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

reencoding?

simplifying

inprocessing

search
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

- 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)
- Glueeminisat (2011)
- Contrasat (2011)
- Lingeling 587f (2011)
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

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

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) \\
\equiv (\neg a \land \neg b) \land h \lor \neg(\neg a \land \neg b) \land (\neg a \land g \lor a \land f) \iff a \land f \lor \neg a \land (b \land g \lor \neg b \land h)
SAT Example: Circuit Equivalence Checking

\[ b \lor a \land c \quad \Leftrightarrow \quad (a \lor b) \land (b \lor c) \]

equivalent?
**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* Turing-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.
Translation into CNF via NNF

**Negation Normal Form (NNF)**  AND/OR form + negations only occur in front of variables

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

\[
a \leftrightarrow (b \land a) \equiv (a \rightarrow (b \land a)) \land (a \leftarrow (b \land a))
\]

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

\[
\equiv (\overline{a} \lor (b \land a)) \land (a \lor (\overline{b} \land \overline{a})) \quad \text{in NNF}
\]

use distributivity of OR over AND  (“multiply out *outer* \lor”)

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

and simplify to finally obtain

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

unfortunately really expensive:

\[
(\land C_i) \lor (\land D_j) \equiv (C_i \lor D_j)
\]

\[O(n^2)\]
Example of Tseitin Transformation: Circuit to CNF

**CNF**

\[ \neg x \land (x \land a) \land \neg a \land \land (x \lor a) \land \land (x \lor c) \land \land (x \lor \neg a \lor \neg c) \land \ldots \]

\[ \neg y \land (y \land b) \land \neg b \land \land (y \lor b) \land \land (y \lor c) \land \land (y \lor \neg b \lor \neg c) \land \ldots \]

\[ o \land (x \leftrightarrow a \land c) \land \neg (x \leftrightarrow a \land c) \land \land (y \leftrightarrow b \lor x) \land \land (y \leftrightarrow b \lor x) \land \land (u \leftrightarrow a \lor b) \land \land (u \leftrightarrow a \lor b) \land \land (v \leftrightarrow b \lor c) \land \land (v \leftrightarrow b \lor c) \land \land (w \leftrightarrow u \land v) \land \land (w \leftrightarrow u \land v) \land \land (o \leftrightarrow y \oplus w) \land \land (o \leftrightarrow y \oplus w) \land \ldots \]
Tseitin Transformation: Input / Output Constraints

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

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

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

**Equivalence:** \[ x \leftrightarrow (y \leftrightarrow z) \iff (x \rightarrow (y \leftrightarrow z)) \land ((y \leftrightarrow z) \rightarrow x) \]
\[ \iff (x \rightarrow ((y \rightarrow z) \land (z \rightarrow y)) \land ((y \leftrightarrow z) \rightarrow x) \]
\[ \iff (x \rightarrow (y \rightarrow z)) \land (x \rightarrow (z \rightarrow y)) \land ((y \leftrightarrow z) \rightarrow x) \]
\[ \iff (\bar{x} \lor \bar{y} \lor z) \land (\bar{x} \lor \bar{z} \lor y) \land ((y \leftrightarrow z) \rightarrow x) \]
\[ \iff (\bar{x} \lor \bar{y} \lor z) \land (\bar{x} \lor \bar{z} \lor y) \land ((y \land z) \lor (\bar{y} \land \bar{z}) \rightarrow x) \]
\[ \iff (\bar{x} \lor \bar{y} \lor z) \land (\bar{x} \lor \bar{z} \lor y) \land (y \land z \rightarrow x) \land (\bar{y} \land \bar{z} \rightarrow x) \]
\[ \iff (\bar{x} \lor \bar{y} \lor z) \land (\bar{x} \lor \bar{z} \lor y) \land (\bar{y} \lor \bar{z} \lor x) \land (y \lor z \lor x) \]
• dates back to the 50’ies:
  1\textsuperscript{st} version is \textit{resolution based}

  second version splits space for time

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

$$C \cup \{v\} \quad D \cup \{\neg v\}$$

\[ \frac{\{v, \neg v\} \cap C = \{v, \neg v\} \cap D = \emptyset}{C \cup D} \]

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.
1. pick variable $x$

2. add all resolvents on $x$

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

For instance given:

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

Resolvents on $a$:

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

Remaining clauses after removing all clauses containing $a$ or $\neg a$:

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

Resolving on $b$ gives the remaining clauses $c \land \neg c$

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

**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 case $x$ 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 satisfiable

if \( \bot \in F \) return unsatisfiable

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

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

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

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

Clauses:

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

Learned clause:

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

Decision: $a$

Clauses:

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

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)

$a = 1$

$b = 0$

$c = 0$

clauses

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

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

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

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

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

\[ \neg a \]

\[ c \]

learn

empty clause
### Simple Data Structures in DP Implementation

#### Variables

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

#### Clauses
BCP Example

Decision level: 0

Control:

Trail:

Assignment:

Variables:

Clauses:

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

Decide

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

-1 2
-2 3
-4 5
```

TrailControldecision level

ClausesVariables
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
Decide

decision level

Control

Trail

Variables

Assignment

Clauses

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

Assign

\[
\begin{array}{c|c|c}
\text{decision level} & \text{Control} & \text{Trail} \\
\hline
2 & 3 & 4 \\
& 0 & 3 \\
& 0 & 2 \\
& & 1 \\
\end{array}
\]

Variables

<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></td>
</tr>
<tr>
<td>1 4</td>
<td>-4 5</td>
</tr>
<tr>
<td>X 5</td>
<td></td>
</tr>
</tbody>
</table>
Example cont.

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 of parts of the graph

- static or dynamically order variables according to the cuts

\[ ... -1 \quad 2 \quad 3 \quad -2 \quad 1 \quad -3 \quad 1 \quad 3 \quad -4 \quad ... \]

Assume no occurrences of 1, 2, -1, -2 on the right side.

Short cut
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)

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

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

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

\[
S_n = \frac{f^{-n}}{(1 - f)} \cdot s_n = \frac{f^{-n}}{(1 - f)} \cdot (1 - f) \cdot \sum_{k=1}^{n} \delta_k \cdot f^{n-k} = \sum_{k=1}^{n} \delta_k \cdot f^{-k}
\] (EVSIDS)
BerkMin’s Dynamic Second Order Heuristics

[GoldbergNovikov-DATE’02]

• 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)