SAT and SMT Solver Basics

Scott West

November 9th, 2011

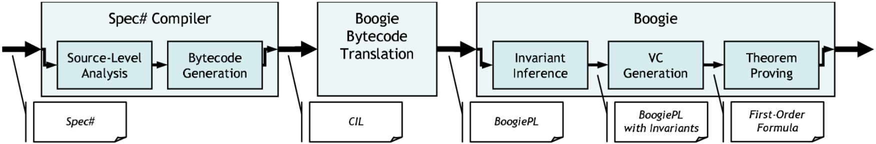
SAT and SMT

- SAT – Satisfiability
- SMT – Satisfiability Modulo Theories, pairing SAT solvers with domain theories.
- Techniques are implemented in
  - SAT solvers, such as MiniSAT, cryptominisat and pingeling.
  - SMT solvers, such as Z3, Yices and CVC3

Who uses SAT and SMT?

SAT and SMT solvers are used in various tools:

- Planning, and AI related activities, demonic testing.
- Bounded model checking.
- Proof assistants, such as Isabelle or Coq.
- Automated proof tools, such as Spec# or Dafny.
- Verification frameworks, such as Boogie or Why.



Boolean formulae

SAT sovlers determine satisfiability for clauses in CNF (conjunctive normal form). For example, we may have some formula

$\phi = (a \vee \neg b) \wedge (b \vee \neg c \vee \neg a)$.

SAT solvers also produce an assignment of variables that will satisfy the formula. For ϕ, above, such an assignment could be $\sigma = \{a, \neg c\}$.

Basic Algorithm

Definition

Given a partial assignment σ and a formula ϕ, if the assignment is enough to conclude either ϕ is true or false, then the algorithm terminates with that judgement.

If σ doesn't have enough information yet, an unassigned variable, l, is chosen and the process is repeated; with both $\sigma' = \sigma \cup \{l\}$ and $\sigma' = \sigma \cup \{\neg l\}$.

This is what we will call $SAT_{basic}(\sigma, \phi)$.

The basic algorithm is an unintelligent state exploration, this is not used in practice.

Basic DPLL Algorithm

Definition

The DPLL (Davis, Putnam, Logemann, Loveland) algorithm extends the basic algorithm by adding boolean constraint propagation. Boolean constraint propagation, $BCP(\sigma, \phi)$, performs resolution on ϕ, propagating the effects of "necessary" assignments.

Example

In $a \wedge (\neg a \vee b)$, we first notice that a must be in σ. Propagating this fact forces b to also be in σ.

Although pure literal assignment (assigning a variable that only ever is seen in ϕ with a single polarity) is a part of DPLL, it is often done as a pre-processing step.

DPLL Analysis

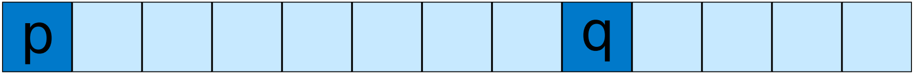
The DPLL family of algorithms improve over the basic algorithm by:

- Affirming facts that *must* be true, as a first step.
- Using logical consequence (BCP) to avoid making more decisions than necessary. A decision is when we choose an unassigned literal l to add to the state σ.

Main components of DPLL

- Picking which variable to make true or false.
- Quickly finding the consequences of such a choice (BCP).
- How to recover from an incorrect decision.

Backtracking

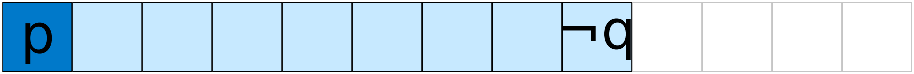


After making decision q, we perform BCP, which leads to a conflict: choosing q has made us to assert l and $\neg l$.

Clearly q was the wrong decision!

Backtracking doesn't extract any information from the conflicts it resolves!

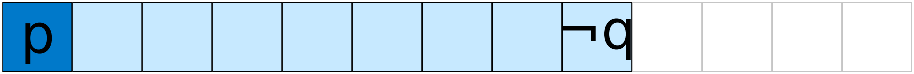
Backtracking



All literals in the decision level of q are rolled-back, and $\neg q$ is added as a consequence of the previous decision level.

Backtracking doesn't extract any information from the conflicts it resolves!

Backtracking



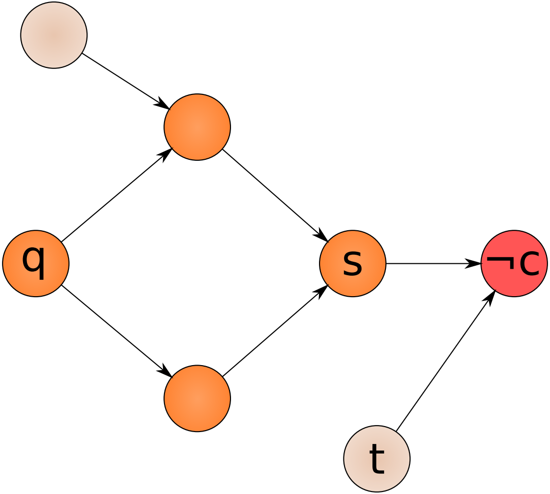
All literals in the decision level of q are rolled-back, and $\neg q$ is added as a consequence of the previous decision level.

Backtracking doesn't extract any information from the conflicts it resolves!

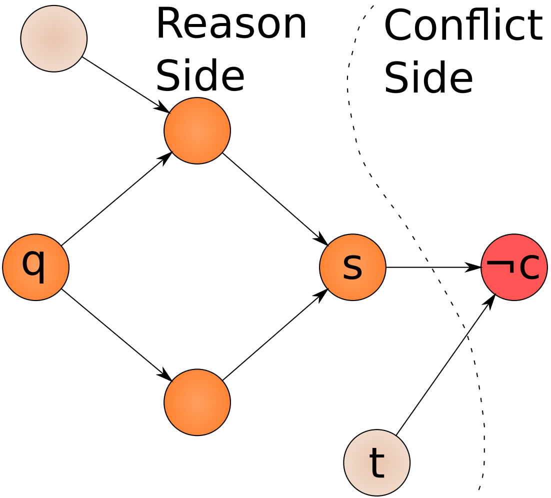
Backjumping

- Backjumping is a smarter form of backtracking.
- Takes into account causes of conflicts.

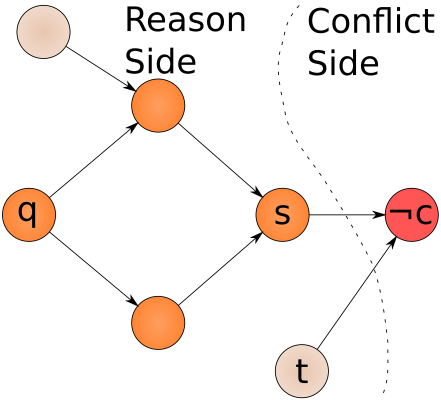
Backjumping Example



Backjumping Example



Backjumping Example



Examining the cut:

- If $s \wedge t$ is true, then it leads to a conflict.
- We can *learn* $\neg s \vee \neg t$ from this, adding it as a clause.
- We *jump* back to the level where t was introduced.
- There are many options for where to make the cut.

Other optimizations

Literal selection

Variable state independent decaying sum (VSIDS) aims to make decisions from recently used facts. Literals are given scores, and the scores increase when the literals are seen in a conflict. The scores periodically are cut in half, this prefers more recently used literals.

Constraint propagation

Two watched literals eliminate much of the time needed to find clauses to perform BCP on. Literals such as p or q are mapped to the clauses in which they appear. When they are made true or false, they are looked up and their clauses examined for propagation.

Random Restarts

Random restarts drop the accumulated assignments, ie, the input arguments (σ, ϕ) become $(\emptyset, \phi)$.

In conjunction with backjumping and clause-learning, this allows "bad" assignments to be discarded while retaining the experience from the work performed so far.

DPLL Implementation Details

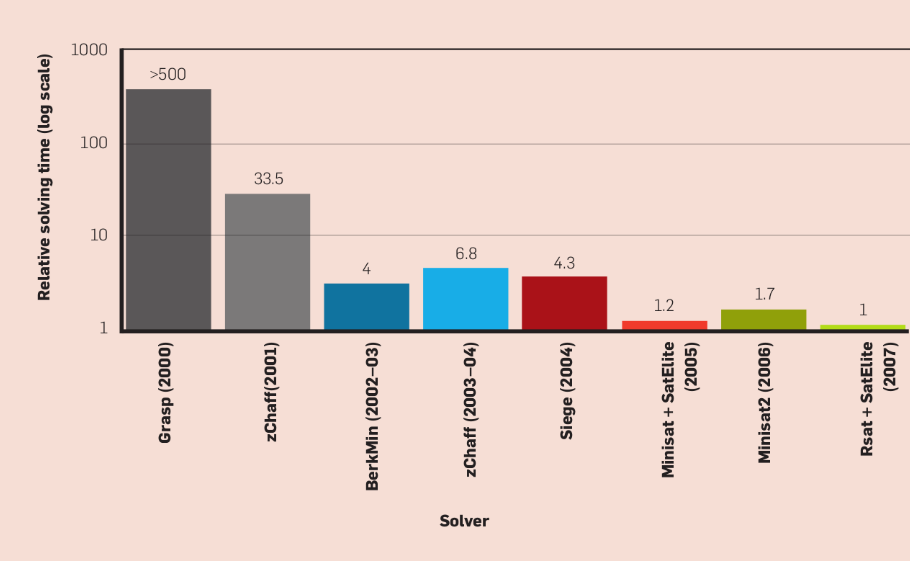
- The literals added to the state are segmented into two types, regular (or consequence) and decision literals.
- The state (σ) in a real SAT solver is (likely) organized into decision levels, corresponding to decisions literals. Each decision level has a series of consequence literals generated by BCP, associated to a decision literal.
- When a "wrong" decision is made, the current decision level is rolled-back, and the negation of that decision is added as a consequence.

Speed concerns

While theoretical exponential-upper-bounds on solving times are known $O(1.321^n)$ (Hertli, Moser, Scheder), real-world solvers use many heuristic techniques (random restarts, VSIDS) to get good results in practice.

SAT solvers are several orders of magnitude faster today than they were 10 years ago.

Speedup over time



SAT solver anecdotes

“To be really good at writing a SAT solver, you have to have a sort of intuition (like the force).” – Mate Soos, author of CryptoMiniSat

“If you want people to work on optimizing a tool, turn it into a competition.” – SAT-COMP organizers

What's next?

Okay, now we can efficiently solve SAT problems, what next?

What's next?

Satisfiability Modulo Theories!

What does SMT give us?

- More expressive language: beyond conjunctions and disjunctions of literals!
- Lowers the burden to entry, no longer is it a requirement to develop a clever translation from your problem to CNF.
- Things like arrays, quantifiers, arithmetic, etc, can now be used, given that there's a corresponding theory.

SMT Solver Styles

There are two main “styles” of SMT solvers for some theory T:

- Eager: convert the formulae into CNF form and feed it directly to a SAT solver.
- Lazy: preserve the formula and interactively communicate back-and-forth with a T-solver.

A T-solver can solve conjunctions of T-terms.

Eager SMT

Eager SMT is able to take advantage of the latest SAT solvers by converting to a portable SAT solver format, such as DIMACS. Any number of solvers can work on the instance in such a case.

However, very often the cost to convert an entire formula from the SMT domain to SAT can be very computationally expensive.

Basic Lazy SMT

1. Replace theory-specific literals with placeholder variables.
2. If the SAT solver finds the formula inconsistent with placeholder variables, it is also T-inconsistent.
3. If the theory is consistent, then there is an assignment given for the placeholder variables.
4. The assignment is given back to the T-solver and it either confirms the assignment or provides a counter-example clause that can be learned by the SAT-solver.

Lazy SMT example

Suppose we want to prove the formula $\neg(f(t) = f(s)) \wedge t = s$, in the domain of uninterpreted functions.

1. For theory-specific literal replacement (here, equality, non-boolean terms, and functions are theory-specific): we take:
   - $a = (f(t) = f(s))$
   - $b = (t = s)$,

   translating the original to $\neg a \wedge b$
2. $SAT(\emptyset, \neg a \wedge b)$ easily gives us $\sigma = \{\neg a, b\}$.
3. We substitute a and b for their original definitions and give them to the T-solver. It gives the counter-example clause $a \vee \neg b$.
4. Finally, we have a new clause for the SAT-solver, $\neg a \wedge b \wedge (a \vee \neg b)$, which it decides is inconsistent. Since it is unsatisfiable, the original formula is also T-inconsistent.

Lazy SMT extensions

Incremental

Do not wait for an entire assignment to be built before checking it with the T-solver. Have the T-solver run incrementally for each new assignment.

Online

If T-inconsistency is found incrementally, backjump using this T-solver provided information.

Theory Propagation

Use the T-solver to actively affirm true literals in the formula. This is active as opposed to the reactive techniques above.

Demo