Practical SAT Solving: Achievements, Problems, Opportunities

- Part 1. Modern resolution based SAT-solvers. What are they “made of”? Why do they work so well?
- Part 2. Linear size resolution proofs that can not be found efficiently (equivalence checking of combinational circuits)
- Part 3. Testing the satisfiability of a formula by computing a stable set of points.
Summary (we are here ↓)

• PART 1
  • Introduction
    • The Satisfiability Problem (SAT)
      • Why is SAT in Such High Demand?
    • DP, DPLL and Resolution
    • Modern SAT-solvers
  • .....
A literal is a Boolean function of one Boolean variable.
The identity function of a variable $x_i$ is called the positive literal of $x_i$ (denoted just by $x_i$).
The negation function of a variable $x_i$ is called the negative literal of $x_i$ (denoted by $\sim x_i$).
A disjunction of literals is called a clause:
For example, \( C = x_1 \lor x_5 \lor \sim x_3 \)

A CNF formula is a conjunction of clauses:
\[
F = (x_1 \lor x_5 \lor \sim x_3) \land (\sim x_7 \lor \sim x_2) \land (x_2 \lor x_3 \lor x_8)
\]
More About Clauses

We assume that a clause has at most one literal of a variable.

\[ x_1 \lor x_5 \lor x_1 \lor x_{10} \Rightarrow x_1 \lor x_5 \lor x_{10} \]
\[ x_1 \lor x_5 \lor \neg x_1 \lor x_{10} \Rightarrow 1 \text{ (tautologous clause)} \]

A clause without literals is called an empty clause. Denote it by \( \Lambda \) (capital lambda). Clause \( \Lambda \) can not be satisfied.

A clause with only one literal is called a unit clause. A unit clause can be satisfied only by setting its only literal to 1.
Assignment

Let $X$ be a set of Boolean variables.

**A complete assignment** to the variables of $X$ is a mapping $X \rightarrow \{0,1\}$.

A mapping $X' \rightarrow \{0,1\}$ where $X' \subseteq X$ is called a partial assignment.

**Example:**

$X=\{x_1, x_2, x_3, x_4, x_5\}$

complete assignment \quad $y = (x_1=0, x_2=1, x_3=0, x_4=1, x_5=1)$

partial assignment \quad $y = (x_1=0, x_3=0, x_5=0)$
Satisfiability Problem (SAT)

A clause is satisfied (falsified) by an assignment \( y \) if \( C(y) = 1 \) (respectively 0).

The satisfiability problem is as follows. Given a set of clauses \( F \), find an assignment satisfying every clause of \( F \) (and so setting \( F \) to 1) or prove that such an assignment does not exist.

An assignment satisfying the clauses of \( F \) (if any) is called a satisfying assignment.
Summary (we are here ↓)

• PART 1
  • Introduction
    • The Satisfiability Problem (SAT)
    • Why Is SAT in Such High Demand?
  • DP, DPLL and Resolution
  • Modern SAT-solvers
• .....
Main Attractions of SAT

- Many practical problems from various domains reduce to SAT:

  logic synthesis, manufacturing test generation, model checking, circuit routing, equivalence checking, software verification and so on.

- In the last few years considerable progress has been made

- Current SAT-solvers can handle practical formulas with millions of variables and clauses
Reducing Circuit Satisfiability to SAT

Network $N$

$g_1$: $y_1 = \text{AND}(x_1, x_2)$,
$g_2$: $y_2 = \text{AND}(x_2, x_3)$,
$g_3$: $z = \text{OR}(y_1, y_2)$

$F(g_1) = C_1 \land C_2 \land C_3$,
$C_1 = \sim x_1 \lor \sim x_2 \lor y_1$,
$C_2 = x_1 \lor \sim y_1$,
$C_3 = x_2 \lor \sim y_1$

$F(N) = F(g_1) \land F(g_2) \land F(g_3)$,
$F(N) \land C$, $C = z$
Universality Also Causes Problems

From a practical viewpoint, the most attractive property of SAT is its **universality**

From a theoretical viewpoint, universality of SAT is the “cause” of its complexity (e.g. one can describe the behavior of a non-deterministic Turing machine by a set of clauses (Cook’s theorem))

So, as usual, “a sin” (**inefficiency of SAT**) is just an extension of “a virtue” (**universality of SAT**)
Summary (we are here ↓)

• PART 1
  • Introduction
  • DP, DPLL and Resolution
    • Resolution Proof System (RPS)
    • DP procedure
    • DPLL
    • Some Theoretical Results About RPS
    • Extended RPS
      • Modern SAT-solvers
  • .....
Resolution Operation

Let $C_1 = x_2 \lor x_5 \lor x_{10}$ and $C_2 = x_2 \lor \neg x_5 \lor x_{12}$ be two clauses having exactly one pair of opposite literals.

The clause $C = x_2 \lor x_{10} \lor x_{12}$ is called the resolvent of $C_1$ and $C_2$ obtained by resolving $C_1$ and $C_2$ in variable $x_5$.

Clause $C$ is said to be obtained from $C_1$ and $C_2$ by the resolution operation.

The main property of the resolution operation:

$C_1 \land C_2 \rightarrow C$,

i.e. if $C_1$ and $C_2$ are satisfied, the resolvent is satisfied too.
Resolution Operation (continued)

If $C_1$ and $C_2$ have more than one pair of clashing literals or do not have clashing literals at all, potential resolvents are “trivial”

$C_1 = x_1 \lor x_3$, $C_2 = \neg x_1 \lor \neg x_3$. By resolving $C_1$ and $C_2$ in variable $x_1$ we obtain a tautologous clause $x_3 \lor \neg x_3$

If $C_1$ and $C_2$ have no clashing literals one can only produce a trivial clause implied by $C_1$ and $C_2$ separately

$C_1 = x_1 \lor x_3$, $C_2 = x_3 \lor x_5$. No variable to resolve in. By combining the literals of $C_1$ and $C_2$ we obtain clause $x_1 \lor x_3 \lor x_5$ implied by $C_1$ and $C_2$ separately
Resolution Proof System (RPS)

Introduced for first-order logic by A. Robinson (1965)

RPS consists only of the resolution operation

Given a CNF formula $F$, a proof of unsatisfiability of $F$ in RPS is a sequence of resolution operations resulting in derivation of an empty clause

This sequence is called a resolution proof of unsatisfiability of $F$
Example

\( F = (x_1 \lor \sim x_2) \land (\sim x_1 \lor \sim x_2) \land x_2 \)

- By resolving \((x_1 \lor \sim x_2)\) and \((\sim x_1 \lor \sim x_2)\) in \(x_1\) we produce the clause \(\sim x_2\).

- By resolving \(\sim x_2\) and \(x_2\) we produce \(\land\) (an empty clause)

The resolution proof above consists of two resolution operations.
Resolution Graph

- Rectangles are original clauses
- Ovals are resolvents
- Variables in which clauses are resolved are shown in red

- In general, a resolution graph is a **DAG**
- A special case of RPS, is **tree-like RPS** (resolution graph is a tree)
RPS Is Sound

RPS is sound:

• The resolvent of $C_1$ and $C_2$ is implied by $C_1 \land C_2$.

• Since an empty clause $C$ is deduced from clauses of $F$ (and their descendants) only by resolution operations, then $C$ is implied by $F$.

• Since an empty clause cannot be satisfied, then $F$ unsatisfiable
RPS Is Complete

Given an unsatisfiable formula $F$, one can always prove its unsatisfiability by building a binary search tree.

Nodes of this tree are assigned variables of $F$ that were split on, leaves are assigned clauses that were falsified.

A binary search tree can be simulated by a tree-like RPS (a special case of RPS). So for any unsatisfiable formula there is a proof in RPS.
PART 1
• Introduction
• DP, DPLL and Resolution
  • Resolution Proof System (RPS)
  • DP procedure
• DPLL
  • Some Theoretical Results About RPS
• Extended RPS
  • Modern SAT-solvers
• .....
Existential Quantification

Let $f(x_1,\ldots,x_n)$ be a Boolean function of $n$ variables.

The function

$$f^*(\ldots,x_i=0,\ldots) \lor f(\ldots,x_i=1,\ldots)$$

is existential quantification of $f$ in variable $x_i$.

$f = 1$ iff $f^* = 1$. That is function $f$ is satisfiable iff $f^*$ is satisfiable.
Davis-Putnam Procedure

Introduced in 1960 (5 years before Robinson’s resolution)

Let $F$ be a CNF formula and $X={x_1,\ldots,x_n}$ be its set of variables.

\[
\begin{align*}
DP(F) \\
X^* = X, \; F^* = F \\
\text{while} \; (X^* \text{ is not empty}) \\
\quad \{x_i = \text{pick\_variable}(X^*); \; X^* = X^* \setminus \{x_i\} \\
\quad F^* = \text{existentially\_quantify}(F^*,x_i)\}
\end{align*}
\]

if ($F^*$ is 0) return (‘unsatisfiable’)
if ($F^*$ is 1 ) return (‘satisfiable’)

\[
\begin{align*}
DP(F) \\
X^* = X, \; F^* = F \\
\text{while} \; (X^* \text{ is not empty}) \\
\quad \{x_i = \text{pick\_variable}(X^*); \; X^* = X^* \setminus \{x_i\} \\
\quad F^* = \text{existentially\_quantify}(F^*,x_i)\}
\end{align*}
\]
Quantification of CNF

Let $F(x_1, \ldots, x_n)$ be a CNF formula. $F = F_s \land F_{x_i} \land F_{\neg x_i}$

$F_s$ consists of the clauses of $F$ independent of $x_i$,
$F_{x_i}$ (respectively $F_{\neg x_i}$) consists of the clauses of $F$ having literal $x_i$ (respectively $\neg x_i$)

$F(..., x_i=0, \ldots) \lor F(..., x_i=1, \ldots) = (F_s \land F_{x_i}) \lor (F_s \land F_{\neg x_i}) = F_s \land (F_{x_i} \lor F_{\neg x_i})$

$$(C_1 \land C_2) \lor (C_3 \land C_4) = (C_1 \lor C_3) \land (C_1 \lor C_4) \land (C_2 \lor C_3) \land (C_2 \lor C_4)$$

Let $F_{x_i} = C'_1 \land C'_2 \ldots C'_k$ and $F_{\neg x_i} = C''_1 \land C''_2 \ldots C''_m$

$F_{x_i} \lor F_{\neg x_i} = (C'_1 \lor C''_1) \land \ldots \land (C'_1 \lor C''_m) \land$

$(C'_2 \lor C''_1) \land \ldots \land (C'_2 \lor C''_m) \land$

$(C'_k \lor C''_1) \land \ldots \land (C'_k \lor C''_m)$
Davis and Puttnam Procedure in Terms of Resolution

\[
DP(F) \\
X^* = X, \ F^* = F \\
\text{while } (X^* \text{ is not empty}) \\
\quad \{x_i = \text{pick\_variable}(X^*); \ X^* = X^* \setminus \{x_i\} \} \\
\quad F^* = \text{add\_resolvents}(F^*, x_i) \ // \text{resolve the clauses of } F^* \text{ in } x_i \\
\quad F^* = \text{rem\_clauses}(F^*, x_i) \} // \text{remove the clauses of } F^* \text{ with variable } x_i \\
\text{if } (F^* == 0) \text{ return } (\text{‘unsatisfiable’}) \\
\text{if } (F^* == 1) \text{ return } (\text{‘satisfiable’})
\]
DP Procedure Generates Huge Number of Clauses

unsat CNF formula, 20 variables, 85 clauses, time=0.2s

<table>
<thead>
<tr>
<th>#iter</th>
<th>2</th>
<th>4</th>
<th>6</th>
<th>8</th>
<th>10</th>
<th>12</th>
<th>14</th>
<th>16</th>
<th>18</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>cnf size</td>
<td>116</td>
<td>309</td>
<td>628</td>
<td>595</td>
<td>532</td>
<td>238</td>
<td>106</td>
<td>24</td>
<td>5</td>
<td>1</td>
</tr>
</tbody>
</table>

unsat CNF formula, 30 variables, 127 clauses, time=9.1s

<table>
<thead>
<tr>
<th>#iter</th>
<th>3</th>
<th>6</th>
<th>9</th>
<th>12</th>
<th>15</th>
<th>18</th>
<th>21</th>
<th>24</th>
<th>27</th>
<th>30</th>
</tr>
</thead>
<tbody>
<tr>
<td>cnf size</td>
<td>162</td>
<td>431</td>
<td>1212</td>
<td>2126</td>
<td>3151</td>
<td>1995</td>
<td>886</td>
<td>151</td>
<td>14</td>
<td>1</td>
</tr>
</tbody>
</table>

unsat CNF formula, 35 variables, 42 clauses, time=4585.6s

<table>
<thead>
<tr>
<th>#iter</th>
<th>4</th>
<th>8</th>
<th>13</th>
<th>16</th>
<th>19</th>
<th>22</th>
<th>27</th>
<th>35</th>
</tr>
</thead>
<tbody>
<tr>
<td>cnf size</td>
<td>458</td>
<td>2558</td>
<td>22025</td>
<td>32489</td>
<td>19531</td>
<td>7696</td>
<td>615</td>
<td>1</td>
</tr>
</tbody>
</table>
Summary (we are here ↓)

- PART 1
  - Introduction
  - DP, DPLL and Resolution
    - Resolution Proof System (RPS)
    - DP procedure
  - DPLL
    - Some Theoretical Results About RPS
    - Extended RPS
  - Modern SAT-solvers
- .....
Restriction of CNF

Let $F(x_1,\ldots,x_n)$. Let $A$ be an assignment (partial or complete) to variables of $X=\{x_1,\ldots,x_n\}$.

Denote by $F_A$ the CNF formula obtained from $F$ by
a) Removing all the clauses of $F$ that are satisfied by $A$

b) Removing from the unsatisfied clauses of $F$ all the literals that are set to 0 by $A$

$F_A$ is said to be the formula $F$ restricted by $A$ or just a restriction of $F$ for short.
DPLL procedure

// Returns ‘sat’ if $F$ is satisfiable
// Returns ‘unsat’ if $F$ is unsatisfiable
// Initially $A = \emptyset$, $F_A = F$

\[
DPLL(F_A, A) \\
\{ \text{if (satisfied}(F_A, A) \text{ return ‘sat’}, A; } \\
\quad \text{if (pure\_literal}(F_A, A)) \quad (F_A, A) \leftarrow apply\_the\_rule(F, A); \\
\quad \text{if (unit\_clauses}(F_A, A)) \quad (F_A, A) \leftarrow run\_BCP(F_A, A); \\
\quad \text{if (falsified}(F_A, A) \text{ return ‘unsat’}; \\
\quad x_i = pick\_variable(F_A, A); \\
\quad A0 = A \cup \{(x_i=0)\}, A1 = A \cup \{(x_i=1)\} \\
\text{if ((DPLL}(F_{A0}, A0) == ‘unsat’) and \\
\quad (DPLL(F_{A1}, A1)) == ‘unsat’) \\
\text{return ‘unsat’} \\
\text{else return ‘sat’}, A} \\
\]
Example

\[ F = (x_1 \lor x_2) \land (x_1 \lor \neg x_2) \land (\neg x_1 \lor x_3) \land (\neg x_1 \lor \neg x_3) \]

\[ x_1 = \text{pick\_assignment}(F, \emptyset), \]
\[ \text{make\_assignment}(F, \emptyset, x_1) \]

\[ A = \{(x_1=0)\} \]
\[ F(x_1=0) = x_2 \lor \neg x_2 \]

\[ A = \{(x_1=1)\} \]
\[ F(x_1=1) = x_3 \lor \neg x_3 \]

\[ \text{run\_BCP}(F,A) \text{ produces a falsified clause} \]
\[ \text{DPLL returns ‘unsat’} \]
Pure Literal Rule

Let $F(x_1,..,x_n)$ be a CNF formula. Suppose that no clause of $F$ has the negative literal of $x_i$. (That is if a clause of $F$ has a literal of $x_i$ it is always positive.)

$F(x_1,..,x_i=1,..,x_n)=0$ implies $F(x_1,..,x_i=0,..,x_n)=0$.

Pure literal rule:

If the current CNF formula has literals of only one polarity of $x_i$, one can set $x_i$ to the value satisfying the clauses having literals of $x_i$.

(Rarely used in current SAT-solvers)
Boolean Constraint Propagation (BCP)

If the current CNF formula has a unit clause $C$, it can only be satisfied by setting the only literal of $C$ to 1.

```
run_BCP(F,A)
{ while (success)
  success = false;
  for every unsat clause $C_i$ of $F$
    {if (empty($C_i$)) return ‘unsat’
     if (unit($C_i$))
      {success = true;
       (F,A) ← make_assignment(F,$C_i$,A);}}}
```
Example

\[ F = (x_1 \lor x_2) \land (\neg x_1 \lor \neg x_2) \land (\neg x_2 \lor x_3) \land (x_1 \lor \neg x_2 \lor \neg x_3) \]

\[ A = \{(x_1 = 0)\}, \quad F_A = x_2 \land (\neg x_2 \lor x_3) \land (\neg x_2 \lor x_3) \]

Applying \textit{run\_BCP}(F,A)

1) Making assignment \( x_2 = 1 \) to satisfy unit clause \( x_2 \):
   \[ A = \{(x_1 = 0), (x_2 = 1)\}, \quad F_A = x_3 \lor \neg x_3 \]

2) Making assignment \( x_3 = 1 \) to satisfy clause \( x_3 \):
   \[ A = \{(x_1 = 0), (x_2 = 1), (x_3 = 1)\}, \quad F_A = \Lambda \]

3) Since \( F_A \) contains an empty clause, \textit{run\_BCP}(F,A) returns ‘unsat’
DPLL and Tree-like RPS

• For unsatisfiable formulas, pure literal rule can be ignored
• Satisfying a unit clause can be represented as branching with an immediate conflict in one branch
• Proof generated by DPLL can be represented as a binary search tree.
• Each leaf of this tree corresponds to a falsified clause
• The corresponding resolution proof is obtained by resolving clauses in branching variables moving from leaves to the root
Summary (we are here ↓)

• PART 1
  • Introduction
  • DP, DPLL and Resolution
    • Resolution Proof System (RPS)
    • DP Procedure
    • DPLL
    • Some Theoretical Results About RPS
    • Extended RPS
  • Modern SAT-solvers
  • .....
Exponential Lower Bounds of Resolution

In 1968, Tseitin proved superpolynomial lower bound for regular RPS (in regular RPS only one variable is resolved along any path from a source to the sink of the resolution graph). He used CNF formulas capturing the fact that for any graph the sum of degrees of the vertexes is even.

In 1985, Haken proved superpolynomial lower bound for RPS. He used CNF formulas capturing the pigeon-hole principle: $n+1$ pigeons can not be allocated in $n$ holes so that no two pigeons share a hole.
Let $K$ be a class of unsatisfiable CNF formulas. Suppose the length (number of literals) of each clause of a CNF $F$ from $K$ is limited by a constant.

The width of a proof $R$ that $F$ is unsatisfiable is the length of the longest clause appearing in $R$. Ben Sasson and Wigderson showed in 1999 that if CNFs from $K$ have polynomial proofs in RPS, there always exist narrow proofs of width bounded by $\sqrt{n}$.

So to show that a class of formulas (consisting of clauses of bounded length) has only superpolynomial proofs in RPS, it suffices to show that in any resolution proof a “long” resolvent will be generated.
Exponential Separation of Tree-like RPS and RPS


Exponential separation of tree-like RPS and RPS. There exists a class of CNF formulas with polynomial (actually linear) proofs in RPS and exponential proofs in tree-like RPS

Alekhnovich, Johannsen 2002

Exponential separation of regular RPS and RPS
Summary (we are here ↓)

• PART 1
  • Introduction
  • DP, DPLL and Resolution
    • Resolution Proof System (RPS)
    • DP Procedure
    • DPLL
    • Some Theoretical Results About RPS
    • Extended RPS
  • Modern SAT-solvers
  • .....
Extended RPS is
“resolution operation” + “extension rule”
(the extension rule was introduced by Tseitin in 1968)

Let $F(X)$ be a CNF formula where $X = \{x_1, \ldots, x_n\}$ An extension rule is to introduce a new variable: $y \equiv g(X')$ where $X' \subseteq X$. The variable $y$ is introduced as a set of clauses.

Suppose that $y \equiv x_m \land x_k$. Then new variable $y$ can be introduced by adding to $F$ the following three clauses:

$$(y \lor \neg x_m \lor \neg x_k), (\neg y \lor x_m), (\neg y \lor x_k)$$
Power of Extended RPS

Extended RPS is a powerful proof system. (In contrast to RPS, it is not broken yet).

However, it has even more non-determinism than RPS. In RPS the main source of non-determinism is in picking a pair of clauses to resolve.

The extension rule adds one more powerful source of non-determinism: picking a new variable to add. So building a SAT-solver based on extended resolution is hard.
Introduction of New Variables in SAT-algorithms

Some SAT-algorithms (Stalmarck algorithm, eqsatz (Chu Min LI)) use equality reasoning by exploring branches $x_k = x_m$ and $x_k \neq x_m$.

This can be simulated as exploring branches $y = 0$ and $y = 1$ where $y$ is a new variable: $y \equiv (x_k = x_m)$.

Introduction of these new variables leads to performance improvement on some classes of formulas. However, unless there is some information about the formula to be solved, using new variables is a gamble.
Proof System Automatizability

Informally, a proof system is **automatizable** if there is an efficient algorithm for finding “good” proofs in this system (Bonet, Pitassi and Raz, 2000)

**Treelike RPS and RPS** are most likely **non-automatizable** (Alekhnovich and Razborov, 2001)

A rule of thumb: the more powerful a proof system is, the “less automatizable” it is.

**Extended RPS** is exponentially more powerful than RPS. RPS is exponentially more powerful than tree-like RPS.

All successful resolution-based SAT-solvers are essentially based on tree-like RPS. Building a SAT-solver based on RPS (let alone extended RPS) is still an open problem.
PART 1

• Introduction
• DP, DPLL and resolution
• Modern SAT-solvers
  • Introduction
  • Right Branch “Elimination”
  • Non-chronological Backtracking
  • Useful Implicates
  • Conflict Driven Learning
  • Decision Making
  • Fast BCP, Restarts and Preprocessing
• Conclusions

• .....
Brief History of SAT-solving (related to resolution)

Existential quantification, DP (1960)
Variable splitting, DPLL (1962)
Learning short clauses, Socrates (1988)
Recursive learning, Hannibal (1994)
Equality reasoning, learning, Stalmarck algorithm (1996)
Fast BCP, SATO (1996)
Improved BCP, new decision making, Chaff (2001)
Better decision making and other improvements BerkMin (2002)
More improvements Siege (2003), Minisat (2005)
Local Search Algorithms

An important class of SAT-solvers are incomplete algorithms meant for solving only satisfiable formulas like GSAT (1992), Walksat (1994) and many others.

Let $F$ be a CNF formula. A typical local search SAT-algorithm maintains a complete assignment $x$ and the set $M(x)$ of clauses of $F$ falsified by $x$. At every step such an algorithm tries to flip an assignment of $x$ to reduce the number of falsified clauses.

If such an assignment does not exist, typically, a random value (whose choice may be constrained by some heuristic) is flipped in $x$.

In this talk, local search algorithms are not considered.
A Typical Resolution-Based SAT-Solver (no restarts or preprocessing)

// F is the original CNF formula, S is solver’s state

\[ Sat\_solver(F,S) \]

{while (true) { // starting the main loop

  if \( (BCP(F,S) == \text{‘conflict’}) \) { // a conflict?

    \[ C = generate\_conflict\_clause(F,S); \] // learn a new clause

    if \( (C \text{ is empty}) \) return \('\text{unsat}')\); // the learned clause is empty

    else \{ \[ F = F \cup C; \] // C is not empty.

      \[ backtrack(F,S); \] \} } // Add C to F and backtrack

  else // no conflict

    if \( (all\_vars\_assigned(F,S)) \) return \('\text{sat}')\); // all vars are assigned

    else make\_assignment(F,S);}} // assign value to a free var
A Typical Resolution-Based SAT-Solver (with preprocessing and restarts)

// F is the original CNF formula, S is solver’s state
Sat_solver(F,S)
{preprocessing(F,S); // learn some short clauses and/or remove vars
  while (true) {
    // starting the main loop
    if (a_condition_holds(F,S) restart; // time to make a restart
    if (BCP(F,S) == ‘conflict’) // a conflict?
      ……
      ……
    else make_assignment(F,S);}} // assign value to a free var
What Is Different from DPLL?

Preprocessing*: New implications may be added as "short" clauses. Some variables may be existentially removed away.

Restarts: If a condition is met (e.g. the number of conflicts exceeds a threshold), a new search tree one is started.

Non-chronological Backtracking: when backtracking a SAT-solver jumps over decision assignments that did not contribute to the last conflict.
What Is Different from DPLL? (cont.)

Conflict clause generation: if a conflict occurs a new nontrivial implication is added to the current formula.

Fast BCP: Skipping clauses that can not be unit

Recursion is not used: Instead of calling itself recursively, a SAT-solver maintains a stack of decision levels.

No right branches: The right branch assignment is derived from a conflict clause