SAT and SMT Solving

Sarah Winkler

SS 2018

Department of Computer Science
University of Innsbruck

Outline

- Summary of Last Week
- Conflict Analysis
- Conflict Driven Clause Learning
- Bit Vectors

Summary of Last Week

Approach

- most state-of-the-art SAT solvers use variation of Davis - Putnam - Logemann - Loveland (DPLL) procedure (1962)
- DPLL is sound and complete backtracking-based search algorithm
- can be described abstractly by transition system (Nieuwenhuis, Oliveras, Tinelli 2006)

Definition (Abstract DPLL)

- decision literal is annotated literal l^d
- state is pair $M \parallel F$ for
  - list M of (decision) literals
  - formula F in CNF
- transition rules

$$M \parallel F \quad \Longrightarrow \quad M' \parallel F' \quad \text{or} \quad \text{FailState}$$

Definition (DPLL Transition Rules)

- unit propagation $M \parallel F,\ C \vee l \implies M\ l \parallel F,\ C \vee l$
  if $M \models \neg C$ and l is undefined in M
- pure literal $M \parallel F \implies M\ l \parallel F$
  if l occurs in F but l^c does not occur in F, and l is undefined in M
- decide $M \parallel F \implies M\ l^d \parallel F$
  if l or l^c occurs in F, and l is undefined in M
- backtrack $M\ l^d\ N \parallel F, C \implies M\ l^c \parallel F, C$
  if $M\ l^d\ N \models \neg C$ and N contains no decision literals
- fail $M \parallel F, C \implies$ FailState
  if $M \models \neg C$ and M contains no decision literals
- backjump $M\ l^d\ N \parallel F, C \implies M\ l' \parallel F, C$
  if $M\ l^d\ N \models \neg C$ and $\exists$ clause $C' \vee l'$ such that
  - $F, C \models C' \vee l'$ backjump clause
  - $M \models \neg C'$ and l' is undefined in M, and l' or l'^c occurs in F or in $M\ l^d\ N$

Definition

basic DPLL $\mathcal{B}$ consists of unit propagation, decide, fail, and backjump

Theorem (Termination)

there are no infinite derivations $\parallel F \implies_{\mathcal{B}} S_1 \implies_{\mathcal{B}} S_2 \implies_{\mathcal{B}} \dots$

Theorem (Correctness)

for derivation with final state S_n*:*

$$\parallel F \implies_{\mathcal{B}} S_1 \implies_{\mathcal{B}} S_2 \implies_{\mathcal{B}} \dots \implies_{\mathcal{B}} S_n$$

- *if* $S_n = \text{FailState}$ *then* F *is unsatisfiable*
- *if* $S_n = M \parallel F'$ *then* F *is satisfiable and* $M \vDash F$

Conflict Analysis

Outline

- Summary of Last Week
- Conflict Analysis
- Conflict Driven Clause Learning
- Bit Vectors

Backjump: Idea

- backjump clause $C' \vee l'$ is entailed by formula (magically detected)
- prefix M of current literal list entails $\neg C'$, so l' must be true

Backjump to Definition

- backjump $\quad M\, l^d\, N \parallel F, C \implies M\, l' \parallel F, C$
  if $M\, l^d\, N \vDash \neg C$ and $\exists$ clause $C' \vee l'$ such that
  - $F, C \vDash C' \vee l'$ — backjump clause
  - $M \vDash \neg C'$ and l' is undefined in M, and l' or l'^c occurs in F or in $M\, l^d\, N$

Example

$$1^d\, 2\, 3^d\, 4^d\, \overline{5} \parallel \overline{1} \vee 2,\ \overline{1} \vee \overline{3} \vee 4 \vee 5,\ \overline{2} \vee \overline{4} \vee \overline{5},\ 4 \vee \overline{5},\ \overline{4} \vee 5,\ \overline{1} \vee \overline{5} \vee 6,\ \overline{2} \vee \overline{5} \vee \overline{6}$$

$$\implies 1^d\, 2\, \overline{5} \parallel \overline{1} \vee 2,\ \overline{1} \vee \overline{3} \vee 4 \vee 5,\ \overline{2} \vee \overline{4} \vee \overline{5},\ 4 \vee \overline{5},\ \overline{4} \vee 5,\ \overline{1} \vee \overline{5} \vee 6,\ \overline{2} \vee \overline{5} \vee \overline{6}$$

$M = 1^d\, 2 \qquad l = 3 \qquad N = 4^d\, \overline{5} \qquad C = \overline{4} \vee 5 \qquad C' = \overline{1} \qquad l' = \overline{5}$

- $1^d\, 2\, 3^d\, 4^d\, \overline{5} \vDash \neg(\overline{4} \vee 5)$
- backjump clause $C' \vee l' = \overline{1} \vee \overline{5}$ satisfies $\varphi \vDash C' \vee l'$
- $1^d\, 2 \vDash 1$ and 5 is undefined in $1^d\, 2$ but occurs in formula

Outline

- Summary of Last Week
- Conflict Analysis
- Conflict Driven Clause Learning
- Bit Vectors

Desirable Properties of Backjump Clauses

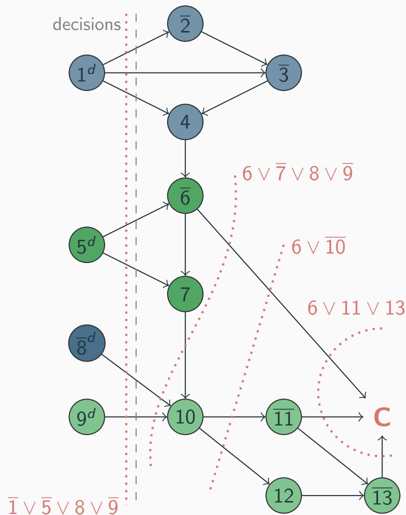
- small
- should trigger progress

How to Determine Backjump Clauses?

- implication graph
- resolution

Example: Implication Graph

$$\varphi = (\overline{1} \vee \overline{2}) \wedge (\overline{1} \vee 2 \vee \overline{3}) \wedge (\overline{1} \vee 3 \vee 4) \wedge (\overline{4} \vee \overline{5} \vee \overline{6}) \wedge (\overline{5} \vee 6 \vee 7) \wedge$$
$$(\overline{7} \vee 8 \vee \overline{9} \vee 10) \wedge (\overline{10} \vee \overline{11}) \wedge (\overline{10} \vee 12) \wedge (\overline{12} \vee \overline{13}) \wedge (6 \vee 11 \vee 13)$$



level	literal	reason
1	1	decision
	$\overline{2}$	$\overline{1} \vee \overline{2}$
	$\overline{3}$	$\overline{1} \vee 2 \vee \overline{3}$
	4	$\overline{1} \vee 3 \vee 4$
2	5	decision
	$\overline{6}$	$\overline{4} \vee \overline{5} \vee \overline{6}$
	7	$\overline{5} \vee 6 \vee 7$
3	$\overline{8}$	decision
4	9	decision
	10	$\overline{7} \vee 8 \vee \overline{9} \vee 10$
	$\overline{11}$	$\overline{10} \vee \overline{11}$
	12	$\overline{10} \vee 12$
	$\overline{13}$	$\overline{12} \vee \overline{13}$

next

What to Learn from That?

Definitions

- cut of implication graph has at least all decision literals on the left, and at least the conflict node on the right
- literal l in implication graph is unique implication point (UIP) if all paths from last decision literal to conflict node go through l
- first UIP is UIP closest to conflict node

Key Observations

- if $l_1 \to l_1', \dots, l_k \to l_k'$ are cut edges then $l_1^c \vee \cdots \vee l_k^c$ is entailed clause
- last decision literal is UIP

Example

- $\overline{1} \vee \overline{5} \vee 8 \vee \overline{9}$
- $6 \vee 11 \vee 13$
- $6 \vee \overline{10}$
- $6 \vee \overline{7} \vee 8 \vee \overline{9}$

Definition (Implication Graph)

Consider DPLL derivation to $\| F \quad \Longrightarrow^*_{\mathcal{B}} M \| F'$.

Implication graph is a directed acyclic graph constructed as follows:

- add node labelled l for … M
- repeat until there is no change:

  or former backjump clause

  if $\exists$ clause $l_1 \vee \dots l_m \vee l'$ in F' such that there are already nodes $l_1^c, \dots, l_m^c$
  - add node l' if not yet present
  - add edges $l_i^c \to l'$ for all $1 \leqslant i \leqslant m$ if not yet present
- if $\exists$ clause $l_1' \vee \cdots \vee l_k'$ in F' such that there are nodes $l_1'^c, \dots, l_k'^c$
  - add conflict node labeled C
  - add edges $l_i'^c \to C$

potential backjump clause

Lemma

if edges intersected by cut are $l_1 \to l_1', \dots, l_k \to l_k'$ *then* $F' \vDash l_1^c \vee l_k^c$

Remarks

- keeping track of implication graph is too expensive in practice
- compute clauses associated with cuts by resolution instead

Definition (Resolution)

$$\frac{C \vee l \qquad C' \vee \neg l}{C \vee C'}$$

(assuming literals in clauses can be reordered)

Example

$$\frac{6 \vee 11 \vee 13 \qquad \overline{12} \vee \overline{13}}{6 \vee 11 \vee \overline{12}}$$

How to Derive Backjump Clause by Resolution

- set C_0 to conflict clause
- let l be last assigned literal such that l^c is in C_0
- while l is no decision literal:
  - C_{i+1} is resolvent of C_i and clause D that led to assignment of l
  - let l be last assigned literal such that l^c is in C_{i+1}

Observation

every C_i corresponds to cut in implication graph

Example

- $C_0 = 6 \vee 11 \vee 13$
- $C_1 = 6 \vee 11 \vee \overline{12}$
- $C_2 = 6 \vee 11 \vee \overline{10}$
- $C_3 = 6 \vee \overline{10}$
- $C_4 = 6 \vee \overline{7} \vee 8 \vee \overline{9}$

$$\frac{6 \vee 11 \vee 13 \qquad \overline{12} \vee \overline{13}}{6 \vee 11 \vee \overline{12}} \qquad \overline{10} \vee 12$$

$$\frac{10}{}$$

Conflict Driven Clause Learning

Observations

- adding backjump clause to formula helps to prune search space (learning)
- if progress is too slow according to some measure, restart procedure:
  due to learned clauses heuristics will lead to different proof search

Definition (DPLL with Learning and Restarts)

DPLL with learning and restarts $\mathcal{R}$ extends system $\mathcal{B}$ by following three rules:

- learn $\quad M \parallel F \implies M \parallel F,\, C$
  if $F \vDash C$ and all atoms of C occur in M or F
- forget $\quad M \parallel F,\, C \implies M \parallel F$
  if $F \vDash C$
- restart $\quad M \parallel F \implies \parallel F$

Theorem (Termination)

any derivation $\| F \Longrightarrow_{\mathcal{R}} S_1 \Longrightarrow_{\mathcal{R}} S_2 \Longrightarrow_{\mathcal{R}} \dots$ *is finite if*

- *it contains no infinite subderivation of learn and forget steps, and*
- *restart is applied with increasing periodicity*

Theorem (Correctness)

for derivation with final state S_n*:*

$$\| F \Longrightarrow_{\mathcal{R}} S_1 \Longrightarrow_{\mathcal{R}} S_2 \Longrightarrow_{\mathcal{R}} \dots \Longrightarrow_{\mathcal{R}} S_n$$

- *if* $S_n = \text{FailState}$ *then* F *is unsatisfiable*
- *if* $S_n = M \| F'$ *then* F *is satisfiable and* $M \models F$

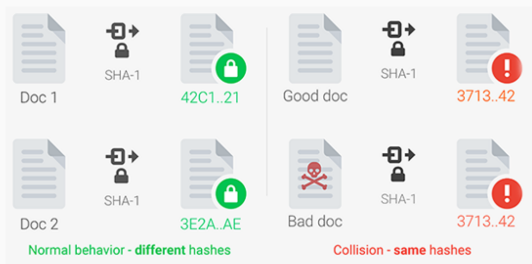
Bit Vectors

Outline

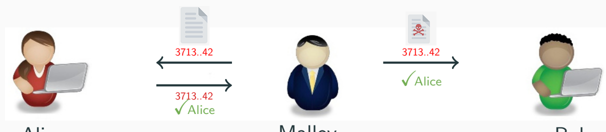
- Summary of Last Week
- Conflict Analysis
- Conflict Driven Clause Learning
- Bit Vectors

Cryptographic Hash Functions

- cryptographic hash function f is one-way hash function (SHA-1, MD5, ...)
- considered infeasible to invert, and to find messages with same hash
- problem: hash collisions



Collision Attack Scenario



More Cryptanalysis using SAT

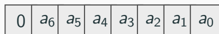
- collision attacks (preimage attacks) for current hash functions such as MD4, MD5, SHA-256, CryptoHash, Keccak, ...
- exhibit classes of weak keys (or prove their absence) for block ciphers such as IDEA, WIDEA-*n*, or MESH-8
- solve inversion problems, e.g. for 20 bit DES key
- reason about crypto primitives
- help prove complexity bounds of certain operations

Tools for SAT/SMT-Based Cryptanalysis

- CryptoMiniSat
- CryptoSMT
- Transalg
- ...

Bit Blasting

Representing Characters as Bit Vectors

- ASCII character a is representable as bit vector for `char` of 8 bits: using propositional variables $a_0, \ldots, a_6$ write a as 

- a_0 is least significant bit, so e.g. $\mathtt{C} = \mathtt{ascii}(67)$ is

Ingredients of (Cryptographic) Hash Functions

- hash functions operate on character strings
- string of length n representable as bit vector of n `chars` ($8n$ bits)
- common operations in hash functions are
  - bitwise XOR $\oplus$
  - addition $+$
  - shifts

Example (Rotation Hash)

```
def rotation_hash(s):
  h = 0
  for i in range(0,len(s)):
    h = (h << 4) ^ (h >> 28) ^ ord(s[i])
  return h
```

Z3

common open source SAT/SMT solver

`z3`: Python interface to Z3

- from `https://github.com/Z3Prover/z3` or via `pip install z3`
- API: `https://z3prover.github.io/api/html/namespacez3py.html`

Building Formulas

- `True`, `False` — boolean constants
- `Bool(`*name*`)` — propositional variable named *name* (calling `Bool(`*name*`)` twice yields same variable)
- `FreshBool(`*pre*`)` — new propositional variable with name prefix *name* (calling `FreshBool(`*pre*`)` twice does not yield same variable!)
- `And(`$a_1, \dots, a_n$`)` — conjunction with arbitrarily many arguments
- `Or(`$a_1, \dots, a_n$`)` — disjunction with arbitrarily many arguments
- `Not(`a`)` — negation
- `Implies(`a, b`)` — implication
- `Xor(`a, b`)` — exclusive or

Solving Formulas

- `Solver()` create new solver object
- `Solver.add(`$\varphi_1, \dots, \varphi_n$`)` require constraints $\varphi_1, \dots, \varphi_n$ to be true
- `Solver.check()` check for satisfiability
- `Solver.model()` returns valuation (after successful call of `check`)

Moreover ...

- `simplify(`φ`)` simplifies formula φ
- `Solver.statistics()` is map of solving statistics

Example

```
from z3 import *
p = Bool('p') # create variable named 'p'
foo1 = FreshBool('foo') # create new variables prefixed 'foo'
foo2 = FreshBool('foo')

phi = Or(p, p, And(foo2, Xor(foo1, Not(foo1)), True), False)
print(phi) # Or(p, p, And(foo!1, Xor(foo!0, Not(foo!0)), True), False)
psi = simplify(phi)
print(psi) # Or(p, foo!1)

solver = Solver()
solver.add(psi) # assert that psi should be true
solver.add(Implies(foo1,p), Or(foo1, foo2)) # assert something else

print solver # [Or(p, foo!1), Implies(foo!0, p), Or(foo!0, foo!1)]
result = solver.check() # check for satisfiability

if result:
  model = solver.model() # get valuation
  print model[p], model[foo1], model[foo2] # False False True
```