SAT and SMT Solvers in Practice

Marijn J.H. Heule

Carnegie Mellon University

20th International Colloquium on Theoretical Aspects of Computing

December 5, 2023, Lima, Peru

`https://github.com/marijnheule/sat-examples.git`

DIMACS: SAT solver input format

The DIMACS format for SAT solvers has three types of lines:

- header: `p cnf n m` in which `n` denotes the highest variable index and `m` the number of clauses
- clauses: a sequence of integers ending with "0"
- comments: any line starting with "c "

$$
\begin{array}{l}
(a \vee b \vee \overline{c}) \wedge \\
(\overline{a} \vee \overline{b} \vee c) \wedge \\
(b \vee c \vee \overline{d}) \wedge \\
(\overline{b} \vee \overline{c} \vee d) \wedge \\
(a \vee c \vee d) \wedge \\
(\overline{a} \vee \overline{c} \vee \overline{d}) \wedge \\
(\overline{a} \vee b \vee d)
\end{array}
$$

```
c example
p cnf 4 7
 1  2 -3 0
-1 -2  3 0
 2  3 -4 0
-2 -3  4 0
 1  3  4 0
-1 -3 -4 0
-1  2  4 0
```

DIMACS: SAT solver output format

The solution line of a SAT solver starts with "`s `":

- `s SATISFIABLE`: The formula is satisfiable
- `s UNSATISFIABLE`: The formula is unsatisfiable
- `s UNKNOWN`: The solver cannot determine satisfiability

In case the formula is satisfiable, the solver emits a certificate:

- lines starting with "`v `"
- a list of integers ending with 0
- e.g. `v -1 2 4 0`

In case the formula is unsatisfiable, then most solvers support emitting a proof of unsatisfiability to a separate file

CaDiCaL: download and install

Most SAT solvers are implemented in C/C++

CaDiCaL is one of the strongest SAT solvers. As the name suggests it is based on CDCL. Recommended for Linux and macOS users.

obtain CaDiCaL:

- `git clone https://github.com/arminbiere/cadical.git`
- cd cadical
- ./configure; make

to run: `./build/cadical` formula.cnf

Kissat: download and install

Most SAT solvers are implemented in C/C++

Kissat is successor of CaDiCaL and it is written in C.
Recommended for Linux and macOS users.

obtain Kissat:

- `git clone https://github.com/arminbiere/kissat.git`
- cd kissat
- ./configure; make

to run: `./build/kissat` formula.cnf

SAT4J: download and install

SAT4J is a SAT solver in Java. It is also based on CDCL. Recommended for windows users.

obtain SAT4J:

- `git clone https://github.com/marijnheule/sat-examples.git`
- cd sat-examples

to run: `java -jar org.sat4j.core-2.3.1.jar` formula.cnf

UBCSAT: download and install

UBCSAT is a collection of local search SAT solvers.

obtain UBCSAT:

- download and unzip `http://ubcsat.dtompkins.com/downloads/ubcsat-beta-12-b18.tar.gz`
- cd ubcsat-beta-12-b18
- make clean; make

to run: `./ubcsat -alg ddfw -i` formula.cnf

there are many LS algorithms to choose from (`-alg`)
`./ubcsat -ha` (shows the available algorithms)

YalSAT: download and install

YalSAT: **y**et **a**nother **l**ocal search **SAT** solver:

obtain YalSAT:

- `git clone https://github.com/arminbiere/yalsat.git`
- cd yalsat
- ./configure.sh; make

to run: `./yalsat` formula.cnf

A powerful local search solver from the author of CaDiCaL and Kissat

Many SAT solvers

Many SAT solvers have been developed

Lots of them participate in the annual SAT competition

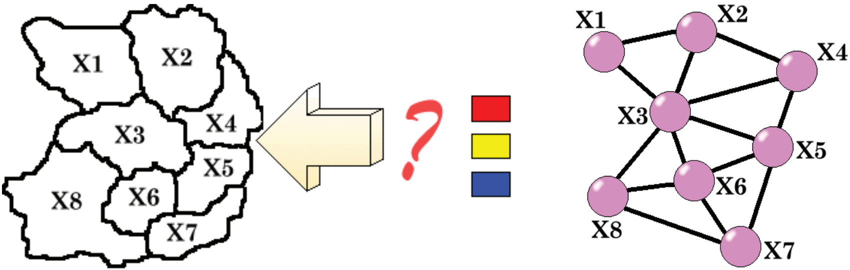
- All code of participants in open source
- Each solver is run on hundreds of benchmarks
- Large timeout 5000 seconds

For details and downloading more solvers visit
`http://satcompetition.org/`

Demo: SAT Solving

Graph coloring

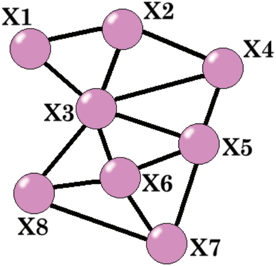
Given a graph $G(V, E)$, can the vertices be colored with k colors such that for each edge $(v, w) \in E$, the vertices v and w are colored differently.



Graph Coloring: Format

- Header starts with `p edge`
- Followed by number of vertices and number of edges

```
p edge 8 13
e 1 2
e 1 3
e 2 3
e 2 4
e 3 5
e 3 6
e 3 8
e 4 5
e 5 6
e 5 7
e 6 7
e 6 8
e 7 8
```



Graph coloring encoding

Variables	**Range**	**Meaning**
$x_{v,i}$	$i \in \{1, \ldots, c\}$ $v \in \{1, \ldots, \lvert V \rvert\}$	node v has color i

Clauses	**Range**	**Meaning**
$(x_{v,1} \vee x_{v,2} \vee \cdots \vee x_{v,c})$	$v \in \{1, \ldots, \lvert V \rvert\}$	v is colored
$(\overline{x}_{v,s} \vee \overline{x}_{v,t})$	$s \in \{1, \ldots, c-1\}$ $t \in \{s+1, \ldots, c\}$	v has at most one color
$(\overline{x}_{v,i} \vee \overline{x}_{w,i})$	$(v, w) \in E$	v and w have a different color
???	???	breaking symmetry

Graph coloring encoding code

```c
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char** argv) {
  FILE* graph  = fopen (argv[1], "r");
  int i, j, a, b, nVertex, nEdge, nColor = atoi  (argv[2]);
  fscanf (graph, " p edge %i %i ", &nVertex, &nEdge);

  printf ("p cnf %i %i\n", nVertex * nColor, nVertex + nEdge * nColor);

  for (i = 0; i < nVertex; i++) {
    for (j = 1; j <= nColor; j++)
      printf ("%i ", i * nColor + j);
    printf ("0\n"); }

  while (1) {
    int tmp = fscanf (graph, " e %i %i ", &a, &b);
    if (tmp == 0 || tmp == EOF) break;
    for (j = 1; j <= nColor; j++)
      printf ("-%i -%i 0\n", (a-1) * nColor + j, (b-1) * nColor + j);
  }
}
```

Demo: Encode, Decode

Graph Coloring: Sudoku

Sudoku can be viewed as a graph coloring problem:

- Each cell is a vertex
- Vertices are connected if they occur in the same row / column / square
- There are 9 colors

The solution must be unique

- At least 17 givens

Who can solve this sudoku?

	4		3					
						7	9	
			6					
			1	4		5		
9							1	
2								6
				7	2			
	5					8		
				9				

Graph Coloring: Sudoku

Sudoku can be viewed as a graph coloring problem:

- Each cell is a vertex
- Vertices are connected if they occur in the same row / column / square
- There are 9 colors

The solution must be unique

- At least 17 givens

Who can solve this sudoku?

1	4	*7*	3	*8*	*9*	*2*	*6*	*5*
5	*8*	*6*	*2*	*1*	*4*	7	9	*3*
3	*9*	*2*	6	*5*	*7*	*1*	*8*	*4*
8	*7*	*3*	1	4	*6*	5	*2*	*9*
9	*6*	*4*	*7*	*2*	*5*	*3*	1	*8*
2	*1*	*5*	*9*	*3*	*8*	*4*	*7*	6
6	*3*	*8*	*5*	7	2	*9*	*4*	*1*
7	5	*9*	*4*	*6*	*1*	8	*3*	*2*
4	*2*	*1*	*8*	9	*3*	*6*	*5*	*7*

Unsatisfiable cores

An unsatisfiable core of an unsatisfiable formula F is a subset of F that is unsatisfiable.

An minimal unsatisfiable core of an unsatisfiable formula such that the removal of any clause makes the formula satisfiable.

Extracting a minimal unsatisfiable core from a formula has many applications, but the computational costs could be high.

- maxSAT
- diagnosis
- formal verification

Proofs

A proof of unsatisfiability is a certificate that a given formula is unsatisfiable.

Various proof producing methods exists (another lecture).

Proof checking tools cannot only validate a proof but also produce additional information about the formula:

- unsatisfiable core
- optimized proof

DRAT-trim is a tool that validates proofs and produces such information

Demo: Core Extraction

Satisfiability Modulo Theories (SMT): Introduction

Consists of five blocks:

- theory `(set-logic ...)`, e.g. QF_UF and QF_LIA
- variables, functions, and types `(declare-const ...)`
- a list of constraints `(assert ...)`
- solving the problem `(check-sat)`
- termination the solver `(exit)`

Satisfiability Modulo Theories (SMT): Introduction

Consists of five blocks:

- theory `(set-logic ...)`, e.g. QF_UF and QF_LIA
- variables, functions, and types `(declare-const ...)`
- a list of constraints `(assert ...)`
- solving the problem `(check-sat)`
- termination the solver `(exit)`

Variable and functions:

- `(declare-const name type)`
- `(declare-fun name (inputTypes) outputType)`
- `(define-fun name (inputTypes) outputType (body))`

SMT Solver: Z3

Z3 is a state-of-the-art SMT solver by Microsoft research

obtain and install Z3:

- `git clone https://github.com/Z3Prover/z3.git`
- cd z3
- ./configure may need to replace python by python3
- cd build
- make

to run: `./build/z3` formula.smt2

SMT: QF_UF example

Example

Does there exist a satisfying assignment for $p \wedge \overline{p}$?

```
(set-logic QF_UF)
(declare-const p Bool)
(assert (and p (not p)))
(check-sat) ; should be UNSAT
(exit)
```

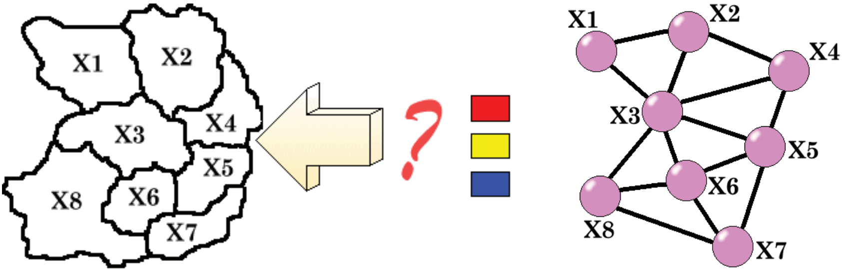
SMT: QF_LIA example

Example

Does there exist an integer x that is larger than an integer y?

```
(set-logic QF_LIA)
(declare-const x Int)
(declare-const y Int)
(assert (> x y))
(check-sat) ; should be SAT
(get-model)
(exit)
```

SMT: Graph Coloring Encoding



Variables:

- Integer variables x_i for each node

Constraints:

- $1 \leq x_i \leq c$
- $x_i \neq x_j$ for $(x_i, x_j) \in E$

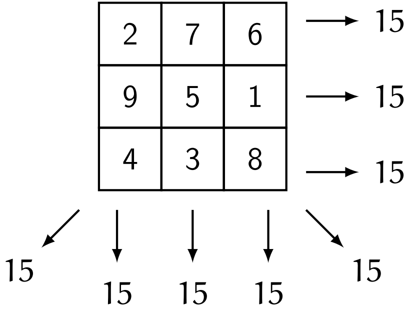
Graph coloring encoding code

```
from z3 import *
import sys

with open(sys.argv[1]) as f:
    content = f.readlines()

nodes=int(content[0].split()[2])
edges=int(content[0].split()[3])
s = Solver()
variables = []

for id in range(1,nodes+1):
        variables.append(Int('x'+str(id)))
        # each node can be assigned a value between 1 and k
        s.add(And(1 <= variables[id-1], variables[id-1] <= int(sys.argv[2])))

for line in content:
  if line[0]=='p':
        continue
  else:
        edge=line.split()
        # if two nodes are connected then they have different colors
        s.add((variables[int(edge[1])-1])!=(variables[int(edge[2])-1]))

s.check() # check if sat/unsat
print(s.model()) # print model
```

Demo: SMT Solving

Magic Squares: Introduction

A $n \times n$ square is called a magic square if each number from 1 to n^2 occurs uniquely and the sum of all rows, columns, and diagonals is the same: $(n^3 + n)/2$



Magic Squares: Linear Arithmetic

```
(set-logic QF_LIA)
(declare-const m_0_0 Int)
(declare-const m_0_1 Int)
...
(declare-const m_2_2 Int)
(assert (and (> m_0_0 0) (<= m_0_0 9)))
(assert (and (> m_0_1 0) (<= m_0_1 9)))
...
(assert (and (> m_2_2 0) (<= m_2_2 9)))
(assert (distinct m_0_0 m_0_1 m_0_2 m_1_0
                  m_1_1 m_1_2 m_2_0 m_2_1 m_2_2))
(assert (= 15 (+ m_0_0 m_0_1 m_0_2)))
(assert (= 15 (+ m_1_0 m_1_1 m_1_2)))
...
(assert (= 15 (+ m_2_0 m_1_1 m_0_2)))
(check-sat)
(get-model)
(exit)
```

Magic Squares: Bitvectors

```
(set-logic QF_BV)
(declare-const m_0_0 (_ BitVec 16))
(declare-const m_0_1 (_ BitVec 16))
...
(declare-const m_2_2 (_ BitVec 16))
(assert (and (bvugt m_0_0 #x0000) (bvule m_0_0 #x0009)))
(assert (and (bvugt m_0_1 #x0000) (bvule m_0_1 #x0009)))
...
(assert (and (bvugt m_2_2 #x0000) (bvule m_2_2 #x0009)))
(assert (distinct m_0_0 m_0_1 m_0_2 m_1_0
                  m_1_1 m_1_2 m_2_0 m_2_1 m_2_2))
(assert (= #x000f (bvadd m_0_0 m_0_1 m_0_2)))
(assert (= #x000f (bvadd m_1_0 m_1_1 m_1_2)))
...
(assert (= #x000f (bvadd m_2_0 m_1_1 m_0_2)))
(check-sat)
(get-model)
(exit)
```

Magic Squares: Theory Differences

The SMT-LIB formula for QF_BV and QF_LIA looks similar...

Magic Squares: Theory Differences

The SMT-LIB formula for QF_BV and QF_LIA looks similar...

The QF_LIA abstracts the problem by turning `(> m_2_2 0)` into a literal ($p \leftrightarrow m_{2,2} > 0$)

Magic Squares: Theory Differences

The SMT-LIB formula for QF_BV and QF_LIA looks similar...

The QF_LIA abstracts the problem by turning `(> m_2_2 0)` into a literal $(p \leftrightarrow m_{2,2} > 0)$

QF_LIA: the solver applies (exponentially) many SAT calls

Magic Squares: Theory Differences

The SMT-LIB formula for QF_BV and QF_LIA looks similar...

The QF_LIA abstracts the problem by turning `(> m_2_2 0)` into a literal ($p \leftrightarrow m_{2,2} > 0$)

QF_LIA: the solver applies (exponentially) many SAT calls

When using QF_BV, the solver applies bitblasting: every bit in each bitvector is turned into a propositional variable. Each constraint, such as `(> m_2_2 0)` is turned into many clauses.

Magic Squares: Theory Differences

The SMT-LIB formula for QF_BV and QF_LIA looks similar...

The QF_LIA abstracts the problem by turning `(> m_2_2 0)` into a literal $(p \leftrightarrow m_{2,2} > 0)$

QF_LIA: the solver applies (exponentially) many SAT calls

When using QF_BV, the solver applies bitblasting: every bit in each bitvector is turned into a propositional variable. Each constraint, such as `(> m_2_2 0)` is turned into many clauses.

QF_BV: the solver applies a single SAT call

Magic Squares: Theory Differences

The SMT-LIB formula for QF_BV and QF_LIA looks similar...

The QF_LIA abstracts the problem by turning `(> m_2_2 0)` into a literal ($p \leftrightarrow m_{2,2} > 0$)

QF_LIA: the solver applies (exponentially) many SAT calls

When using QF_BV, the solver applies bitblasting: every bit in each bitvector is turned into a propositional variable. Each constraint, such as `(> m_2_2 0)` is turned into many clauses.

QF_BV: the solver applies a single SAT call

Compare: $n \geq 5$ is hard for QF_LIA, $n \leq 10$ is easy for QF_BV

Magic Squares: Demo

```
SAT with assignment:
m_2_2 ↦ 2
m_2_1 ↦ 9
m_2_0 ↦ 4
m_1_2 ↦ 7
m_1_1 ↦ 5
m_1_0 ↦ 3
m_0_2 ↦ 6
m_0_1 ↦ 1
m_0_0 ↦ 8

Square:
8 1 6
3 5 7
4 9 2
```

Verification: Equivalence Checking

SAT and SMT solvers are crucial for verification tasks

- Equivalence checking
- Bounded model checking

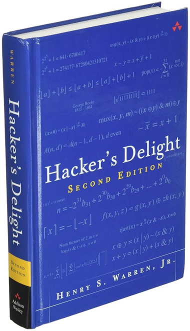
Equivalence checking:

- Are two hardware/software designs functionally equivalent?
- Does any input to both produces the same output?
- Typically one is unoptimized and the other is optimized

Verification: Popcount

Popcount: count the number of 1's in a bitvector

```
int popCount32 (unsigned int x) {
  x = x - ((x >> 1) & 0x55555555);
  x = (x & 0x33333333) + ((x >> 2) & 0x33333333);
  x = ((x + (x >> 4) & 0xf0f0f0f) * 0x1010101) >> 24;
  return x; }
```



Verification: General Setup

```
(set-logic QF_BV)
(declare-const x (_ BitVec 32))

(define-fun fast ((x (_ BitVec 32))) (_ BitVec 32)
    ...

(define-fun slow ((x (_ BitVec 32))) (_ BitVec 32)
    ...

(assert (not (= (fast x) (slow x))))
(check-sat) ; expect UNSAT
(exit)
```

Verification: Specification

```
(define-fun slow ((x (_ BitVec 32))) (_ BitVec 32)
 (bvadd
   (ite (= #b1 ((_ extract  0  0) x)) #x00000001 #x00000000)
   (ite (= #b1 ((_ extract  1  1) x)) #x00000001 #x00000000)
   (ite (= #b1 ((_ extract  2  2) x)) #x00000001 #x00000000)
                ...
   (ite (= #b1 ((_ extract 30 30) x)) #x00000001 #x00000000)
   (ite (= #b1 ((_ extract 31 31) x)) #x00000001 #x00000000)))
```

Verification: Code conversion

```
int popCount32 (unsigned int x) {
  x = x - ((x >> 1) & 0x55555555);
  x = (x & 0x33333333) + ((x >> 2) & 0x33333333);
  x = ((x + (x >> 4) & 0xf0f0f0f) * 0x1010101) >> 24;
  return x; }
```

```
(define-fun line1 ((x (_ BitVec 32))) (_ BitVec 32)
  (bvsub x (bvand (bvlshr x #x00000001) #x55555555)))

(define-fun line2 ((x (_ BitVec 32))) (_ BitVec 32)
  (bvadd (bvand x #x33333333)
         (bvand (bvlshr x #x00000002) #x33333333)))

(define-fun line3 ((x (_ BitVec 32))) (_ BitVec 32)
  (bvlshr (bvmul (bvand (bvadd (bvlshr x #x00000004)
        x) #x0f0f0f0f) #x01010101) #x00000018))

(define-fun fast ((x (_ BitVec 32))) (_ BitVec 32)
   (line3 (line2 (line1 x))))
```

Demo: Verification