Introduction to Satisfiability Solving with Practical Applications

Niklas Een
SAT solvers

Inner workings
The SAT problem

A literal $p$ is a variable $x$ or its negation $\neg x$.

A clause $C$ is a disjunction of literals: $x_2 \lor \neg x_{41} \lor x_{15}$

A CNF is a conjunction of clauses:

$$(x_2 \lor \neg x_{41} \lor x_{15}) \land (x_6 \lor \neg x_2) \land (x_{31} \lor \neg x_{41} \lor \neg x_6 \lor x_{156})$$

The SAT-problem is:

- Find a boolean assignment
- such that each clause has a true literal

First problem shown to be NP-complete (1971)
What’s a clause?

A clause of size \( n \) can be viewed as \( n \) propagation rules:

\[
a \lor b \lor c
\]

is equivalent to:

\[
(\neg a \land \neg b) \rightarrow c
\]
\[
(\neg a \land \neg c) \rightarrow b
\]
\[
(\neg b \land \neg c) \rightarrow a
\]

**Example:** Consider the constraint

\[
t = \text{AND}(x, y)
\]

\[
\neg t \land y \rightarrow \neg x
\]
Example

\{3, 6, -7, 8\} \rightarrow \{3, 6, -7, 8\} \rightarrow \{3, 6, -7, 8\} \rightarrow ...  
\{1, 4, 7\} \rightarrow \{1, 4, 7\} \rightarrow \{1, 4, 7\} \rightarrow ...

\{-8, 4\} \rightarrow \{-8, 4\} \rightarrow \{-8, 4\} \rightarrow ...  
\{-1, -3, 8\} \rightarrow \{-1, -3, 8\} \rightarrow \{-1, -3, 8\} \rightarrow ...

\{-3, -4, -8\} \rightarrow \{-3, -4, -8\} \rightarrow \{-3, -4, -8\} \rightarrow ...

\{-1, -2, 3, 4, -6\} \rightarrow \{-1, -2, 3, 4, -6\} \rightarrow \{-1, -2, 3, 4, -6\} \rightarrow ...

\{3, 6, -7, 8\} \rightarrow \{3, 6, -7, 8\} \rightarrow \{3, 6, -7, 8\} \rightarrow ...  
\{1, 4, 7\} \rightarrow \{1, 4, 7\} \rightarrow \{1, 4, 7\} \rightarrow ...

\{-8, 4\} \rightarrow \{-8, 4\} \rightarrow \{-8, 4\} \rightarrow ...  
\{-1, -3, 8\} \rightarrow \{-1, -3, 8\} \rightarrow \{-1, -3, 8\} \rightarrow ...

\{-3, -4, -8\} \rightarrow \{-3, -4, -8\} \rightarrow \{-3, -4, -8\} \rightarrow ...

\{-1, -2, 3, 4, -6\} \rightarrow \{-1, -2, 3, 4, -6\} \rightarrow \{-1, -2, 3, 4, -6\} \rightarrow ...

CONFLICT!  
(backtrack)
Search Components

Decision heuristic

- Static \((x_1, x_2, x_3 \ldots)\)
- State based
  - Shortest non-satisfied clause, most common literal etc.
- History based
  - Pick variables that lead to conflicts in the past.

Propagation

Backtracking

Search Tree

\[ x_4 = 1 \]
\[ x_1 = 1 \]
\[ x_3 = 1 \]
\[ x_1 = 0 \]
\[ \Rightarrow x_8 = 0 \]
\[ \Rightarrow x_8 = 0 \]

Conflict

Satisfying Assignment

\[ Satisfying \ Assignment \]
Search Components

Decision heuristic

Propagation
- Unit propagation ("BCP")
- Unate propagation
- Probing/Dilemma
- Equivalence classes

Backtracking

Unate (pure literal)

\[
\begin{align*}
\{3, 6, -7, 8\} \\
\{1, 4, 7\} \\
\{-8, 4\} \\
\{-1, -3, 8\} \\
\{-3, -4, -8\} \\
\{-1, -2, 3, 4, -6\}
\end{align*}
\]
Search Components

Decision heuristic

Propagation

Backtracking

- Flip last decision (standard recursive backtracking)
- Conflict analysis:
  - Learn an asserting clause
  - [...]

May be expressed in any variables, not just decisions.
Must have only one variable from the last decision level.

Asserting clause: \{\neg a, \neg b, c\}

What if b was irrelevant?
Search Components

Decision heuristic

Propagation

Backtracking

- Flip last decision
  (standard recursive backtracking)

- Conflict analysis:
  - Learn an asserting clause
  - Backjumping
  - No recursion
  - Can be viewed as a resolution strategy, guided by conflicts.
  - Together with variable activity, most important innovation.

```dpll(assign){
    "do BCP";
    if "conflict": return FALSE;
    if "complete assign": return TRUE;
    "pick decision variable x";
    return dpll(assign[x=0]) || dpll(assign[x=1]);
}
```

```forever{
    "do BCP"
    if "no conflict":
        if "complete assign": return TRUE;
        "pick decision x=0 or x=1";
    else:
        if "at top-level": return FALSE;
        "analyze conflict"
        "undo assignments"
        "add conflict clause"
}
```
Conflicting clause:
\{\neg x10587, \neg x10592, \neg x10588\}

One option:
- Trace back to decision variables
- Would learn:
  \{x10646, x9444, \neg x10373, \neg x10635, \neg x10637\}

Other option:
- Stop earlier
- Asserting if only one literal left at the highest decision level
- Keep expanding nodes from that level
Conflict Analysis – Resolution View

### Decision Table

<table>
<thead>
<tr>
<th>Decision</th>
<th>Implications</th>
</tr>
</thead>
<tbody>
<tr>
<td>¬a</td>
<td>–</td>
</tr>
<tr>
<td>¬b</td>
<td>c</td>
</tr>
<tr>
<td>¬d</td>
<td>e, ¬f</td>
</tr>
</tbody>
</table>

### Clause Database

- `reason for c`
- `reason for ¬f`
- `reason for e`

**conflicting clause**

### Conflict Clause Minimization:
Continue to resolve if result is a strict subset

### Start with the conflicting clause

- `{f, ¬e, ¬c}`

### Resolve with reason of last assigned literal

- `{¬f, ¬e, d, b}`

### Keep resolving until only one literal of last decision level
- `{¬e, d, ¬c, b}`
- `{e, d, ¬c, b, a}`

### Resolve on f

- `{¬e, d, ¬c, b}`
- `{e, d, ¬c, b, a}`

### Resolve on e

- `{d, ¬c, b, a}`

### Done!

### Resolution:

{\(x, A\) res. \(¬x, B\) = \(A, B\)}

**blue** = last decision level

**Or not?**
Variable Activity

The VSIDS activity heuristic:
- Bump literals of the learned (conflict) clause
- Decay by halving activity periodically

Modified activity heuristic:
- Bump variables of all clauses participating in analysis
- Decay after each conflict

Effect:
- Give preference to the very latest conflicts (Berkmin/VMTF)
- Longer memory (15000 decays before minimal float value)
Execution of CDCL Solver

- **Green** – Activity of decision variable
- **Red** – Length of learned clause
- **Yellow** – Decision depth when conflict occurred
Other Techniques

Two watched literals
- not moved during backtrack;
- migrate to silent places
- improves with length of clauses
- most BCP in learned clauses (often 90%), which are long

Restarts with polarity memoization
- frequent restarts, except sometimes: 1, 1, 2, 1, 2, 4, 1, 1, 2, 1, 2, 4, 8...
- not real restarts
- compresses assignment stack => more focus on active variables

Conflict-clause deletion
- remove clauses that don’t participate in conflict analysis
- handles subsumed clauses better than original scheme (based on length)

CNF preprocessing
- variable elimination
- subsumption, self-subsuming resolution
Other Techniques (cont.)

Better CNF generation
- If problem on circuit form:
  - Technology mapping for CNF
  - Fanout aware variable elimination
- Certain constraints (e.g. cardinality constraints) have known efficient encodings.

Improvements to incremental SAT
- Domain specific adjustments

<table>
<thead>
<tr>
<th>Method</th>
<th>Approx. #conflicts (Characteristics)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BMC</td>
<td>100</td>
</tr>
<tr>
<td>Interpolation</td>
<td>1,000 (clause deletion, proof logging)</td>
</tr>
<tr>
<td>PDR</td>
<td>10,000 (local problems, limited proof logging)</td>
</tr>
<tr>
<td>SAT-sweeping</td>
<td>100,000 (local problems)</td>
</tr>
</tbody>
</table>
Practical SAT is an experimental science.

There are three types of papers:
- The conclusion is wrong.
- The conclusion is correct, but not for the stated reasons.
- The conclusion is correct, the stated reasons are valid, but the experimental data does not support it.

It is hard to improve the CDCL algorithm.
Applying SAT solvers
Solving puzzles
Slither Link

Rules

1. Each number must be surrounded by that many edges.
2. All edges must form a single closed loop.
Slither Link

Rules

1. Each number must be surrounded by that many edges.
2. All edges must form a single closed loop.

Constraints

A. Rule 1 is easily expressed:
   - Let $e_1, e_2, e_3, e_4$ be the edges around a number $k$.
   - Encode in CNF: $\text{card}(e_1, e_2, e_3, e_4) = k$

B. An approximation of rule 2 can be enforced locally:
   - Every crossing should have either zero or two edges.
   - Encode as: $\text{card}(e_1, e_2, e_3, e_4) = 0 \text{ or } 2$

Example. $k = 1$:
\[
\{e_1, e_2, e_3, e_4\}, \quad \{-e_1, -e_2\}, \quad \{-e_1, -e_3\}, \quad \{-e_1, -e_4\}, \\
\{-e_2, -e_3\}, \quad \{-e_2, -e_4\}, \quad \{-e_3, -e_4\}
\]
Slither Link (cont.)

Lets run it...
...close, but no cigar.

But with a CEGAR!

Refine by prohibiting these particular cycles.

Repeat
Repeat
Done!
Incremental solution works well for larger sizes too.

**Exercise:** Formulate a SAT encoding that will solve *Slither Link* non-incrementally (one SAT call only).
Other nice puzzles

Heyawake
Hanjie
Kakuro
Reflections

...try one with SAT

http://games.erdener.org/laser/
Applying SAT solvers

Verification
Incremental SAT

MiniSat API

- void addClause(Vec<Lit> clause)
- bool solve(Vec<Lit> assumps)
- bool readModel(Var x) \(\quad \text{– for SAT results}\)
- bool assumpUsed(Lit p) \(\quad \text{– for UNSAT results}\)

The method \textit{solve()} treats the literals in \textit{assumps} as unit clauses to be temporary assumed during the SAT-solving.

More clauses can be added after \textit{solve()} returns, then incrementally another SAT-solving executed.
Allows for...

Refinement loop
- More clauses can be added with \texttt{addClause()}

Restricted clause deletion
- Clauses can be tagged by an \textit{activation literal} "a":
  \[
  \{\neg a, p_0, p_1, \ldots, p_n\}, \{\neg a, q_0, q_1, \ldots, q_m\}, \ldots
  \]
- Activated by passing \texttt{a} as part of \texttt{assumps} to \texttt{solve()}
- Deleted by \texttt{addClause}\{\neg a\}

Poor-mans proof logging
- If we have several sets of clauses $A_1, A_2, \ldots$ with different activation literals \texttt{a}_1, \texttt{a}_2, \ldots, \texttt{assumpUsed()} tells us which sets were used for proving UNSAT
- Also works for output of cones of logic in a circuit
Bit-level Verification

Design is given as a netlist of:
- AND gates
- PIs
- Flops

Wires can be complemented. A special output is marked as the property.
Bounded Model Checking

Unroll the design for 1, 2, 3, etc. time-frames. Check if the property can fail in the last frame.

for $k$ in $1..\infty$:
    $p_{bad} = \text{CNF}(\text{logic cone of } \text{Bad}_k)$
    if (solve($\{p_{bad}\}$))
        return CounterExample
    addClause($\{\neg p_{bad}\}$)

Questions

- Why grow trace ”forward”?
- Increase by more than one frame at a time?
- How about SAT preprocessing?
- Better just skip incremental SAT?
Conclusions

• SAT-solvers are implication engines.

• Clauses are the "assembly language" of propositional reasoning.

• Two important techniques of CDCL solvers are:
  • Conflict analysis
  • Variable activity

• Most applications use incremental SAT and encode an abstraction of the real problem.