Dress Code Tutorial Speaker as SAT Problem

- propositional logic:
  - variables: tie  shirt
  - negation: ¬ (not)
  - disjunction: ∨ (or)
  - conjunction: ∧ (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?

\[ (\neg \text{tie} \lor \text{shirt}) \land (\text{tie} \lor \text{shirt}) \land (\neg \text{tie} \lor \neg \text{shirt}) \]
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.)
What is Practical SAT Solving?

- Encoding
- Simplifying
- Inprocessing
- CDCL
- Search
- Reencoding
## Equivalence Checking If-Then-Else Chains

<table>
<thead>
<tr>
<th>original C code</th>
<th>optimized C code</th>
<th>How to check that these two versions are equivalent?</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>if (!a &amp;&amp; !b) h();</code>&lt;br&gt;else if (!a) g();&lt;br&gt;else f();</td>
<td><code>if (a) f();</code>&lt;br&gt;else if (b) g();&lt;br&gt;else h();</td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>if (!a) {</code>&lt;br&gt;<code>if (!b) h();</code>&lt;br&gt;<code>else g();</code>&lt;br&gt;<code>} else f();</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>if (a) f();</code>&lt;br&gt;else {<code>&lt;br&gt;</code>if (!b) h();<code>&lt;br&gt;</code>else g();<code>&lt;br&gt;</code>}`</td>
<td></td>
</tr>
</tbody>
</table>

How to check that these two versions are equivalent?
Compilation

original \equiv \text{if } \neg a \land \neg b \text{ then } h \text{ else } \text{if } \neg a \text{ then } g \text{ else } f
\equiv (\neg a \land \neg b) \land h \lor (\neg (\neg a \land \neg b) \land \text{if } \neg a \text{ then } g \text{ else } f
\equiv (\neg a \land \neg b) \land h \lor (\neg (\neg a \land \neg b) \land (\neg a \land g \lor a \land f)

optimized \equiv \text{if } a \text{ then } f \text{ else } \text{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)

\neg (\neg a \land \neg b) \land h \lor (\neg (\neg a \land \neg b) \land (\neg a \land g \lor a \land f) \not\Rightarrow 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

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

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

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

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

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

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

negation/sign are edge attributes
not part of node
4-bit adder

8-bit adder
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: \( x = (a \lor y) \), \( y = (b \lor x) \)
  - 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 \( true \)
  - \( l_1 + \cdots + l_n \geq k \) 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: at-most-one quadratic, at-most-two cubic, etc.
  - quadratic \( O(k \cdot n) \) encoding goes back to Shannon
  - linear \( O(n) \) parallel counter encoding [Sinz’05]

- many variants even for 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
  \[
  2 \cdot a + b + c + d + 2 \cdot e \geq 3
  \]
  - actually used to handle MaxSAT in SAT4J for configuration in Eclipse
### BDD-Based Encoding of Cardinality Constraints

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

<table>
<thead>
<tr>
<th>( l_1 )</th>
<th>( l_2 )</th>
<th>( l_3 )</th>
<th>( l_4 )</th>
<th>( l_5 )</th>
<th>( l_6 )</th>
<th>( l_7 )</th>
<th>( l_8 )</th>
<th>( l_9 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( l_2 )</td>
<td>( l_3 )</td>
<td>( l_4 )</td>
<td>( l_5 )</td>
<td>( l_6 )</td>
<td>( l_7 )</td>
<td>( l_8 )</td>
<td>( l_9 )</td>
<td>( 0 )</td>
</tr>
<tr>
<td>( l_3 )</td>
<td>( l_4 )</td>
<td>( l_5 )</td>
<td>( l_6 )</td>
<td>( l_7 )</td>
<td>( l_8 )</td>
<td>( l_9 )</td>
<td>( -1 )</td>
<td></td>
</tr>
<tr>
<td>( l_4 )</td>
<td>( l_5 )</td>
<td>( l_6 )</td>
<td>( l_7 )</td>
<td>( l_8 )</td>
<td>( l_9 )</td>
<td>( -1 )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( 0 )</td>
<td>( 0 )</td>
<td>( 0 )</td>
<td>( 0 )</td>
<td>( 0 )</td>
<td>( 0 )</td>
<td>( 0 )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

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

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

minimal but **not** arc consistent:

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

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

- 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
variables are encoded as integers:
c
tie' becomes '1'
'shirt' becomes '2'
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 \[\text{[EénSörensson’03]}\]

- 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 \[\text{[BalyoBierelserSinz’16]}\]
IPASIR Model

UNKnown

SOLVING

UNSAT

add assume

add assume

solve

solve

interrupted

failed

solve

val
#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;
}
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));
DP / DPLL

- dates back to the 50’ies:
  - 1st version DP is resolution based
  - 2nd version D(P)LL splits space for time

- ideas:
  - 1st version: eliminate the two cases of assigning a variable in space or
  - 2nd 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 ($\leq 20$ years) optimizations:
  backjumping, learning, UIPs, dynamic splitting heuristics, fast data structures
  (we will have a look at each of them)
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
Bounded Variable Elimination
[EénBiere-SAT’05]

Replace
\[
(\bar{x} \lor a)_1 (\bar{x} \lor c)_4 (\bar{x} \lor b)_2 (x \lor d)_5
\]
by
\[
(a \lor \bar{a} \lor \bar{b})_{13} (a \lor d)_{15} (c \lor d)_{45} (b \lor \bar{a} \lor \bar{b})_{23} (b \lor d)_{25} (c \lor \bar{a} \lor \bar{b})_{34}
\]

- number of clauses not increasing
- strengthen and remove subsumed clauses too
- most important and most effective preprocessing we have

Bounded Variable Addition
[MantheyHeuleBiere-HVC’12]

Replace
\[
(a \lor d) \quad (a \lor e) \\
(b \lor d) \quad (b \lor e) \\
(c \lor d) \quad (c \lor e)
\]
by
\[
(\bar{x} \lor a) \quad (\bar{x} \lor b) \\n(x \lor d) \quad (x \lor e)
\]

- number of clauses has to decrease strictly
- reencodes for instance naive at-most-one constraint encodings
D(P)LL Procedure

\[DPLL(F)\]

\[F := \text{BCP}(F)\]

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

⇒ CDCL
DPLL Example

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

Decision ordering:
\[ a, b, c \]

BCP labels:
\[ a = 1 \]
\[ b = 1 \]
\[ c = 0 \]
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)

$a = 1$

$b = 1$

$c = 0$

decision

$\neg c$

BCP

$a$

$b$

$a \lor b \lor \neg c$

$\neg a \lor b \lor \neg c$

$\neg a \lor b \lor c$

$\neg a \lor b \lor \neg c$

$a \lor b \lor \neg c$

$a \lor b \lor c$

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

Decision $a = 1$

$b = 0$

$c = 0$

Clauses:

- $\neg a \lor \neg b \lor \neg c$
- $\neg a \lor b \lor \neg c$
- $a \lor b \lor \neg c$
- $a \lor b \lor \neg c$
- $a \lor b \lor c$

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

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

clauses
\[ \neg a \lor \neg b \lor \neg c \]
\[ \neg a \lor \neg b \lor c \]
\[ \neg a \lor b \lor \neg c \]
\[ \neg a \lor b \lor c \]
\[ a \lor \neg b \lor \neg c \]
\[ a \lor b \lor \neg c \]
\[ a \lor b \lor c \]
\[ \neg a \lor \neg b \]
\[ \neg a \]
\[ c \]
Conflict Driven Clause Learning (CDCL)

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

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

learn
\[
\bot
\]

empty clause
Implication Graph

top−level

unit

da = 1 @ 0
unit

bb = 1 @ 0

a = 1 @ 0

= 1 @ 2

f g = 1 @ 2

h = 1 @ 2

i = 1 @ 2

l = 1 @ 3

= 1 @ 1

c

k= 1 @ 3

r = 1 @ 4

s = 1 @ 4

t = 1 @ 4

y = 1 @ 4

y = 1 @ 4

x z = 1 @ 4

κ conflict
Conflict

top-level  

unit  $a = 1 \oplus 0$  unit  $b = 1 \oplus 0$

decision  $c = 1 \oplus 1$  $d = 1 \oplus 1$  $e = 1 \oplus 1$

decision  $f = 1 \oplus 2$  $g = 1 \oplus 2$  $h = 1 \oplus 2$  $i = 1 \oplus 2$

decision  $k = 1 \oplus 3$  $l = 1 \oplus 3$

decision  $r = 1 \oplus 4$  $s = 1 \oplus 4$  $t = 1 \oplus 4$  $y = 1 \oplus 4$

top-level  

don't think  

$x = 1 \oplus 4$  $z = 1 \oplus 4$  $\kappa$  conflict
Antecedents / Reasons

\[ d \land g \land s \rightarrow t \equiv (d \lor \overline{g} \lor \overline{s} \lor t) \]
Conflicting Clauses

\[
\neg (y \land z) \equiv (\neg y \lor \neg z)
\]
Resolving Antecedents $1^{st}$ Time

deploy $a = 1 @ 0$
unit $b = 1 @ 0$
decision $c = 1 @ 1$
deploy $d = 1 @ 1$
deploy $e = 1 @ 1$
deploy $f = 1 @ 2$
decision $g = 1 @ 2$
decision $h = 1 @ 2$
deploy $i = 1 @ 2$
deploy $k = 1 @ 3$
decision $l = 1 @ 3$
decision $r = 1 @ 4$
deploy $s = 1 @ 4$
deploy $t = 1 @ 4$
deploy $y = 1 @ 4$
deploy $x = 1 @ 4$
deploy $z = 1 @ 4$

cflict $\neg h \lor \neg i \lor \neg t \lor \neg y$ }


$\neg y \lor \neg z$
Resolving Antecedents 1\textsuperscript{st} Time

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

\[
\frac{(\overline{h} \lor i \lor \overline{t} \lor y)}{(y \lor z)} \quad (\overline{h} \lor i \lor \overline{t} \lor z)
\]
Resolvents = Cuts = Potential Learned Clauses

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

\[
\begin{align*}
&\overline{h} \lor i \lor \overline{i} \lor y \\
&(\overline{y} \lor \overline{z}) \\
&(\overline{h} \lor i \lor \overline{i} \lor \overline{z})
\end{align*}
\]
Potential Learned Clause After 1 Resolution

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

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

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

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

top-level

unit

$a = 1 @ 0$

unit

$b = 1 @ 0$

decision

$c = 1 @ 1$

d = 1 @ 1

e = 1 @ 1$

decision

$f = 1 @ 2$

$g = 1 @ 2$

$h = 1 @ 2$

$i = 1 @ 2$

decision

$k = 1 @ 3$

$l = 1 @ 3$

decision

$r = 1 @ 4$

$s = 1 @ 4$

t = 1 @ 4

$y = 1 @ 4$

decision

$x = 1 @ 4$

$z = 1 @ 4$

$\kappa = \text{conflict}$

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

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

self subsuming resolution
1st UIP Clause after 4 Resolutions

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

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

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

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

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

\[ (h \lor i) \quad \frac{(\bar{d} \lor \bar{g} \lor \bar{s} \lor \bar{h} \lor \bar{i})}{(\bar{d} \lor \bar{g} \lor \bar{s} \lor h)} \quad \text{self subsuming resolution} \]
Locally Minimized Learned Clause

\((\overline{d} \lor g \lor \overline{s} \lor \overline{h})\)
Minimizing Locally Minimized Learned Clause Further?

$$d = 1 \oplus 1$$

$$e = 1 \oplus 1$$

$$f = 1 \oplus 2$$

$$g = 1 \oplus 2$$

$$h = 1 \oplus 2$$

$$i = 1 \oplus 2$$

$$k = 1 \oplus 3$$

$$l = 1 \oplus 3$$

$$m = 1 \oplus 4$$

$$n = 1 \oplus 4$$

$$o = 1 \oplus 4$$

$$p = 1 \oplus 4$$

$$q = 1 \oplus 4$$

$$r = 1 \oplus 4$$

$$s = 1 \oplus 4$$

$$t = 1 \oplus 4$$

$$k = 1 \oplus 4$$

$$\overline{c} \lor \overline{f} \lor \overline{s} \lor \overline{h}$$
Recursively Minimizing Learned Clause

\[
\frac{(\overline{d} \lor \overline{b} \lor e)}{(b \lor \overline{d} \lor \overline{g} \lor \overline{s})} \quad \frac{(\overline{e} \lor \overline{g} \lor h)}{(d \lor \overline{g} \lor \overline{s} \lor \overline{h})}
\]
Recursively Minimized Learned Clause

\[ (d \lor g \lor s) \]
Decision Heuristics

- number of variable occurrences in (remaining unsatisfied) clauses (LIS)
  - eagerly satisfy many clauses
  - many variations were 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 \[\text{MoskewiczMadiganZhaoZhangMalik’01}\]
  - most solvers rely on the exponential variant in MiniSAT (EVSIDS)
  - recently showed VMTF as effective as VSIDS \[\text{BiereFröhlich-SAT’15}\] survey

- look-ahead
  - spent more time in selecting good variables (and simplification)
  - related to our Cube & Conquer paper \[\text{HeuleKullmanWieringaBiere-HVC’11}\]
  - “The Science of Brute Force” \[\text{Heule & Kullman CACM August 2017}\]
Variable Scoring Schemes
[BiereFröhlich-SAT’15]

\[ s \text{ old score} \quad s' \text{ new score} \]

<table>
<thead>
<tr>
<th>variable score ( s' ) after ( i ) conflicts</th>
<th>bumped</th>
<th>not-bumped</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>STATIC</strong></td>
<td>( s )</td>
<td>( s )</td>
</tr>
<tr>
<td><strong>INC</strong></td>
<td>( s + 1 )</td>
<td>( s )</td>
</tr>
<tr>
<td><strong>SUM</strong></td>
<td>( s + i )</td>
<td>( s )</td>
</tr>
<tr>
<td><strong>VSIDS</strong></td>
<td>( h_i^{256} \cdot s + 1 )</td>
<td>( h_i^{256} \cdot s )</td>
</tr>
<tr>
<td><strong>NVSIDS</strong></td>
<td>( f \cdot s + (1 - f) )</td>
<td>( f \cdot s )</td>
</tr>
<tr>
<td><strong>EVSIDS</strong></td>
<td>( s + g^i )</td>
<td>( s )</td>
</tr>
<tr>
<td><strong>ACIDS</strong></td>
<td>( (s + i)/2 )</td>
<td>( s )</td>
</tr>
<tr>
<td><strong>VMTF</strong></td>
<td>( i )</td>
<td>( s )</td>
</tr>
<tr>
<td><strong>VMTF’</strong></td>
<td>( b )</td>
<td>( s )</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
```c
int basic_cdcl_loop () {
    int res = 0;

    while (!res)
        if (unsat) res = 20;
        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 [AudemardSimon-IJCAI’09]
  - LBD = number of decision-levels in the learned clause
  - allows arithmetic increase of number of kept learned clauses
  - keep clauses with small LBD forever ( ≤ 2…5)
  - large fixed cache useful for hard satisfiable instances (crypto) [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
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

[Knuth’12]

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

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

\( (1, 1), (2, 1), (2, 2), (3, 1), (4, 1), (4, 2), (4, 4), (5, 1), \ldots \)
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

- “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**
  - and thus makes CDCL actually a rather “local” search procedure

- **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]
```c
int basic_cdcl_loop_with_reduce_and_restart () {
  int res = 0;
  while (!res)
    if (unsat) res = 20;
    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;
}
```
int Internal::search () {
  int res = 0;
  START (search);
  while (!res)
    if (unsat) res = 20;
    else if (!propagate ()) analyze (); // analyze propagated conflict
    else if (iterating) iterate (); // report learned unit
    else if (satisfied ()) res = 10; // all variables satisfied
    else if (terminating ()) break; // limit hit or asynchronous abort
    else if (terminating ()) restart (); // restart by backtracking
    else if (reducing ()) reduce (); // collect useless learned clauses
    else if (probing ()) probe (); // failed literal probing
    else if (subsuming ()) subsume (); // subsumption algorithm
    else if (eliminating ()) elim (); // bounded variable elimination
    else if (compactifying ()) compact (); // collect internal variables
    else decide (); // otherwise pick next decision
  STOP (search);
  return res;
}
Two-Watched Literal Schemes

- original idea from SATO [ZhangStickel'00]
  - 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 [MoskewiczMadiganZhaoZhangMalik'01]
  - watch pointers can move arbitrarily
  - no update needed during backtracking

- one watch is enough to ensure correctness
  - reduces visiting clauses by 10x
    - particularly useful for large and many learned clauses

- blocking literals [ChuHarwoodStuckey'09]

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

- cache start of search for replacement [Gent-JAIR'13]
Proofs / RUP / DRUP

- original idea for proofs: proof traces / sequence consisting of “learned clauses”
- can be checked clause by clause through unit propagation
- reverse unit implied clauses (RUP) \[\text{GoldbergNovikov'03}] [\text{VanGelder'12}]
- deletion information (DRUP): proof trace of added and deleted clauses
- RUP in SAT competition 2007, 2009, 2011, DRUP since 2013 to certify UNSAT

Blocked Clauses
[\text{Kullman-DAM'99}] [\text{JärvisaloHeuleBiere-JAR'12}]

- clause \((a \lor l)\) “blocked” on \(l\) w.r.t. CNF \((\overline{a} \lor b) \land (l \lor c) \land (\overline{l} \lor \overline{a})\)
  - all resolvents of \(C\) on \(l\) with clauses \(D\) in \(F\) are tautological
- blocked clauses are “redundant” too
  - adding or removing blocked clauses does not change satisfiability status
  - however it might change the set of models
Resolution Asymmetric Tautologies (RAT)

"Inprocessing Rules"  [JärvisaloHeuleBiere-IJCAR'12]

- justify complex preprocessing algorithms in Lingeling
  - examples are adding blocked clauses or variable elimination
  - interleaved with research (forgetting learned clauses = reduce)
- need more general notion of redundancy criteria
  - simply replace “resolvents are tautological” by “resolvents on $l$ are RUP”

\[(a \lor l) \quad \text{RAT on } l \quad \text{w.r.t.} \quad (\bar{a} \lor b) \land (l \lor c) \land (\bar{l} \lor b)_D\]

- deletion information is again essential (DRAT)
- now mandatory in the main track of the last two SAT competitions
- pretty powerful: can for instance also cover symmetry breaking
Propagation Redundant (PR)

“Short Proofs Without New Variables” [HeuleKieslBiere-CADE’17] best paper

- more general than RAT: short proofs for pigeon hole formulas without new variables
- $C$ propagation redundant if $\exists$ (partial) assignment $\omega$ satisfying $C$ with $F | C \vdash F | \omega$
- Satisfaction Driven Clause Learning (SDCL) [HeuleKieslSeidlBiere-HVC’17]
  - first automatically generated PR proofs
  - prune paths for which we have other at least as satisfiable paths
- translate PR to DRAT [HeuleBiere-TACAS’18]
  - only one additional variable needed
  - shortest proofs for pigeon hole formulas
  - in general quadratic
Parallel SAT

- application level parallelism
- guiding path principle
- portfolio (with sharing)
- (concurrent) cube & conquer

⇒ Handbook of Parallel Constraint Reasoning

⇒ still many low-level programming issues left
Personal SAT Solver History

- DPL
- SAT
- NP complete
- 1960
- DP
- 1970
- Tseitin Encoding
- 1980
- WalkSAT
- GSAT
- 1990
- CDCL
- BMC
- SMT
- LBD
- VSIDS
- 2000
- 1st SAT competition
- 2010
- Phase Saving
- Proofs
- SAT Chapter
- Donald Knuth
- Look Ahead
- QBF
- Massively Parallel
- SAT for Planning
- Avatar
- Arithmetic Solvers
- Massively Parallel
- BMC
- SMT
- Cube & Conquer
- Panic
- SAT everywhere
- QBF working
Satisfiability (SAT), Satisfiability Modulo Theories (SMT), and Automated Reasoning (AR) continue to make rapid advances and find novel uses in a wide variety of applications, both in computer science and beyond. The SAT/SMT/AR Summer School aims to bring a select group of students up to speed quickly in this exciting research area. The school continues the successful line of Summer Schools that ran from 2011 to 2015 as SAT/SMT Summer Schools and added AR in 2016.

The summer school will be taking place in the School of Computer Science at the University of Manchester. The school will take place on 3-6 July 2018 (preceding FLoC).