node, use negation as conflict clause
- But there are other possibilities:
  - Assignment $\{x_2 \mapsto 1, x_3 \mapsto 1\}$ too leads to conflict
  - Hence, $(\neg x_2 \ \vee \ \neg x_3)$ possible candidate for learning too.

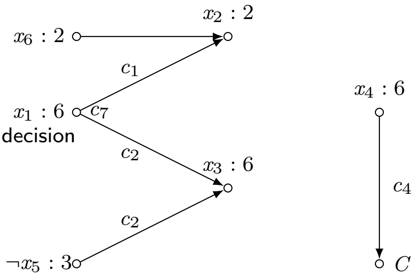
Choosing Conflict Clauses

- Another possibility to derive conflict clauses: Compute separating cut in the implication graph
- I.e. the set of edges whose removal breaks all paths from the root nods to C.
  - Two partitions:
    - reason side – includes all the roots
    - conflict side – conflict node C
  - Set of nodes in the reason side adjacent to the removed edges form a conflict clause

Choosing Conflict Clauses

- Another possibility to derive conflict clauses: Compute separating cut in the implication graph

- I.e. set of edges whose removal breaks all paths from the root nodes to C.
  - Set of nodes in the reason side adjacent to the removed edges form a conflict clause: $(\neg x_2 \vee \neg x_3)$



Backtracking

- Recall: AnalyzeConflict has two goals.
- First goal: Deriving conflict clauses ✓
- Second goal: Figure out what level to backtrack to
- Backtrack to level d means delete all variable assignments made after level d (but assignments at level d not deleted)
- Next: Talk about how to infer a good level to backtrack to

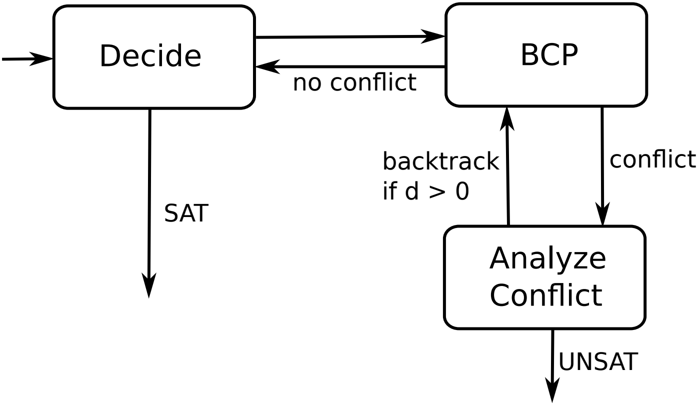
Backtracking and Asserting Clauses

- A good strategy: We want to backtrack to a level where BCP forces at least one assignment
- Asserting clause is a clause with exactly one literal at the last decision level, e.g., $(\neg x_2 \lor \neg x_3)$ in the last implication graph
- Asserting clauses can be found using unique implication points

Choosing Backtracking Level

- Question: Given an asserting clause, to what level should we backtrack?
- Answer:
- Since asserting clause contains only one literal, say l', from the highest decision level, backtracking to d will assert l'!

Recall: SAT Solver Architecture



- Decision heuristics for choosing variable order and truth assignment

Decision Heuristics

- Important part of SAT solvers, but something of a black art
- Can come up with hundreds of heuristics with varying tradeoffs
- We'll only talk about two:
  1. Dynamic Largest Individual Sum (DLIS)
  2. Variable State Independent Decaying Sum (VSIDS)

Dynamic Largest Individual Sum (DLIS)

- This heuristic chooses the literal that satisfies the largest number of currently unsatisfied clauses.
- A clause is unsatisfied if the clause does not evaluate to true under the current partial assignment.
- Example: $(x_1 \lor \neg x_2) \land (\neg x_2 \lor x_3) \land (\neg x_1 \lor x_2 \lor \neg x_3)$
- What assignment would DLIS pick for this formula? (assuming no assignments so far)
- How is this heuristic dynamic?
- Thus, overhead can be high and must be implemented carefully to minimize bookkeeping

Variable State Independent Decaying Sum (VSIDS)

- Similar to DLIS, but tries to reduce overhead and favor literals involved in conflicts (i.e. conflict-driven)
- Count number of clauses in which the literal appears, but disregard if the clause it appears in is satisfied or not
- Specifically, initialize the score of each literal to the number of clauses in which literal appears
- Every time we add a conflict clause involving literal l, increase the score of that literal by 1
- Periodically divide scores of all literals by 2
  $\Rightarrow$ decaying sum

Variable State Independent Decaying Sum (VSIDS), cont.

- Favors literals involved in conflicts
- If a literal doesn't appear in a recent conflict, its score will decay over time
- Much cheaper compared to DLIS because we don't need to scan all clauses to figure out which ones are satisfied
- Introduced in the CHAFF SAT solver from Princeton, written by undergrads!

Implementation Tricks

- To build competitive SAT solvers, it is important to minimize overhead of implementing Decide, BCP, and Analyze Conflict
- Very important because SAT solver might be searching through hundreds of thousands of assignments!
- We'll talk about two issues:
    1. number of conflict clauses
    2. trick to perform BCP fast: watch literals

Conflict Clauses

- Recall: After analyzing conflict, we add new conflict clause to our clause database
- Pro: Conflict clauses quickly block bad assignments and prevent future mistakes
- Con: More clauses = more overhead

$\Rightarrow$ Tradeoff between conflict prevention and minimizing overhead

Conflict Clauses, cont.

- For this reason, many SAT solvers do not keep all the conflict clauses they derive
- For example, they put a limit on the number of conflict clauses they derive
- Typically, keep most recent conflict clauses since they are most relevant to current part of search space
- Can guarantee termination of algorithm even if we do not keep all conflict clauses!

Implementing BCP

- Implementing BCP efficiently is very important because SAT solvers spend a lot of time doing BCP
- Naive implementation of BCP: Requires scanning all currently unsatisfied clauses
- But industrial SAT contain hundreds of thousands of clauses, so scanning all unsatisfied clauses too expensive!
- A more intelligent implementation: Keep mapping from each literal to all clauses in which each literal appears
- But this is still very expensive because typically each literals appears in many clauses

The Trick: Watch Literals

- Modern SAT solvers use a more clever trick to perform BCP fast: watch literals
- Observe: Ultimate purpose of BCP is to figure out which variable assignments imply which others
- Question: If we are performing unit resolution between l and clause $c = (\neg l \vee l_1, \ldots \vee l_k)$, under what condition will a new assignment be implied?
- Answer:
- Idea: Since a clause will not imply new variable assignment unless it has only two literals left, we only need to look at clauses that have at most two unassigned literals!

Watch Literals

- To efficiently detect clauses with at most two unassigned literals, select two unassigned literals in each unsatisfied clause as watch literals

- Invariant: Watch literals are always unassigned!

- To maintain invariant: If a watch literal is assigned a truth value and clause has other unassigned literals, choose any unassigned literal in clause to be new watch literal

- If a watch literal is assigned a truth value and there are no other unassigned non-watch literals left, BCP implies an assignment to the only remaining watch literal!

Watch Literals, cont.

- Question: Given this invariant, if we make assignment l, which clauses can imply new variable assignments?
- Answer:
  - If $\neg l$ does not appear, we can't perform unit resolution
  - If $\neg l$ appears but is not a watch literal, then clause has more than two unassigned literals $\Rightarrow$ won't imply new assignment!
- Watch literal trick makes BCP much faster because much fewer clauses contain negation of current literal as a watch literal!
- Yields huge improvement in SAT solver performance!

Practical SAT Solving Summary

- Most competitive solvers today are based on DPLL
- But they extend DPLL in three ways: non-chronological backtracking, conflict clause learning, decision heuristics, engineering tricks (watch literals)
- Referred to as CDCL: conflict-driven clause learning
- Most competitive SAT solvers based on CDCL
- But there are also other kinds of SAT solvers not based on CDCL, for instance, perform stochastic search (e.g., WalkSAT)
- Stochastic SAT solvers perform well on randomly-generated SAT instances, but not so well on industrial ones