Modern SAT Solvers

Part A

Vienna Winter School on Verification

6. February 2012

TU Vienna, Austria

Armin Biere
Institute for Formal Models and Verification
Johannes Kepler University, Linz, Austria

http://fmv.jku.at
What is Practical SAT Solving?

- Encoding
- Simplifying
- Inprocessing
- Search
- Reencoding?
Results of the SAT competition/race winners on the SAT 2009 application benchmarks, 20mn timeout

- Limmat (2002)
- Zchaff (2002)
- Berkmin (2002)
- Forklift (2003)
- Siege (2003)
- SatELite (2005)
- Minisat 2 (2006)
- Picosat (2007)
- Rsat (2007)
- Minisat 2.1 (2008)
- Precosat (2009)
- Glucose (2009)
- Clasp (2009)
- Cryptominisat (2010)
- Lingeling (2010)
- Minisat 2.2 (2010)
- Glucose 2 (2011)
- Glueminisat (2011)
- Contrasat (2011)

[Le Berre'11]
Results of the SAT competition/race winners on the SAT 2009 application benchmarks, 20mn timeout
SAT Example: Equivalence Checking *If-Then-Else* Chains

**original C code**

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

**optimized C code**

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

How to check that these two versions are equivalent?
1. represent procedures as *independent* boolean variables

\[
\text{original} := \begin{cases} 
  \text{if } \neg a \land \neg b \text{ then } h \\
  \text{else if } \neg a \text{ then } g \\
  \text{else } f 
\end{cases} 
\]

\[
\text{optimized} := \begin{cases} 
  \text{if } a \text{ then } f \\
  \text{else if } b \text{ then } g \\
  \text{else } h 
\end{cases} 
\]

2. compile if-then-else chains into boolean formulae

\[
\text{compile}(\text{if } x \text{ then } y \text{ else } z) \equiv (x \land y) \lor (\neg x \land z)
\]

3. check *equivalence* of the following boolean formulae

\[
\text{compile}(\text{original}) \iff \text{compile}(\text{optimized})
\]

4. same problem as checking the following formula to be *unsatisfiable*

\[
\text{compile}(\text{original}) \not\iff \text{compile}(\text{optimized})
\]
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(\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 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 a \land \neg b) \land h \lor \neg(\neg a \land \neg b) \land (\neg a \land g \lor a \land f) \Leftrightarrow a \land f \lor \neg a \land (b \land g \lor \neg b \land h)
SAT Example: Circuit Equivalence Checking

$b \lor a \land c$

$(a \lor b) \land (b \lor c)$

equivalent?

$b \lor a \land c \Leftrightarrow (a \lor b) \land (b \lor c)$
**SAT (Satisfiability)** the classical NP complete Problem:

Given a propositional formula $f$ over $n$ propositional variables $V = \{x, y, \ldots\}$.

Is there an assignment $\sigma : V \rightarrow \{0, 1\}$ with $\sigma(f) = 1$?

**SAT belongs to NP**

There is a *non-deterministic* Touring-machine deciding SAT in polynomial time:

*guess* the assignment $\sigma$ (linear in $n$), calculate $\sigma(f)$ (linear in $|f|$)

**Note:** on a *real* (deterministic) computer this would still require $2^n$ time

**SAT is complete for NP** (see complexity / theory class)

**Implications for us:**
general SAT algorithms are probably exponential in time (unless NP = P)
**Definition**

A formula in *Conjunctive Normal Form* (CNF) is a conjunction of clauses

\[ C_1 \land C_2 \land \ldots \land C_n \]

Each clause \( C \) is a disjunction of literals

\[ C = L_1 \lor \ldots \lor L_m \]

And each literal is either a plain variable \( x \) or a negated variable \( \bar{x} \).

**Example**  \((a \lor b \lor c) \land (\bar{a} \lor \bar{b}) \land (\bar{a} \lor \bar{c})\)

**Note 1:** Two notions for negation: in \( \bar{x} \) and \( \neg \) as in \( \neg x \) for denoting negation.

**Note 2:** The original SAT problem is actually formulated for CNF.

**Note 3:** SAT solvers mostly also expect CNF as input.
Negation Normal Form (NNF)  AND/OR form + negations only occur in front of variables

use De’Morgan (push negations inward) to translate into NNF

\[
a \leftrightarrow (b \land a) \equiv (a \rightarrow (b \land a)) \land (a \leftarrow (b \land a)) \\
\equiv (\overline{a} \lor (b \land a)) \land (a \lor (\overline{b} \lor \overline{a})) \\
\equiv (\overline{a} \lor (b \land a)) \land (a \lor (\overline{b} \lor \overline{a}))
\]

in NNF

use distributivity of OR over AND  (“multiply out \textit{outer} \lor ”)

\[
(\overline{a} \lor b) \land (\overline{a} \lor a) \land (a \lor \overline{b} \lor \overline{a})
\]

and simplify to finally obtain  \((\overline{a} \lor b)\)

unfortunaly really expensive:  \((\bigwedge C_i) \lor (\bigwedge D_j) \equiv \bigwedge (C_i \lor D_j) \quad O(n^2)\)
Example of Tseitin Transformation: Circuit to CNF

\[
\begin{align*}
& \bigwedge o \\
& \bigwedge (x \leftrightarrow a \land c) \\
& \bigwedge (y \leftrightarrow b \lor x) \\
& \bigwedge (u \leftrightarrow a \lor b) \\
& \bigwedge (v \leftrightarrow b \lor c) \\
& \bigwedge (w \leftrightarrow u \land v) \\
& \bigwedge (o \leftrightarrow y \oplus w)
\end{align*}
\]

\[
\begin{align*}
& \bigwedge o \land (x \rightarrow a) \land (x \rightarrow c) \land (x \leftarrow a \land c) \land \ldots \\
& \bigwedge o \land (\bar{x} \lor a) \land (\bar{x} \lor c) \land (x \lor \bar{a} \lor \bar{c}) \land \ldots
\end{align*}
\]
Tseitin Transformation: Input / Output Constraints

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

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

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

Equivalence: \[ x \leftrightarrow (y \leftrightarrow z) \iff (x \rightarrow (y \leftrightarrow z)) \land ((y \leftrightarrow z) \rightarrow x) \]
\[ \iff (x \rightarrow ((y \rightarrow z) \land (z \rightarrow y)) \land ((y \leftrightarrow z) \rightarrow x) \]
\[ \iff (x \rightarrow (y \rightarrow z)) \land (x \rightarrow (z \rightarrow y)) \land ((y \leftrightarrow z) \rightarrow x) \]
\[ \iff (\overline{x} \lor \overline{y} \lor z) \land (\overline{x} \lor \overline{z} \lor y) \land ((y \land z) \lor (\overline{y} \land \overline{z})) \rightarrow x) \]
\[ \iff (\overline{x} \lor \overline{y} \lor z) \land (\overline{x} \lor \overline{z} \lor y) \land ((y \land z) \lor (\overline{y} \land \overline{z})) \rightarrow x) \]
\[ \iff (\overline{x} \lor \overline{y} \lor z) \land (\overline{x} \lor \overline{z} \lor y) \land (\overline{y} \lor \overline{z} \lor x) \land (y \lor z \lor x) \]
• dates back to the 50’ies:
  1st version is *resolution based*
  second version splits space for time

• ideas:
  – eliminate the two cases of assigning a variable in space (1st version) or
  – case analysis in time, e.g. try $x = 0, 1$ in turn and recurse (2nd version)

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

• recent ($\leq 15$ years) optimizations:
  backjumping, learning, UIPs, dynamic splitting heuristics, fast data structures
  (we will have a look at each of them)
Resolution Rule

\[
\begin{align*}
C \cup \{v\} & \quad D \cup \{\neg v\} \\
\hline
\{v, \neg v\} \cap C = \{v, \neg v\} \cap D = \emptyset \\
C \cup D
\end{align*}
\]

Read:

resolving the two antecedent clauses \(C \cup \{v\}\) and \(D \cup \{\neg v\}\), both above the line, on the variable \(v\), results in the resolvent (clause) \(D \cup C\) below the line.
Eliminating Variables with Resolution

1. pick variable $x$

2. add all resolvents on $x$

3. remove all clauses with $x$ and $\bar{x}$

For instance given:

$$(a \lor b) \land (a \lor \bar{b}) \land (\bar{a} \lor b) \land (\bar{a} \lor \bar{b} \lor c) \land (\bar{a} \lor \bar{b} \lor \bar{c})$$

Resolvents on $a$:

$$\frac{(a \lor b) \land (\bar{a} \lor b)}{b} \quad \frac{(a \lor b) \land (\bar{a} \lor \bar{b} \lor c)}{\bar{b} \lor c} \quad \frac{(a \lor b) \land (\bar{a} \lor \bar{b} \lor \bar{c})}{\bar{b} \lor \bar{c}}$$

Remaining clauses after removing all clauses containing $a$ or $\bar{a}$:

$$b \land (\bar{b} \lor c) \land (\bar{b} \lor \bar{c})$$

Resolving on $b$ gives the remaining clauses

$$c \land \bar{c}$$

Which finally (resolving on $c$) gives the inconsistent empty clause

\textit{corresponds to eliminate a Tseitin variable for OR by distributivity}
Problems with Resolution Based DP

- if variables have many occurrences, then many resolvents are added
  - in the worst $x$ and $\neg x$ occur in half of the clauses ... 
  - ... then the number of clauses increases quadratically 
  - clauses become longer and longer

- unfortunately in real world examples the CNF explodes

- currently practically only useful
  - in the context of bounded variable elimination (preprocessing)
  - as in SatELite preprocessor [EénBiere05]
DPLL Procedure

\[ DPLL(F) \]

\[ F := BCP(F) \]

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

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

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

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

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

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

BCP

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

clauses

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

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

Decision

\[
\neg a
\]

\[
\neg b
\]

\[
\neg c
\]

\[
c
\]

\[
b
\]

\[
\neg c
\]

Search

\[
a = 1
\]

\[
b = 1
\]

\[
c = 0
\]
Conflict Driven Clause Learning (CDCL)

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

Decision a

Decision b

BCP c

Decision b

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

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

Decision: \( a \)

- \( \neg b \) BCP
- \( \neg c \) BCP

Learn: \( \neg a \)

Clauses:

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

Modern SAT Solvers (Part A) — Vienna Winter School on Verification — February 2012 — Armin Biere — JKU Linz
Conflict Driven Clause Learning (CDCL)

\[ \neg a \quad \text{BCP} \]

\[ \neg b \quad \text{BCP} \]

\[ \neg c \quad \text{decision} \]

\[ a = 1 \]

\[ b = 0 \]

\[ c = 0 \]

clauses

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

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

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

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

\[ a \lor b \lor c \]

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

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

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

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

\[ \neg a \]

\[ c \]

learn
Conflict Driven Clause Learning (CDCL)

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

\( a \) BCP
\( c \) BCP
\( b \) BCP

clauses
\( \neg a \lor \neg b \lor \neg c \)
\( \neg a \lor \neg b \lor c \)
\( \neg a \lor b \lor \neg c \)
\( \neg a \lor b \lor c \)
\( a \lor b \lor c \)
\( a \lor b \lor \neg c \)
\( a \lor b \lor c \)
\( \neg a \lor \neg b \)

learn

empty clause
BCP Example

```
Assignment

<table>
<thead>
<tr>
<th>Variables</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Clauses

<table>
<thead>
<tr>
<th>-1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>-2</td>
<td>3</td>
</tr>
<tr>
<td>-4</td>
<td>5</td>
</tr>
</tbody>
</table>
```

decision level  Control  Trail
Example cont.

Decide

<table>
<thead>
<tr>
<th>Variables</th>
<th>Assignment</th>
<th>Clauses</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\overline{X}$</td>
<td>1</td>
<td>$-1$</td>
</tr>
<tr>
<td>$X$</td>
<td>2</td>
<td>$-2$</td>
</tr>
<tr>
<td>$X$</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>$X$</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>$\overline{X}$</td>
<td>5</td>
<td>$-4$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>decision level</th>
<th>Control</th>
<th>Trail</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Modern SAT Solvers (Part A) — Vienna Winter School on Verification — February 2012 — Armin Biere — JKU Linz
Example cont.

Assign

<table>
<thead>
<tr>
<th>Decision level</th>
<th>Control</th>
<th>Trail</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Variables

<table>
<thead>
<tr>
<th>Assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
</tbody>
</table>

Clauses

<table>
<thead>
<tr>
<th>Clauses</th>
</tr>
</thead>
<tbody>
<tr>
<td>$-1$</td>
</tr>
<tr>
<td>$-2$</td>
</tr>
<tr>
<td>$-4$</td>
</tr>
</tbody>
</table>
Example cont.

BCP

<table>
<thead>
<tr>
<th>Assignment</th>
<th>Clauses</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1</td>
<td>−1 2</td>
</tr>
<tr>
<td>1 2</td>
<td>−2 3</td>
</tr>
<tr>
<td>1 3</td>
<td>−4 5</td>
</tr>
<tr>
<td>X 4</td>
<td></td>
</tr>
<tr>
<td>X 5</td>
<td></td>
</tr>
</tbody>
</table>
Example cont.

Decide

decision level

Control

Trail

Assignment

Clauses

Variables

Modern SAT Solvers (Part A) — Vienna Winter School on Verification — February 2012 — Armin Biere — JKU Linz
Example cont.

Assign

decision level

Control

Trail

Variables

Assignment

Clauses

Modern SAT Solvers (Part A) — Vienna Winter School on Verification — February 2012 — Armin Biere — JKU Linz
Decision Heuristics

- **static heuristics:**
  - one *linear* order determined before solver is started
  - usually quite fast to compute, since only calculated once
  - and thus can also use more expensive algorithms

- **dynamic heuristics**
  - typically calculated from number of occurrences of literals (in unsatisfied clauses)
  - could be rather expensive, since it requires traversal of all clauses (or more expensive updates in BCP)
  - effective *second order* dynamic heuristics (e.g. VSIDS in Chaff)
Cut Width Heuristics

- not really used in practice, but instructive to understand why SAT solvers might work

- view CNF as a graph:
  - clauses as nodes, edges between clauses with same variable

- a cut is a set of variables that splits the graph in two parts

- recursively find short cuts that cut off parts of the graph

- static or dynamically order variables according to the cuts

---

![Diagram of CNF as a graph with short cut and variable order](image)
int sat (CNF cnf)
{
  SetOfVariables cut = generate_good_cut (cnf);
  CNF assignment, left, right;

  left = cut_off_left_part (cut, cnf);
  right = cut_off_right_part (cut, cnf);

  forall_assignments (assignment, cut)
  {
    if (sat (apply (assignment, left)) && sat (apply (assignment, right)))
      return 1;
  }

  return 0;
}
Other popular Decision Heuristics

• Dynamic Largest Individual Sum (DLIS)
  
  – fastest dynamic *first order* heuristic (e.g. GRASP solver)
  
  – choose literal (variable + phase) which occurs most often (ignore satisfied clauses)
  
  – requires explicit traversal of CNF (or more expensive BCP)

• look-ahead heuristics (e.g. SATZ or MARCH solver)   **failed literals, probing**
  
  – do trial assignments and BCP for all unassigned variables (both phases)
  
  – if BCP leads to conflict, force toggled assignment of current trial decision
  
  – optionally learn binary clauses and perform equivalent literal substitution
  
  – decision: most balanced w.r.t. prop. assignments / sat. clauses / reduced clauses

  – see also our recent Cube & Conquer paper [HeuleKullmanWieringaBiere-HVC’11]
Exponential VSIDS (EVSIDS)

**Chaff** precision of score traded for faster decay

- increment score of involved variables by 1
- decay score of all variables every 256 conflicts by halving the score
- sort priority queue after decay and not at every conflict

**MiniSAT** uses EVSIDS

- also just update score of involved variables as actually LIS would also do
- dynamically adjust increment: \( \delta' = \delta \cdot \frac{1}{f} \) (typically increment \( \delta \) by 5%)
- use floating point representation of score
- “rescore” to avoid overflow in regular intervals
- EVSIDS linearly related to NVSIDS
Relating EVSIDS and NVSIDS

(consider only one variable)

\[ \delta_k = \begin{cases} 
1 & \text{if involved in } k\text{-th conflict} \\
0 & \text{otherwise} 
\end{cases} \]

\[ i_k = (1 - f) \cdot \delta_k \]

\[ s_n = (\ldots (i_1 \cdot f + i_2) \cdot f + i_3) \cdot f \cdots ) \cdot f + i_n = \sum_{k=1}^{n} i_k \cdot f^{n-k} = (1 - f) \cdot \sum_{k=1}^{n} \delta_k \cdot f^{n-k} \quad \text{(NVSIDS)} \]

\[ S_n = \frac{f^{-n}}{(1 - f)} \cdot s_n = \frac{f^{-n}}{(1 - f)} \cdot (1 - f) \cdot \sum_{k=1}^{n} \delta_k \cdot f^{n-k} = \sum_{k=1}^{n} \delta_k \cdot f^{-k} \quad \text{(EVSIDS)} \]
• observation:
  – recently added conflict clauses contain all the good variables of VSIDS
  – the order of those clauses is not used in VSIDS

• basic idea:
  – simply try to satisfy recently learned clauses first
  – use VSIDS to chose the decision variable for one clause
  – if all learned clauses are satisfied use other heuristics
  – intuitively obtains another order of localization (no proofs yet)

• mixed results as other variants VMTF, CMTF (var/clause move to front)