

SAT and SMT Solving

Sarah Winkler

Computational Logic Group
Department of Computer Science
University of Innsbruck

lecture 10
SS 2019

Outline

- Summary of Last Week
- Bit Vectors

Bounds for Satisfiability

Aim: Find Bound B

$A\vec{x} \leq \vec{b}$ satisfiable $\iff (A\vec{x} \leq \vec{b} \quad \text{and} \quad \forall i. -B \leq \vec{x}_i \leq B)$ satisfiable

Approach

- 1 represent $\{\vec{x} \mid A\vec{x} \leq \vec{b}\}$ as $\text{hull}(X) + \text{cone}(V)$
 - ▶ increase dimension by 1: represent $\{\vec{x} \mid A\vec{x} \leq \vec{b}\}$ by $\{\vec{z} \mid A\vec{z} \leq \vec{0}\}$
 - ▶ use FMW to represent $\{\vec{z} \mid A\vec{z} \leq \vec{0}\}$ as $\text{cone}(V)$
- 2 derive bound B for hull + cone representation

Theorem (Farkas, Minkowski, Weyl)

A cone C is polyhedral iff it is finitely generated

Corollary (for 2)

If $|c| \leq b$ for all coefficients c of vectors in $X \cup V$ then for $B := b \cdot (1 + n)$ have
 $(\text{hull}(X) + \text{cone}(V)) \cap \mathbb{Z}^n = \emptyset \iff (\text{hull}(X) + \text{cone}(V)) \cap \{-B, \dots, B\}^n = \emptyset$

Theorem

$B = (\max(\{1\} \cup \{\|v\| \mid v \text{ is row vector of } A\}))^n$

Gomory Cuts

Definition (Cut)

given solution α to problem over \mathbb{R}^n , cut is inequality $a_1x_1 + \dots + a_nx_n \leq b$ which is not satisfied by α but by every \mathbb{Z}^n -solution

Gomory Cuts: Assumptions

- ▶ DPLL(T) Simplex returned solution α and final tableau A such that

$$A\vec{x}_N = \vec{x}_B \qquad -\infty \leq l_i \leq x_i \leq u_i \leq +\infty$$

- ▶ for some $i \in B$ have $\alpha(x_i) \notin \mathbb{Z}$ and for all $j \in N$ value $\alpha(x_j)$ is l_j or u_j

Notation

- ▶ write $c = \alpha(x_i) - \lfloor \alpha(x_i) \rfloor$
- ▶ $L^+ = \{j \in L \mid \alpha(x_j) = l_j \text{ and } A_{ij} \geq 0\}$ $U^+ = \{j \in U \mid \alpha(x_j) = u_j \text{ and } A_{ij} \geq 0\}$
 $L^- = \{j \in L \mid \alpha(x_j) = l_j \text{ and } A_{ij} < 0\}$ $U^- = \{j \in U \mid \alpha(x_j) = u_j \text{ and } A_{ij} < 0\}$

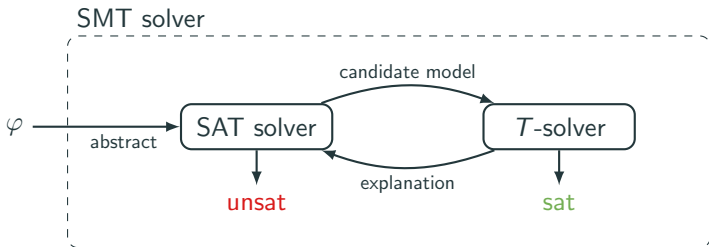
Lemma (Gomory Cut)

$$\sum_{j \in L^+} \frac{A_{ij}}{1-c} (x_j - l_j) - \sum_{j \in U^-} \frac{A_{ij}}{1-c} (u_j - x_j) - \sum_{j \in L^-} \frac{A_{ij}}{c} (x_j - l_j) + \sum_{j \in U^+} \frac{A_{ij}}{c} (u_j - x_j) \geq 1$$

Outline

- Summary of Last Week
- Bit Vectors

How to Be Lazy



Theory T

- ▶ equality logic
- ▶ equality + uninterpreted functions (EUF)
- ▶ linear real arithmetic (LRA)
- ▶ linear integer arithmetic (LIA)
- ▶ bitvectors (BV)
- ▶ arrays (A)

T -solving method

- equality graphs ✓
- congruence closure ✓
- DPLL(T) Simplex ✓
- DPLL(T) Simplex + cuts ✓
- bit-blasting

Definition (Bit Vector Theory)

- ▶ **variable** \mathbf{x}_k is list of length k of propositional variables $x_{k-1} \dots x_2 x_1 x_0$
- ▶ **constant** n_k is bit list of length k
- ▶ formulas built according to grammar

$formula := (formula \vee formula) \mid (formula \wedge formula) \mid (\neg formula) \mid atom$

$atom := term \ rel \ term \mid true \mid false$

$rel := = \mid \neq \mid \geq_u \mid \geq_s \mid >_u \mid >_s$

$term := (term \ binop \ term) \mid (unop \ term) \mid var \mid constant \mid term[i:j] \mid$
 $(formula \ ? \ term : \ term)$

$binop := + \mid - \mid \times \mid \div_u \mid \div_s \mid \%_u \mid \%_s \mid \ll \mid \gg_u \mid \gg_s \mid \& \mid \mid \mid \wedge \mid ::$

$unop := \sim \mid -$

- ▶ **axioms** are equality axioms plus rules for arithmetic/comparison/bitwise operations on bit vectors of length k
- ▶ **solution** assigns bit list of length k to variables \mathbf{x}_k

Examples

▶ $\mathbf{x}_4 + \mathbf{y}_4 = \mathbf{7}_4$

satisfiable: $v(\mathbf{x}_4) = \mathbf{4}_4$ and $v(\mathbf{y}_4) = \mathbf{3}_4$

▶ $\mathbf{x}_4 + \mathbf{2}_4 <_u \mathbf{x}_4$

satisfiable: $v(\mathbf{x}_4) = \mathbf{15}_4$

overflow semantics!

▶ $(\mathbf{x}_4 \times \mathbf{y}_4 = \mathbf{6}_4) \wedge (\mathbf{x}_4 \& \mathbf{y}_4 = \mathbf{2}_4)$

satisfiable: $v(\mathbf{x}_4) = \mathbf{3}_4$, $v(\mathbf{y}_4) = \mathbf{2}_4$

▶ $(\mathbf{x}_4 \geqslant_u \mathbf{y}_4) \wedge \neg(\mathbf{x}_4 \geqslant_s \mathbf{y}_4)$

satisfiable: $v(\mathbf{x}_4) = \mathbf{8}_4$, $v(\mathbf{y}_4) = \mathbf{0}_4$

▶ $(\mathbf{x}_4 \lll \mathbf{2}_4 = \mathbf{12}_4) \wedge (\mathbf{x}_4 + \mathbf{1}_4 = \mathbf{12}_4)$

satisfiable: $v(\mathbf{x}_4) = \mathbf{11}_4$

▶ $(\mathbf{8}_4 \ggg_u \mathbf{2}_4 = \mathbf{2}_4) \wedge (\mathbf{8}_4 \ggg_s \mathbf{2}_4 = \mathbf{14}_4)$

holds

▶ $(\mathbf{x}_4[1:0] :: \mathbf{x}_4[3:2] = \mathbf{2}_4) \wedge (\mathbf{y}_4[2:0] = \mathbf{7}_3)$

satisfiable: $v(\mathbf{x}_4) = \mathbf{8}_4$ and $v(\mathbf{y}_4) = \mathbf{15}_4$

unsigned \ggg_u shifts in 0s
signed \ggg_s shifts in sign bits

$\mathbf{x}[i:j]$ denotes $x_i \dots x_j$
and $::$ is concatenation

Notation for Constants

- ▶ n_k is binary representation of n in k bits
- ▶ xn_k is binary representation of hexadecimal n in k bits

Example

- ▶ $0_1, 3_2, 10_4, 1024_{32, \dots}$
- ▶ $x0_4, xa_4, xb0_8, x11cf_{16}, xfffffff_{32, \dots}$

More examples

negation uses two's complement

- ▶ $-a_4 = a_4$

satisfiable: $v(a_4) = -8_4 = x8_4$

- ▶ $a_8 \div_u b_8 = a_8 \gg_u 1_8$

satisfiable: $v(a_8) = 8_8$ and $v(b_8) = 2_8$

satisfied for powers of 2 (and 0)

- ▶ $a_8 \& (a_8 - 1_8) = 0_8$

satisfiable: $v(a_8) = 8_8$ or $x0_8, x1_8, x2_8, x4_8, x8_8, x10_8, x20_8, x40_8, x80_8$

Remarks

- ▶ theory is decidable because carrier is finite
- ▶ common decision procedures use translation to SAT (**bit blasting**)
 - ▶ eager: no DPLL(T), bit-blast entire formula to SAT problem
 - ▶ lazy: second SAT solver as BV theory solver, bit-blast only BV atoms
- ▶ solvers heavily rely on **preprocessing via rewriting**

Example (Preprocessing)

$$\begin{aligned} \mathbf{x}_1 \neq \mathbf{0}_1 \wedge (\mathbf{y}_3 :: \mathbf{x}_1) \%_u \mathbf{2}_4 = \mathbf{0}_4 &\rightarrow \mathbf{x}_1 = \mathbf{1}_1 \wedge (\mathbf{y}_3 :: \mathbf{x}_1) \%_u \mathbf{2}_4 = \mathbf{0}_4 \\ &\rightarrow (\mathbf{y}_3 :: \mathbf{1}_1) \%_u \mathbf{2}_4 = \mathbf{0}_4 \rightarrow \mathbf{F} \end{aligned}$$

Definition (Bit Blasting: Formulas)

bit blasting transformation \mathbf{B} transforms BV formula into propositional formula:

$$\mathbf{B}(\varphi \vee \psi) = \mathbf{B}(\varphi) \vee \mathbf{B}(\psi)$$

$$\mathbf{B}(\varphi \wedge \psi) = \mathbf{B}(\varphi) \wedge \mathbf{B}(\psi)$$

$$\mathbf{B}(\neg\varphi) = \neg\mathbf{B}(\varphi)$$

$$\mathbf{B}(t_1 \text{ rel } t_2) = \mathbf{B}_r(u_1 \text{ rel } u_2) \wedge \varphi_1 \wedge \varphi_2 \quad \text{if } \mathbf{B}_t(t_1) = (u_1, \varphi_1) \text{ and } \mathbf{B}_t(t_2) = (u_2, \varphi_2)$$

bit blasting \mathbf{B}_t for term t
returns (result u , side condition φ)

\mathbf{B}_r transforms atom into propositional formula

Definition (Bit Blasting: Atoms)

for bit vectors \mathbf{x}_k and \mathbf{y}_k set

► equality

$$\mathbf{B}_r(\mathbf{x}_{k+1} = \mathbf{y}_{k+1}) = (x_k \leftrightarrow y_k) \wedge \cdots \wedge (x_1 \leftrightarrow y_1) \wedge (x_0 \leftrightarrow y_0)$$

► inequality

$$\mathbf{B}_r(\mathbf{x}_{k+1} \neq \mathbf{y}_{k+1}) = (x_k \oplus y_k) \vee \cdots \vee (x_1 \oplus y_1) \vee (x_0 \oplus y_0)$$

► unsigned greater-than or equal

$$\mathbf{B}_r(\mathbf{x}_1 \geq_u \mathbf{y}_1) = y_0 \rightarrow x_0$$

$$\mathbf{B}_r(\mathbf{x}_{k+1} \geq_u \mathbf{y}_{k+1}) = (x_k \wedge \neg y_k) \vee ((x_k \leftrightarrow y_k) \wedge \mathbf{B}(\mathbf{x}[k-1:0] \geq \mathbf{y}[k-1:0]))$$

► unsigned greater-than

$$\mathbf{B}(\mathbf{x}_k >_u \mathbf{y}_k) = \mathbf{B}(\mathbf{x}_k \geq \mathbf{y}_k) \wedge \mathbf{B}(\mathbf{x}_k \neq \mathbf{y}_k)$$

Definition (Bit Blasting: Bitwise Operations)

for bit vectors \mathbf{x}_k and \mathbf{y}_k use fresh variable \mathbf{z}_k and set

- ▶ bitwise and

$$\mathbf{B}_t(\mathbf{x}_k \& \mathbf{y}_k) = (\mathbf{z}_k, \varphi) \quad \varphi = \bigwedge_{i=0}^{k-1} z_i \leftrightarrow (x_i \wedge y_i)$$

- ▶ bitwise or

$$\mathbf{B}_t(\mathbf{x}_k | \mathbf{y}_k) = (\mathbf{z}_k, \varphi) \quad \varphi = \bigwedge_{i=0}^{k-1} z_i \leftrightarrow (x_i \vee y_i)$$

- ▶ bitwise exclusive or

$$\mathbf{B}_t(\mathbf{x}_k \wedge \mathbf{y}_k) = (\mathbf{z}_k, \varphi) \quad \varphi = \bigwedge_{i=0}^{k-1} z_i \leftrightarrow (x_i \oplus y_i)$$

- ▶ bitwise negation

$$\mathbf{B}_t(-\mathbf{x}_k) = (\mathbf{z}_k, \varphi) \quad \varphi = \bigwedge_{i=0}^{k-1} z_i \leftrightarrow \neg x_i$$

Definition (Bit Blasting: Concatenation, Extraction, If)

► concatenation

$$\mathbf{B}_t(\mathbf{x}_k :: \mathbf{y}_m) = (\mathbf{x}_k \mathbf{y}_m, \top)$$

for bit vectors \mathbf{x}_k and \mathbf{y}_m

► extraction

$$\mathbf{B}_t(\mathbf{x}[n:m]) = (\mathbf{z}_{n-m+1}, \varphi) \quad \varphi = \bigwedge_{i=0}^{n-m} z_i \leftrightarrow x_{i+m}$$

for bit vector \mathbf{x}_k , $k > n \geq m \geq 0$ and fresh variable \mathbf{z}_{n-m+1}

► if-then-else

$$\mathbf{B}_t(p ? \mathbf{x}_k : \mathbf{y}_k) = (\mathbf{z}_k, \varphi) \quad \varphi = \bigwedge_{i=0}^{k-1} (p \rightarrow (z_i \leftrightarrow x_i)) \wedge (\neg p \rightarrow (z_i \leftrightarrow y_i))$$

for formula p and bit vectors \mathbf{x}_k and \mathbf{y}_k

Definition (Bit Blasting: Addition and Subtraction)

► addition

$$\mathbf{B}_t(\mathbf{x}_k + \mathbf{y}_k) = (\mathbf{s}_k, \varphi)$$

where

$$\varphi = (c_0 \leftrightarrow x_0 \wedge y_0) \wedge (s_0 \leftrightarrow x_0 \oplus y_0) \wedge \bigwedge_{i=1}^{k-1} (c_i \leftrightarrow \text{min2}(x_i, y_i, c_{i-1})) \wedge (s_i \leftrightarrow x_i \oplus y_i \oplus c_{i-1})$$

ripple-carry adder:
 c_k are carry bits

for fresh variables \mathbf{s}_k and \mathbf{c}_k and $\text{min2}(a, b, d) = (a \wedge b) \vee (a \wedge d) \vee (b \wedge d)$

► unary minus

$$\mathbf{B}_t(-\mathbf{x}_k) = \mathbf{B}_t(\sim \mathbf{x}_k + \mathbf{1}_k)$$

► subtraction

$$\mathbf{B}_t(\mathbf{x}_k - \mathbf{y}_k) = \mathbf{B}_t(\mathbf{x}_k + (-\mathbf{y}_k))$$

Definition (Bit Blasting: Multiplication and Division)

for bit vectors \mathbf{x}_k and \mathbf{y}_k set

► multiplication

$$\mathbf{B}_t(\mathbf{x}_k \times \mathbf{y}_k) = \mathbf{B}_t(\text{mul}(\mathbf{x}_k, \mathbf{y}_k, 0))$$

where

$$\text{mul}(\mathbf{x}_k, \mathbf{y}_k, k) = \mathbf{0}_k$$

$$\text{mul}(\mathbf{x}_k, \mathbf{y}_k, i) = \text{mul}(\mathbf{x}_k \ll \mathbf{1}_k, \mathbf{y}_k, i + 1) + (y_i ? \mathbf{x}_k : \mathbf{0}_k) \quad \text{if } i < k$$

shift-and-add

► unsigned division

$$\mathbf{B}_t(\mathbf{x}_k \div_u \mathbf{y}_k) = (\mathbf{q}_k, \varphi)$$

$$\varphi = \mathbf{B}(\mathbf{y}_k \neq \mathbf{0}_k \rightarrow (\mathbf{q}_k \times \mathbf{y}_k + \mathbf{r}_k = \mathbf{x}_k \wedge \mathbf{r}_k < \mathbf{y}_k \wedge \mathbf{q}_k < \mathbf{x}_k))$$

for fresh variables \mathbf{q}_k and \mathbf{r}_k

Example (SMT-LIB 2 for BV)

$(a_4 + b_4 <_u b_4) \wedge (a_4 \neq 10_4) \wedge (a_4 \& b_4 = 8_4)$ is expressed as

```
(declare-const a (_ BitVec 4))
(declare-const b (_ BitVec 4))
(assert (bvult (bvadd a b) b))
(assert (not (= a #xa)))
(assert (= (bvand a b) #b1000))
(check-sat)
```



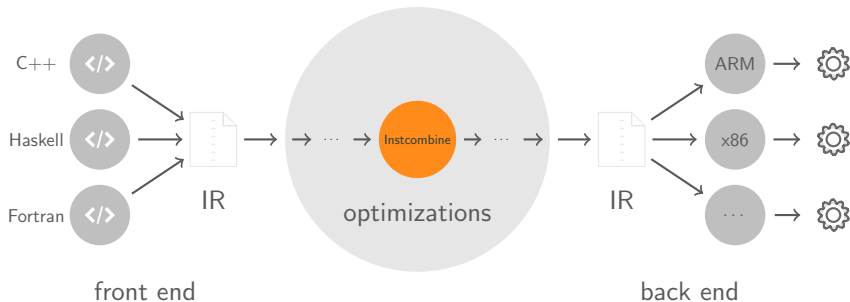
BV in SMT-LIB 2

- ▶ `(_ BitVec k)` is sort of bitvectors of length k
- ▶ `#xa` is constant in hexadecimal
- ▶ `#b1000` is constant in binary
- ▶ `bvadd`, `bvsub`, `bvmul` are arithmetic operations, `bvudiv` and `bvstdiv` are unsigned and signed division
- ▶ `bvult` and `bvule` are unsigned, `bvslt` and `bvsle` are signed $<$ and \leq
- ▶ `bvshl`, `bvlshr`, `bvashr` are shifts
- ▶ `bvand`, `bvor` are bitwise logical operations

Application: Verifying Compiler Optimizations (1)

LLVM

- ▶ open-source umbrella project: set of reusable toolchain components: libraries, assemblers, compilers, debuggers, ...
- ▶ compilation toolchain includes peephole optimizations in **Instcombine** pass



Application: Verifying Compiler Optimizations (2)

Instcombine Pass

- ▶ over 1000 algebraic simplifications of expressions
 - ▶ transform multiplies with constant power-of-two argument into shifts
 - ▶ bitwise operators with constant operands are always grouped so that shifts are performed first, then ors, then ands, then xors
 - ▶ changing bitwidth of variables
 - ▶ ...
- ▶ code is community maintained
- ▶ sometimes optimizations have errors—and compiler bugs are critical

Example

```
int foo(int z) {  
  int x = 4 * (z | 1001);  
  return -256 & x;  
}
```

.....→

```
define i32 @foo(i32) #0 {  
  %2 = or i32 %0, 1001  
  %3 = mul nsw i32 4, %2  
  %4 = xor i32 -256, %3  
  ret i32 %4  
}
```

Instcombine →

```
define i32 @foo(i32) #0 {  
  %2 = shl i32 %0, 2  
  %3 = or i32 %2, 4004  
  %4 = xor i32 %3, -256  
  ret i32 %4  
}
```

Application: Verifying Compiler Optimizations (3)

Alive Project

- ▶ represent Instcombine optimizations in domain-specific language, e.g.

```
Name: PR20186
```

```
%a = sdiv %X, C
```

```
%r = sub 0, %a
```

```
=>
```

```
%r = sdiv %X, -C
```

- ▶ check correctness by means of SMT encoding

```
(declare-const x (_ BitVec 32))
(declare-const c (_ BitVec 32))
(declare-const before (_ BitVec 32))
(declare-const after (_ BitVec 32))
(assert (= before (bvsub #x00000000 (bvdiv x c))))
(assert (= after (bvdiv x (bvneg c))))
(assert (not (= before after)))
(assert (not (= c #x00000000)))
(check-sat)
```



- ▶ **wrong** for `c = x = #x80000000`

Bit Vectors in python/z3

```
from z3 import *
x = BitVec('x', 32) # create variable named x with 32 bits
c = BitVec('c', 32)

before = BitVecVal(0, 32) - (x / c)
after = x / - c

solver = Solver()
solver.add(c != BitVecVal(0, 32)) # exclude case where c=0
solver.add(after != before)

result = solver.check()
if result == z3.sat:
    m = solver.model()
    print m[x], m[c] # 2147483648 2147483648
    print m.eval(before), m.eval(after) # 4294967295 1
```

Application: Detecting Nontermination

```
int bsearch(int a[], int k, unsigned int lo, unsigned int hi) {
    unsigned int mid;
    while (lo < hi) {
        mid = (lo + hi)/2;
        if (a[mid] < k)
            lo = mid + 1;
        else if (a[mid] > k)
            hi = mid - 1;
        else
            return mid;
    }
    return -1;
}
```

- ▶ (former) implementation of binary search in Java library
- ▶ loops for inputs $lo=1$ and $hi=UINT_MAX$ if $a[0] < k$.
- ▶ SMT encoding can find values such that parameters stay the same in recursive call



Daniel Kroening and Ofer Strichman

Bit Vectors

Chapter 6 of Decision Procedures — An Algorithmic Point of View
Springer, 2008



Nuno Lopes, David Menendez, Sarantosh Nagarakatte, and John Regehr.

Provably Correct Peephole Optimizations with Alive.

Proc. 36th PLDI, pp. 22–32, 2013.