Dress Code of a Speaker at a Master Class as SAT Problem

- propositional logic:
  - variables     tie   shirt
  - negation    \( \neg \) (not)
  - disjunction  \( \lor \) (or)
  - conjunction  \( \land \) (and)

- clauses (conditions / constraints)
  1. clearly one should not wear a tie without a shirt \( \neg \text{tie} \lor \text{shirt} \)
  2. not wearing a tie nor a shirt is impolite \( \text{tie} \lor \text{shirt} \)
  3. wearing a tie and a shirt is overkill \( \neg (\text{tie} \land \text{shirt}) \equiv \neg \text{tie} \lor \neg \text{shirt} \)

- Is this formula in conjunctive normal form (CNF) satisfiable?

\[ (\neg \text{tie} \lor \text{shirt}) \land (\text{tie} \lor \text{shirt}) \land (\neg \text{tie} \lor \neg \text{shirt}) \]
SAT Competition Winners on the SC2020 Benchmark Suite

CPU time

solved instances

data produced by Armin Biere and Marijn Heule
some recent Tweets

SAT solvers get faster and faster: all-time winners of the SAT Competition on 2020 instances, featuring our new solver Kissat (fmv.jku.at/kissat), which won in 2020. The web page also has runtime CDFs for 2011 and 2019.

The largest ones have millions of variables and clauses. The planning track had even larger ones. See the variable and clause distribution plot for the main track:

Eventually I will need to support 64-bit variable indices (Lingeling has $2^{*27}$, CaDiCaL indeed $2^{*31}$ and Kissat $2^{*28}$ as compromise though it could easily do half a billion).
Handbook of Satisfiability

Frontiers in Artificial Intelligence and Applications

Editors:
Armin Biere
Hans van Maaren
Toby Walsh

Copyright © 2009, Ios Press. All rights reserved.
ISBN 978-1-59033-949-7 (Hardcover)
ISBN 978-1-59033-950-3 (Electronic)
ISBN 978-1-59033-951-0 (PDF)

This is the title page of the Handbook of Satisfiability. It includes the editors' names and the copyright information.
Part I. Theory and Algorithms

John Franco, John Martin: 
_A History of Satisfiability_. 3-74

Steven David Prestwich: 
_CNF Encodings_. 75-97

Adnan Darwiche, Knot Pipatsrisawat: 
_Complete Algorithms_. 99-130

João P. Marques Silva, Inês Lynce, Sharad Malik: 
_Conflict-Driven Clause Learning SAT Solvers_. 131-153

Marijn Heule, Hans van Maaren: 
_Look-Ahead Based SAT Solvers_. 155-184

Henry A. Kautz, Ashish Sabharwal, Bart Selman: 
_Incomplete Algorithms_. 185-203

Oliver Kullmann: 
_Fundamentals of Branching Heuristics_. 205-244

Dimitris Achlioptas: 
_Random Satisfiability_. 245-270

Carla P. Gomes, Ashish Sabharwal: 
_Exploiting Runtime Variation in Complete Solvers_. 271-288

Karem A. Sakallah: 
_Symmetry and Satisfiability_. 289-338

Hans Kleine Büning, Oliver Kullmann: 
_Minimal Unsatisfiability and Autarkies_. 339-401

Evgeny Dantsin, Edward A. Hirsch: 
_Worst-Case Upper Bounds_. 403-424

Marko Samer, Stefan Szeider: 
_Fixed-Parameter Tractability_. 425-454

Part II. Applications and Extensions

Armin Biere: 
_Bounded Model Checking_. 457-481

Jussi Rintanen: 
_Planning and SAT_. 483-504

Daniel Kroening: 
_Software Verification_. 505-532

Hantao Zhang: 
_Combinatorial Designs by SAT Solvers_. 533-568

Fabrizio Montali, Rémi Monasson, Guilhem Semerjian, Francesco Zamponi: 
_Connections to Statistical Physics_. 569-611

Chu Min Li, Felip Manyà: 
_MaxSAT, Hard and Soft Constraints_. 613-631

Carla P. Gomes, Ashish Sabharwal, Bart Selman: 
_Model Counting_. 633-654

Rolf Drechsler, Tommi A. Junttila, Ilkka Niemelä: 
_Non-Clausal SAT and ATPG_. 655-693

Olivier Roussel, Vasco M. Manquinho: 
_Pseudo-Boolean and Cardinality Constraints_. 695-733

Hans Kleine Büning, Uwe Bubeck: 
_Theory of Quantified Boolean Formulas_. 735-760

Enrico Giunchiglia, Paolo Marin, Massimo Narizzano: 
_Reasoning with Quantified Boolean Formulas_. 761-780

Roberto Sebastiani, Armando Tacchella: 
_SAT Techniques for Modal and Description Logics_. 781-824

Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, Cesare Tinelli: 
_Satisfiability Modulo Theories_. 825-885

Stephen M. Majercik: 
_Stochastic Boolean Satisfiability_. 887-925
Special thanks are due to Armin Biere, Randy Bryant, Sam Buss, Niklas Eén, Ian Gent, Marijn Heule, Holger Hoos, Svante Janson, Peter Jeavons, Daniel Kroening, Oliver Kullmann, Massimo Lauria, Wes Pegden, Will Shortz, Carsten Sinz, Niklas Sörensson, Udo Wermuth, Ryan Williams, and... for their detailed comments on my early attempts at exposition, as well as to numerous other correspondents who have contributed crucial corrections. Thanks also to Stanford’s Information Systems Laboratory for providing extra computer power when my laptop machine was inadequate.

* * * * *

Wow—Section 7.2.2.2 has turned out to be the longest section, by far, in The Art of Computer Programming. The SAT problem is evidently a “killer app,” because it is key to the solution of so many other problems. Consequently I can only hope that my lengthy treatment does not also kill off my faithful readers! As I wrote this material, one topic always seemed to flow naturally into another, so there was no neat way to break this section up into separate subsections. (And anyway the format of TAOCP doesn’t allow for a Section 7.2.2.2.1.)

I’ve tried to ameliorate the reader’s navigation problem by adding subheadings at the top of each right-hand page. Furthermore, as in other sections, the exercises appear in an order that roughly parallels the order in which corresponding topics are taken up in the text. Numerous cross-references are provided...
The SAT problem is evidently a killer app, because it is key to the solution of so many other problems. SAT-solving techniques are among computer science’s best success stories so far, and these volumes tell that fascinating tale in the words of the leading SAT experts.

Donald Knuth

... Clearly, efficient SAT solving is a key technology for 21st century computer science. I expect this collection of papers on all theoretical and practical aspects of SAT solving will be extremely useful to both students and researchers and will lead to many further advances in the field.

Edmund Clarke
What is Practical SAT Solving?

1st part
- encoding

reencoding

simplifying

other talks

2nd part
- search
Equivalence Checking If-Then-Else Chains

original C code

```c
if(!a && !b) h();
else if(!a) g();
else f();
else h();
```

```c
if(!a) {
    if(a) f();
    if(!b) h();
    \[\Rightarrow\]
    else
    {
        \[\Rightarrow\]
        else g(); if(!b) h();
    }
    else f();
}
```

optimized C code

```c
if(a) f();
else if(b) g();
else h();
```

How to check that these two versions are equivalent?
Compilation

\[
\begin{align*}
\text{original} & \equiv \text{if } \neg a \land \neg b \text{ then } h \text{ else if } \neg a \text{ then } g \text{ else } f \\
& \equiv (\neg a \land \neg b) \land h \lor (\neg (a \land b)) \land (\text{if } \neg a \text{ then } g \text{ else } f) \\
& \equiv (\neg a \land \neg b) \land h \lor (\neg (a \land b)) \land (\neg a \land g \lor a \land f)
\end{align*}
\]

\[
\begin{align*}
\text{optimized} & \equiv \text{if } a \text{ then } f \text{ else if } b \text{ then } g \text{ else } h \\
& \equiv a \land f \lor \neg a \land (\text{if } b \text{ then } g \text{ else } h) \\
& \equiv a \land f \lor \neg a \land (b \land g \lor \neg b \land h)
\end{align*}
\]

\[
(\neg a \land \neg b) \land h \lor (\neg (a \land b)) \land (\neg a \land g \lor a \land f) \not\iff a \land f \lor \neg a \land (b \land g \lor \neg b \land h)
\]

satisfying assignment gives counter-example to equivalence
Tseitin Transformation: Circuit to CNF

\[ o \land (x \rightarrow a) \land (x \rightarrow c) \land (x \leftarrow a \land c) \land \ldots \]

\[ o \land (\overline{x} \lor a) \land (\overline{x} \lor c) \land (x \lor \overline{a} \lor \overline{c}) \land \ldots \]
Tseitin Transformation: Gate Constraints

Negation: \[ x \leftrightarrow \overline{y} \Leftrightarrow (x \rightarrow \overline{y}) \land (\overline{y} \rightarrow x) \]
\[ \Leftrightarrow (\overline{x} \lor \overline{y}) \land (y \lor x) \]

Disjunction: \[ x \leftrightarrow (y \lor z) \Leftrightarrow (y \rightarrow x) \land (z \rightarrow x) \land (x \rightarrow (y \lor z)) \]
\[ \Leftrightarrow (\overline{y} \lor x) \land (\overline{z} \lor x) \land (\overline{x} \lor y \lor z) \]

Conjunction: \[ x \leftrightarrow (y \land z) \Leftrightarrow (x \rightarrow y) \land (x \rightarrow z) \land ((y \land z) \rightarrow x) \]
\[ \Leftrightarrow (\overline{x} \lor y) \land (\overline{x} \lor z) \land ((y \land z) \lor x) \]
\[ \Leftrightarrow (\overline{x} \lor y) \land (\overline{x} \lor z) \land (\overline{y} \lor \overline{z} \lor x) \]

Equivalence: \[ x \leftrightarrow (y \leftrightarrow z) \Leftrightarrow (x \rightarrow (y \leftrightarrow z)) \land ((y \leftrightarrow z) \rightarrow x) \]
\[ \Leftrightarrow (x \rightarrow ((y \rightarrow z) \land (z \rightarrow y))) \land ((y \leftrightarrow z) \rightarrow x) \]
\[ \Leftrightarrow (x \rightarrow (y \rightarrow z)) \land (x \rightarrow (z \rightarrow y)) \land ((y \leftrightarrow z) \rightarrow x) \]
\[ \Leftrightarrow (\overline{x} \lor \overline{y} \lor z) \land (\overline{x} \lor \overline{z} \lor y) \land ((y \leftrightarrow z) \lor x) \]
\[ \Leftrightarrow (\overline{x} \lor \overline{y} \lor z) \land (\overline{x} \lor \overline{z} \lor y) \land ((y \land z) \lor (\overline{y} \land \overline{z})) \lor x) \]
\[ \Leftrightarrow (\overline{x} \lor \overline{y} \lor z) \land (\overline{x} \lor \overline{z} \lor y) \land ((y \land z) \lor (\overline{y} \land \overline{z})) \lor x) \]
\[ \Leftrightarrow (\overline{x} \lor \overline{y} \lor z) \land (\overline{x} \lor \overline{z} \lor y) \land (y \lor z \lor x) \]
Bit-Blasting of Bit-Vector Addition

addition of 4-bit numbers $x, y$ with result $s$ also 4-bit: \[ s = x + y \]

\[ [s_3, s_2, s_1, s_0]_4 = [x_3, x_2, x_1, x_0]_4 + [y_3, y_2, y_1, y_0]_4 \]

\[ [s_3, \cdot]_2 = \text{FullAdder}(x_3, y_3, c_2) \]
\[ [s_2, c_2]_2 = \text{FullAdder}(x_2, y_2, c_1) \]
\[ [s_1, c_1]_2 = \text{FullAdder}(x_1, y_1, c_0) \]
\[ [s_0, c_0]_2 = \text{FullAdder}(x_0, y_0, \text{false}) \]

where

\[ [s, o]_2 = \text{FullAdder}(x, y, i) \quad \text{with} \]
\[ s = x \text{xor} y \text{xor} i \]
\[ o = (x \land y) \lor (x \land i) \lor (y \land i) = ((x + y + i) \geq 2) \]
Boolector Architecture

O1 = bottom up simplification
O2 = global but almost linear
O3 = normalizing (often non-linear) [default]
Intermediate Representations

- encoding directly into CNF is hard, so we use intermediate levels:
  1. application level
  2. bit-precise semantics world-level operations (bit-vectors)
  3. bit-level representations such as And-Inverter Graphs (AIGs)
  4. conjunctive normal form (CNF)

- encoding “logical” constraints is another story
XOR as AIG

\[ x \text{ xor } y \equiv (\bar{x} \land y) \lor (x \land \bar{y}) \equiv \overline{(\bar{x} \land y)} \land (x \land \bar{y}) \]

negation/sign are edge attributes

not part of node
bit-vector of length 16 shifted by bit-vector of length 4
Encoding Logical Constraints

- Tseitin construction suitable for most kinds of “model constraints”
  - assuming simple operational semantics: encode an interpreter
  - small domains: one-hot encoding, large domains: binary encoding

- harder to encode properties or additional constraints
  - temporal logic / fix-points
  - environment constraints

- example for fix-points / recursive equations: \( x = (a \lor y), \quad y = (b \lor x) \)
  - has unique least fix-point \( x = y = (a \lor b) \)
  - and unique largest fix-point \( x = y = \text{true} \)
  - but unfortunately . . .
  - . . . only largest fix-point can be (directly) encoded in SAT
  - otherwise need stable models / logical programming / ASP
Example of Logical Constraints: Cardinality Constraints

- given a set of literals \( \{l_1, \ldots l_n\} \)
  - constraint the number of literals assigned to \textit{true}
  - \( l_1 + \cdots + l_n \geq k \) or \( l_1 + \cdots + l_n \leq k \) or \( l_1 + \cdots + l_n = k \)
  - combined make up exactly all fully symmetric boolean functions

- multiple encodings of cardinality constraints
  - naïve encoding exponential: \textit{at-most-one} quadratic, \textit{at-most-two} cubic, etc.
  - quadratic \( O(k \cdot n) \) encoding goes back to Shannon
  - linear \( O(n) \) parallel counter encoding \cite{Sinz'05}

- many variants even for \textit{at-most-one} constraints
  - for an \( O(n \cdot \log n) \) encoding see Prestwich’s chapter in Handbook of SAT

- Pseudo-Boolean constraints (PB) or 0/1 ILP constraints have many encodings too
  \[
  2 \cdot \overline{a} + \overline{b} + c + \overline{d} + 2 \cdot e \geq 3
  \]

actually used to handle MaxSAT in SAT4J for configuration in Eclipse
BDD-Based Encoding of Cardinality Constraints

\[ 2 \leq l_1 + \cdots l_9 \leq 3 \]

If-Then-Else gates (MUX) with “then” edge downward, dashed “else” edge to the right
Tseitin Encoding of If-Then-Else Gate

\[
x \leftrightarrow (c \ ? \ t : e) \iff (x \to (c \to t)) \land (x \to (\bar{c} \to e)) \land (\bar{x} \to (c \to \bar{t})) \land (\bar{x} \to (\bar{c} \to \bar{e}))
\]

\[
\iff (\bar{x} \lor \bar{c} \lor t) \land (\bar{x} \lor c \lor e) \land (x \lor \bar{c} \lor \bar{t}) \land (x \lor c \lor \bar{e})
\]

minimal but **not** arc consistent:

- if \( t \) and \( e \) have the same value then \( x \) needs to have that too

- possible additional clauses

\[
(t \land e \rightarrow x) \equiv (t \lor e \lor \bar{x}) \quad \quad (t \lor e \rightarrow x) \equiv (\bar{t} \lor \bar{e} \lor x)
\]

- but can be learned or derived through preprocessing (ternary resolution)
  keeping those clauses redundant is better in practice
DIMACS Format

$ cat example.cnf
c comments start with 'c' and extend until the end of the line
c
 c variables are encoded as integers:
c
 c  'tie' becomes '1'
c  'shirt' becomes '2'
c
 c header 'p cnf <variables> <clauses>'
c
p cnf 2 3
-1  2  0  c  !tie or shirt
 1  2  0  c  tie or shirt
-1 -2  0  c  !tie or !shirt

$ picosat example.cnf
s SATISFIABLE
v -1 2 0
SAT Application Programmatic Interface (API)

- incremental usage of SAT solvers
  - add facts such as clauses incrementally
  - call SAT solver and get satisfying assignments
  - optionally retract facts

- retracting facts
  - remove clauses explicitly: complex to implement
  - push / pop: stack like activation, no sharing of learned facts
  - MiniSAT assumptions [EénSörensson’03]

- assumptions
  - unit assumptions: assumed for the next SAT call
  - easy to implement: force SAT solver to decide on assumptions first
  - shares learned clauses across SAT calls

- IPASIR: Reentrant Incremental SAT API
  - used in the SAT competition / race since 2015 [BalyoBierelserSinz’16]
```c
#include "ipasir.h"
#include <assert.h>
#include <stdio.h>
#define ADD(LIT) ipasir_add (solver, LIT)
#define PRINT(LIT) \
  printf (ipasir_val (solver, LIT) < 0 ? " -" #LIT : " " #LIT)

int main () {
  void * solver = ipasir_init ();
  enum { tie = 1, shirt = 2 };
  ADD (-tie); ADD ( shirt); ADD (0);
  ADD ( tie); ADD ( shirt); ADD (0);
  ADD (-tie); ADD (-shirt); ADD (0);
  int res = ipasir_solve (solver);
  assert (res == 10);
  printf ("satisfiable:"); PRINT (shirt); PRINT (tie); printf ("\n");
  printf ("assuming now: tie shirt\n");
  ipasir_assume (solver, tie); ipasir_assume (solver, shirt);
  res = ipasir_solve (solver);
  assert (res == 20);
  printf ("unsatisfiable, failed:");
  if (ipasir_failed (solver, tie)) printf (" tie");
  if (ipasir_failed (solver, shirt)) printf (" shirt");
  printf ("\n");
  ipasir_release (solver);
  return res;
}
```

$ ./example
satisfiable: shirt -tie
assuming now: tie shirt
unsatisfiable, failed: tie
IPASIR Functions

const char * ipasir_signature ();

void * ipasir_init ();

void ipasir_release (void * solver);

void ipasir_add (void * solver, int lit_or_zero);

void ipasir_assume (void * solver, int lit);

int ipasir_solve (void * solver);

int ipasir_val (void * solver, int lit);

int ipasir_failed (void * solver, int lit);

void ipasir_set_terminate (void * solver, void * state,
                           int (*terminate)(void * state));
```cpp
#include "cadical.hpp"
#include <cassert>
#include <iostream>
using namespace std;
#define ADD(LIT) solver.add (LIT)
#define PRINT(LIT) \
  (solver.val (LIT) < 0 ? " -" #LIT : " " #LIT)
int main () {
  CaDiCaL::Solver solver; solver.set ("quiet", 1);
  enum { tie = 1, shirt = 2 };  
  ADD (-tie), ADD ( shirt), ADD (0);
  ADD ( tie), ADD ( shirt), ADD (0);
  ADD (-tie), ADD (-shirt), ADD (0);
  int res = solver.solve ();
  assert (res == 10);
  cout << "satisfiable:" << PRINT (shirt) << PRINT (tie) << endl;
  cout << "assuming now: tie shirt" << endl;
  solver.assume (tie), solver.assume (shirt);
  res = solver.solve ();
  assert (res == 20);
  cout << "unsatisfiable, failed:";
  if (solver.failed (tie)) cout << " tie";
  if (solver.failed (shirt)) cout << " shirt";
  cout << endl;
  return res;
}
```
DP / DPLL

- dates back to the 50’ies:
  1\textsuperscript{st} version DP is \textit{resolution based} ⇒ preprocessing
  2\textsuperscript{nd} version D(P)LL splits space for time ⇒ \textbf{CDCL}

- \textbf{ideas:}
  - 1\textsuperscript{st} version: eliminate the two cases of assigning a variable in space or
  - 2\textsuperscript{nd} version: case analysis in time, e.g. try $x = 0, 1$ in turn and recurse

- most successful SAT solvers are based on variant (CDCL) of the second version
  works for very large instances

- recent ($\leq 25$ years) optimizations:
  backjumping, learning, UIPs, dynamic splitting heuristics, fast data structures
DP Procedure

forever

if $F = \top$ return satisfiable

if $\bot \in F$ return unsatisfiable

pick remaining variable $x$

add all resolvents on $x$

remove all clauses with $x$ and $\neg x$

$\Rightarrow$ Bounded Variable Elimination
DPLL Procedure

\[ DPLL(F) \]

\[ F := BCP(F) \]

if \( F = \top \) return \text{satisfiable}

if \( \bot \in F \) return \text{unsatisfiable}

pick remaining variable \( x \) and literal \( l \in \{x, \neg x\} \)

if \( DPLL(F \land \{l\}) \) returns \text{satisfiable} return \text{satisfiable}

return \( DPLL(F \land \{\neg l\}) \)

\( \Rightarrow \) CDCL
DPLL Example

Decision

\[ \neg a \vee \neg b \vee \neg c \]

\[ \neg a \vee b \vee c \]

\[ \neg a \vee \neg b \vee \neg c \]

\[ a \vee \neg b \vee \neg c \]

\[ a \vee b \vee \neg c \]

\[ a \vee b \vee c \]
Conflict Driven Clause Learning (CDCL)
[MarqueSilvaSakallah'96]

- first implemented in the context of GRASP SAT solver
  - name given later to distinguish it from DPLL
  - not recursive anymore
- essential for SMT
- learning clauses as no-goods
- notion of implication graph
- (first) unique implication points
Conflict Driven Clause Learning (CDCL)

\[
\begin{align*}
    a &= 1 \\
    b &= 1 \\
    c &= 0
\end{align*}
\]

BCP

decision

\[
\begin{align*}
    \neg a \lor \neg b \lor \neg c \\
    \neg a \lor \neg b \lor c \\
    a \lor \neg b \lor \neg c \\
    a \lor \neg b \lor c \\
    a \lor b \lor \neg c \\
    a \lor b \lor c
\end{align*}
\]

learn \( \neg a \lor \neg b \)
Conflict Driven Clause Learning (CDCL)

\[ a = 1 \]
\[ b = 0 \]
\[ c = 0 \]

Decision: \( a \)

BCP

Clauses:

\[ \neg a \lor \neg b \lor \neg c \]
\[ \neg a \lor b \lor c \]
\[ a \lor \neg b \lor \neg c \]
\[ a \lor b \lor \neg c \]
\[ a \lor b \lor c \]
\[ \neg a \lor \neg b \]

Learn: \( \neg a \)
Conflict Driven Clause Learning (CDCL)

\[
\begin{align*}
  a &= 1 \\
  b &= 0 \\
  c &= 0
\end{align*}
\]
Conflict Driven Clause Learning (CDCL)

\[
\begin{align*}
  a &= 1 \\
  b &= 0 \\
  c &= 0
\end{align*}
\]

\[
\begin{align*}
  \neg a & \quad \text{BCP} \\
  c & \quad \text{BCP} \\
  b & \quad \text{BCP}
\end{align*}
\]

clauses

\[
\begin{align*}
  \neg a \lor \neg b \lor \neg c \\
  \neg a \lor \neg b \lor c \\
  \neg a \lor b \lor \neg c \\
  \neg a \lor b \lor c \\
  \neg a \lor \neg b \\
  \neg a \\
  c
\end{align*}
\]

learn

empty clause
Implication Graph

d = 1 @ 1
b = 1 @ 0
a = 1 @ 0
f g = 1 @ 2
h = 1 @ 2
i = 1 @ 2
l = 1 @ 3
c = 1 @ 1
d = 1 @ 1
e = 1 @ 1
f = 1 @ 2
g = 1 @ 2
h = 1 @ 2
i = 1 @ 2
k = 1 @ 3
l = 1 @ 3
r = 1 @ 4
s = 1 @ 4
t = 1 @ 4
y = 1 @ 4
x = 1 @ 4
z = 1 @ 4
κ conflict
Antecedents / Reasons

d \land g \land s \rightarrow t \equiv (\overline{d} \lor \overline{g} \lor \overline{s} \lor t)
\[
\neg (y \land z) \equiv (\bar{y} \lor \bar{z})
\]
Resolving Antecedents 1\textsuperscript{st} Time

\begin{align*}
\text{top-level} & \quad \text{unit} \quad a = 1 @ 0 \quad \text{unit} \quad b = 1 @ 0 \\
\text{decision} & \quad c = 1 @ 1 \quad \rightarrow \quad d = 1 @ 1 \quad \rightarrow \quad e = 1 @ 1 \\
\text{decision} & \quad f = 1 @ 2 \quad \rightarrow \quad g = 1 @ 2 \quad \rightarrow \quad h = 1 @ 2 \quad \rightarrow \quad i = 1 @ 2 \\
\text{decision} & \quad k = 1 @ 3 \quad \rightarrow \quad l = 1 @ 3 \\
\text{decision} & \quad r = 1 @ 4 \quad \rightarrow \quad s = 1 @ 4 \quad \rightarrow \quad t = 1 @ 4 \quad \rightarrow \quad y = 1 @ 4 \\
\text{decision} & \quad x = 1 @ 4 \quad \rightarrow \quad z = 1 @ 4 \quad \rightarrow \quad \kappa \quad \text{conflict} \\
\end{align*}

\((\overline{h} \lor \overline{i} \lor \overline{j} \lor y)\quad (y \lor \overline{z})\)
Resolving Antecedents 1st Time

\[
\begin{align*}
\text{top-level} & \quad \text{unit} & a &= 1 @ 0 & \text{unit} & b &= 1 @ 0 \\
\text{decision} & \quad c &= 1 @ 1 & \quad d &= 1 @ 1 & \quad e &= 1 @ 1 \\
\text{decision} & \quad f &= 1 @ 2 & \quad g &= 1 @ 2 & \quad h &= 1 @ 2 & \quad i &= 1 @ 2 \\
\text{decision} & \quad k &= 1 @ 3 & \quad l &= 1 @ 3 \\
\text{decision} & \quad r &= 1 @ 4 & \quad s &= 1 @ 4 & \quad t &= 1 @ 4 & \quad y &= 1 @ 4 \\
\quad x &= 1 @ 4 & \quad z &= 1 @ 4 & \quad \kappa & & \text{conflict} \\
\end{align*}
\]

\[
(h \lor \bar{i} \lor i \lor y) \quad (y \lor z)
\]

\[
(h \lor \bar{i} \lor i \lor z)
\]
Resolvents = Cuts = Potential Learned Clauses

d = 1 @ 1
e = 1 @ 1
b = 1 @ 0
a = 1 @ 0
f g = 1 @ 2
l = 1 @ 3
= 1 @ 1
c
k = 1 @ 3
r = 1 @ 4
s = 1 @ 4
t = 1 @ 4
x = 1 @ 4
z = 1 @ 4
κ conflict

(\bar{h} \lor \bar{i} \lor \bar{t} \lor y) (\bar{y} \lor z)

(\bar{h} \lor \bar{i} \lor \bar{t} \lor \bar{z})
Potential Learned Clause After 1 Resolution

\[(\overline{h} \lor \overline{i} \lor \overline{i} \lor z)\]
Resolving Antecedents 2\textsuperscript{nd} Time

\[
\begin{align*}
\text{top–level} & \quad \text{unit} & \quad a = 1 @ 0 & \quad \text{unit} & \quad b = 1 @ 0 \\
\text{decision} & \quad c = 1 @ 1 & \quad d = 1 @ 1 & \quad e = 1 @ 1 \\
\text{decision} & \quad f = 1 @ 2 & \quad g = 1 @ 2 & \quad h = 1 @ 2 & \quad i = 1 @ 2 \\
\text{decision} & \quad k = 1 @ 3 & \quad l = 1 @ 3 \\
\text{decision} & \quad r = 1 @ 4 & \quad s = 1 @ 4 & \quad t = 1 @ 4 & \quad y = 1 @ 4 \\
\quad & \quad x = 1 @ 4 & \quad z = 1 @ 4 \\
\end{align*}
\]

\[
(\overline{d} \lor \overline{g} \lor \overline{s} \lor t) & \quad (\overline{h} \lor \overline{i} \lor \overline{t} \lor \overline{z}) \\
(\overline{d} \lor \overline{g} \lor \overline{s} \lor \overline{h} \lor \overline{i} \lor \overline{z})
\]
Resolving Antecedents 3rd Time

\[(\bar{x} \lor z) \quad (\bar{d} \lor \bar{g} \lor \bar{s} \lor \bar{h} \lor \bar{i} \lor \bar{z})\]

\[(\bar{x} \lor d \lor g \lor s \lor h \lor i)\]
Resolving Antecedents 4\textsuperscript{th} Time

\[ (\overline{s} \lor x) \quad (\overline{x} \lor \overline{d} \lor \overline{g} \lor \overline{s} \lor \overline{h} \lor \overline{i}) \]

self subsuming resolution
1st UIP Clause after 4 Resolutions

UIP = unique implication point dominates conflict on the last level
If $y$ has never been used to derive a conflict, then skip $\bar{y}$ case.

Immediately jump back to the $\bar{x}$ case – assuming $x$ was used.
Resolving Antecedents 5\textsuperscript{th} Time

\[
\begin{align*}
&\text{top-level} \quad \text{unit} \quad a = 1 @ 0 \quad \text{unit} \quad b = 1 @ 0 \\
\text{decision} \quad c = 1 @ 1 \quad d = 1 @ 1 \quad e = 1 @ 1 \\
\text{decision} \quad f = 1 @ 2 \quad g = 1 @ 2 \quad h = 1 @ 2 \quad i = 1 @ 2 \\
\text{decision} \quad k = 1 @ 3 \quad l = 1 @ 3 \\
\text{decision} \quad r = 1 @ 4 \quad s = 1 @ 4 \quad t = 1 @ 4 \quad y = 1 @ 4 \\
\end{align*}
\]

\[
\begin{align*}
x = 1 @ 4 \quad z = 1 @ 4 \quad \kappa \quad \text{conflict}
\end{align*}
\]

\[
\begin{align*}
(\bar{I} \lor \bar{r} \lor s) & \quad (\bar{d} \lor \bar{g} \lor \bar{s} \lor \bar{h} \lor \bar{i}) \\
(\bar{I} \lor \bar{r} \lor \bar{d} \lor \bar{g} \lor \bar{h} \lor \bar{i})
\end{align*}
\]
Decision Learned Clause

\[(\bar{d} \lor \bar{g} \lor \bar{l} \lor \bar{r} \lor \bar{h} \lor \bar{i})\]
1st UIP Clause after 4 Resolutions

\[(\overline{d} \lor \overline{g} \lor \overline{s} \lor \overline{h} \lor \overline{i})\]
Locally Minimizing 1st UIP Clause

\[
(\overline{h} \lor i) (\overline{d} \lor \overline{g} \lor \overline{s} \lor \overline{h} \lor \overline{i})
\]

\[
(\overline{d} \lor \overline{g} \lor \overline{s} \lor \overline{h})
\]

self subsuming resolution
Locally Minimized Learned Clause

\[
\bar{d} \lor \bar{g} \lor \bar{s} \lor \bar{h}
\]
Minimizing Locally Minimized Learned Clause Further?

\[ (\overline{d} \lor \overline{g} \lor \overline{s} \lor \overline{h}) \]
Recursively Minimizing Learned Clause

\[
\begin{align*}
  b &= 1 @ 0 \\
  e &= 1 @ 1 \\
  h &= 1 @ 2 \\
  i &= 1 @ 2 \\
  s &= 1 @ 4 \\
  t &= 1 @ 4 \\
  y &= 1 @ 4 \\
  x &= 1 @ 4 \\
  z &= 1 @ 4 \\
  \kappa &= \text{conflict}
\end{align*}
\]

\[
\begin{align*}
  & (\bar{e} \lor \bar{g} \lor h) \\
  & (\bar{d} \lor \bar{g} \lor \bar{s} \lor \bar{h}) \\
  & (\bar{d} \lor \bar{b} \lor e) \\
  & (\bar{e} \lor \bar{d} \lor \bar{g} \lor s) \\
  & (b) \\
  & (b \lor \bar{d} \lor \bar{g} \lor \bar{s}) \\
  & (\bar{d} \lor \bar{g} \lor \bar{s})
\end{align*}
\]
Recursively Minimized Learned Clause

\((\overline{d} \lor \overline{g} \lor \overline{s})\)
Decision Heuristics

- number of variable occurrences in (remaining unsatisfied) clauses (LIS)
  - eagerly satisfy many clauses with many variations studied in the 90ies
  - actually expensive to compute

- dynamic heuristics
  - focus on variables which were useful recently in deriving learned clauses
  - can be interpreted as reinforcement learning
  - started with the VSIDS heuristic \cite{MoskewiczMadiganZhaoZhangMalik01}
  - most solvers rely on the exponential variant in MiniSAT (EVSIDS)
  - recently showed VMTF as effective as VSIDS \cite{BiereFröhlichSAT15} survey

- look-ahead
  - spent more time in selecting good variables (and simplification)
  - related to our Cube & Conquer paper \cite{HeuleKullmanWieringaBiereHVC11}
  - “The Science of Brute Force” \cite{HeuleKullmanCACM17}

- EVSIDS during stabilization VMTF otherwise \cite{BiereSATRace19}
Fast VMTF Implementation

- Siege SAT solver [Ryan Thesis 2004] used variable move to front (VMTF)
  - bumped variables moved to head of doubly linked list
  - search for unassigned variable starts at head
  - variable selection is an online sorting algorithm of scores
  - classic “move-to-front” strategy achieves good amortized complexity

- fast simple implementation for caching searches in VMTF [BiereFröhlich’SAT15]
  - doubly linked list does not have positions as an ordered array
  - bump = move-to-front = dequeue then insertion at the head

- time-stamp list entries with “insertion-time”
  - maintained invariant: all variables right of next-search are assigned
  - requires (constant time) update to next-search while unassigning variables
  - occasion ally (32-bit) time-stamps will overflow: update all time stamps
### Variable Scoring Schemes

[BiereFröhlich-SAT’15]

<table>
<thead>
<tr>
<th>Old score $s$</th>
<th>New score $s'$ after $i$ conflicts</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>STATIC</strong></td>
<td>bumped</td>
</tr>
<tr>
<td>INC</td>
<td>$s$</td>
</tr>
<tr>
<td>SUM</td>
<td>$s + 1$</td>
</tr>
<tr>
<td></td>
<td>$s + i$</td>
</tr>
<tr>
<td>VSIDS</td>
<td>$h_i^{256} \cdot s + 1$</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>NVSIDS</td>
<td>$f \cdot s + (1 - f)$</td>
</tr>
<tr>
<td>EVSIDS</td>
<td>$s + g^i$</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>ACIDS</td>
<td>$(s + i)/2$</td>
</tr>
<tr>
<td>VMTF$_1$</td>
<td>$i$</td>
</tr>
<tr>
<td>VMTF$_2$</td>
<td>$b$</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$0 < f < 1 \quad g = 1/f \quad h_i^m = 0.5 \quad \text{if } m \text{ divides } i \quad h_i^m = 1 \text{ otherwise}$

$i$ conflict index  \quad $b$ bumped counter
Basic CDCL Loop

```c
int basic_cdcl_loop () {
    int res = 0;

    while (!res)
        if (unsat) res = 20; // analyze propagated conflict
        else if (!propagate ()) analyze (); // analyze propagated conflict
        else if (satisfied ()) res = 10; // all variables satisfied
        else decide (); // otherwise pick next decision

    return res;
}
```
Reducing Learned Clauses

- keeping all learned clauses slows down BCP
  - so SATO and RelSAT just kept only “short” clauses

- better periodically delete “useless” learned clauses
  - keep a certain number of learned clauses
  - if this number is reached MiniSAT reduces (deletes) half of the clauses
  - then maximum number kept learned clauses is increased geometrically

- LBD (glucose level / glue) prediction for usefulness
  - LBD = number of decision-levels in the learned clause
  - allows arithmetic increase of number of kept learned clauses
  - keep clauses with small LBD forever ($\leq 2 \ldots 5$)
  - three Tier system by

- eagerly reduce hyper-binary resolvents derived in inprocessing

[AudemardSimon-IJCAI’09]
[Chanseok Oh]
Restarts

- often it is a good strategy to abandon what you do and restart
  - for satisfiable instances the solver may get stuck in the unsatisfiable part
  - for unsatisfiable instances focusing on one part might miss short proofs
  - restart after the number of conflicts reached a restart limit

- avoid to run into the same dead end
  - by randomization (either on the decision variable or its phase)
  - and/or just keep all the learned clauses during restart

- for completeness dynamically increase restart limit
  - arithmetically, geometrically, Luby, Inner/Outer

- Glucose restarts  [AudemardSimon-CP’12]
  - short vs. large window exponential moving average (EMA) over LBD
  - if recent LBD values are larger than long time average then restart

- interleave “stabilizing” (no restarts) and “non-stabilizing” phases  [Chanseok Oh]
  - call it now “stabilizing mode” and “focused mode”
Luby’s Restart Intervals
70 restarts in 104448 conflicts
unsigned
luby (unsigned i)
{
    unsigned k;

    for (k = 1; k < 32; k++)
        if (i == (1 << k) - 1)
            return 1 << (k - 1);

    for (k = 1;; k++)
        if ((1 << (k-1)) <= i && i < (1 << k) - 1)
            return luby (i - (1 << (k-1)) + 1);

}

limit = 512 * luby (++restarts);
...  // run SAT core loop for 'limit' conflicts
Reluctant Doubling Sequence

\[(u_1, v_1) = (1, 1)\]

\[(u_{n+1}, v_{n+1}) = ((u_n \& -u_n == v_n) ? (u_n + 1, 1) : (u_n, 2v_n))\]

\[(1,1), (2,1), (2,2), (3,1), (4,1), (4,2), (4,4), (5,1), \ldots\]
Restart Scheduling with Exponential Moving Averages

[BiereFröhlich-POS’15]

- LBD — fast $EMA$ of LBD with $\alpha = 2^{-5}$
- restart — slow $EMA$ of LBD with $\alpha = 2^{-14}$ (ema-14)
- inprocessing — $CMA$ of LBD (average)
Phase Saving and Rapid Restarts

- **phase assignment:**
  - assign decision variable to 0 or 1?
  - “Only thing that matters in **satisfiable** instances” [Hans van Maaren]

- “**phase saving**” as in RSat [PipatsrisawatDarwiche’07]
  - pick phase of last assignment (if not forced to, do not toggle assignment)
  - initially use statically computed phase (typically LIS)
  - so can be seen to maintain a **global full assignment**

- rapid restarts
  - varying restart interval with bursts of restarts
  - not only theoretically avoids local minima
  - works nicely together with phase saving

- reusing the trail can reduce the cost of restarts [RamosVanDerTakHeule-JSAT’11]

- target phases of largest conflict free trail / assignment
  [Biere-SAT-Race-2019] [BiereFleury-POS-2020]
int basic_cdcl_loop_with_reduce_and_restart () {

    int res = 0;

    while (!res)
        if (unsat) res = 20; // analyze propagated conflict
    else if (!propagate ()) analyze (); // analyze propagated conflict
    else if (satisfied ()) res = 10; // all variables satisfied
    else if (restarting ()) restart (); // restart by backtracking
    else if (reducing ()) reduce (); // collect useless learned clauses
    else decide (); // otherwise pick next decision

    return res;
}
while (!res) {
    if (unsat) res = 20;
    else if (!propagate ()) analyze ();  // propagate and analyze
    else if (iterating) iterate ();      // report learned unit
    else if (satisfied ()) res = 10;    // found model
    else if (search_limits_hit ()) break;  // decision or conflict limit
    else if (terminated_asynchronously ()) break;  // externally terminated
    break;
    else if (restarting ()) restart ();   // restart by backtracking
    else if (rephasing ()) rephase ();   // reset variable phases
    else if (reducing ()) reduce ();     // collect useless clauses
    else if (probing ()) probe ();      // failed literal probing
    else if (subsuming ()) subsume ();   // subsumption algorithm
    else if (eliminating ()) elim ();    // variable elimination
    else if (compacting ()) compact ();  // collect variables
    else if (conditioning ()) condition ();  // globally blocked clauses
    else res = decide ();                 // next decision
}
https://github.com/arminbiere/cadical
https://fmv.jku.at/cadical
Two-Watched Literal Schemes

- original idea from SATO
  - invariant: always watch two non-false literals
  - if a watched literal becomes false replace it
  - if no replacement can be found clause is either unit or empty
  - original version used head and tail pointers on Tries

- improved variant from Chaff
  - watch pointers can move arbitrarily
  - no update needed during backtracking

  one watch is enough to ensure correctness
  - but looses arc consistency

  reduces visiting clauses by 10x
  - particularly useful for large and many learned clauses

- blocking literals

- special treatment of short clauses (binary [PilarskiHu’02] or ternary [Ryan’04])

- cache start of search for replacement
Things we did not discuss …

- advanced preprocessing and inprocessing
  IJCAI-JAIR 2019 award for [HeuleJärvisaloLonsingSeidlBiere-JAIR-2015]
  (many) best papers with Marijn Heule and Benjamin Kiesl
  [PhD thesis of Benjamin Kiesl 2019]

- proofs (Marijn Heule), certificates for UNSAT, interpolation

- relation to proof complexity
  Banff, Fields, Dagstuhl seminars

- extensions formalisms: QBF, Pseudo-Boolean, #SAT, …

- local search
  this year’s best solvers have all local search in it

- challenges: arithmetic reasoning (and proofs)
  best paper [KaufmannBiereKauers-FMCAD’17] [PhD thesis Daniela Kaufmann 2020]

- chronological backtracking
  [RyvchinNadel-SAT’18] [MöhleBiere-SAT’19]

- incremental SAT solving
  best student paper [FazekasBiereScholl-SAT’19] [PhD thesis of Katalin Fazekas in 2020]

- parallel and distributed SAT solving
  Handbook of Parallel Constraint Reasoning, …
Personal SAT Solver History

- **1960**: SAT for Planning
- **1970**: NP complete
- **1980**: Tseitin Encoding
- **1990**: BMC
- **2000**: SMT
- **2010**: ProbSAT

**Look Ahead**

- **1980**: WalkSAT
- **1990**: GSAT

**Portfolio Phase**

- **2000**: CDCL
- **2010**: Phase Saving

- **2010**: Proofs
- **2010**: Avatar

**Handbook of SAT**

- **1980**: Cube & Conquer
- **1980**: SAT Chapter Donald Knuth

**SAT & SMT everywhere**

- **1980**: BMC
- **1990**: BMC
- **1960**: BMC

**QBF working**

- **1980**: Massively Parallel
- **1990**: Massively Parallel

**SMT**

- **1980**: SMT
- **1990**: SMT

**DPLL**

- **1980**: DPLL
- **1990**: DPLL

**BMC**

- **1990**: BMC
- **2000**: BMC

**VSIDS**

- **1990**: VSIDS
- **2000**: VSIDS

**LBD**

- **1990**: LBD
- **2000**: LBD

**Inprocessing**

- **2000**: Inprocessing
- **2010**: Inprocessing

**Solvers**

- **2010**: Solvers
- **2010**: Solvers

**Arithmetic Solvers**

- **2010**: Arithmetic Solvers
- **2010**: Arithmetic Solvers

**SAT & SMT everywhere**

- **2010**: SAT & SMT everywhere
- **2010**: SAT & SMT everywhere

**NP complete**

- **1980**: NP complete
- **1990**: NP complete

**SAT VSIDSCDCL**

- **1980**: SAT VSIDSCDCL
- **1990**: SAT VSIDSCDCL

**1st SAT competition**

- **1980**: 1st SAT competition
- **1990**: 1st SAT competition

**Tseitin Encoding**

- **1980**: Tseitin Encoding
- **1990**: Tseitin Encoding

**Portfolio**

- **1980**: Portfolio
- **1990**: Portfolio

**Bounded Variable Elimination**

- **1980**: Bounded Variable Elimination
- **1990**: Bounded Variable Elimination

**Avatars**

- **1980**: Avatars
- **1990**: Avatars

**Massively Parallel**

- **1980**: Massively Parallel
- **1990**: Massively Parallel