Understanding and using SAT solvers
A practitioner perspective

Daniel Le Berre

CRIL-CNRS UMR 8188

Summer School 2009: Verification Technology, Systems & Applications
Nancy, October 12-16, 2009

1. Contains material provided by Prof. Joao Marques Silva
Agenda

Introduction to SAT

Early approaches to tackle SAT problems

The CDCL framework

From GRASP to CHAFF

Anatomy of a modern CDCL SAT solver

Practicing SAT

Some results from the SAT Competition 2009
Disclaimer

- Not a complete view of the subject
- Limited to one branch of SAT research (CDCL solvers)
- From an AI background point of view
- From a SAT solver designer
- For a broader picture of the area, see the handbook *edited this year by the community*
Remember that the best solvers for practical SAT solving in the 90’s were based on local search.

This decade has been the one of Conflict Driven Clause Learning solvers.

The next one may rise a new kind of solvers ...
Agenda

Introduction to SAT

Early approaches to tackle SAT problems

The CDCL framework

From GRASP to CHAFF

Anatomy of a modern CDCL SAT solver

Practicing SAT

Some results from the SAT Competition 2009
The SAT problem

Definition
Input: A set of clauses built from a propositional language with $n$ variables.
Output: Is there an assignment of the $n$ variables that satisfies all those clauses?
The SAT problem

Definition
Input: A set of clauses built from a propositional language with \( n \) variables.
Output: Is there an assignment of the \( n \) variables that satisfies all those clauses?

Example

\[ C_1 = \{ \neg a \lor b, \neg b \lor c \} = (\neg a \lor b) \land (\neg b \lor c) = (a' + b).(b' + c) \]

\[ C_2 = C_1 \cup \{ a, \neg c \} = C_1 \land a \land \neg c \]

For \( C_1 \), the answer is yes, for \( C_2 \) the answer is no.

\[ C_1 \models \neg(a \land \neg c) = \neg a \lor c \]
Where are clauses coming from? AI point of view

Suppose:

- a) I like free software
- b) I should start a free software project
- c) I should use a free software language

Then $C_1$ could represent the beliefs:

- $a \rightarrow b$: If I like free software, then I should start a free software project.
- $b \rightarrow c$: If I start a free software project, then I should use a free software language.

What happens if I like free software and I do not use a free software language ($a \land \neg c$)? This is inconsistent with my beliefs. From $C_1$ I can deduce $a \rightarrow c$: If I like free software, then I should use a free software language.
Definition
Given an initial state $s_0$, a state transition relation $ST$, a goal state $g$ and a bound $k$.
Is there a way to reach $g$ from $s_0$ using $ST$ within $k$ steps?
Is there a succession of states $s_0, s_1, s_2, \ldots, s_k = g$ such that
$\forall \ 0 \leq i < k \ (s_{i-1}, s_i) \in ST$?

- The problems are generated for increasing $k$.
- For small $k$, the problems are usually **UNSATISFIABLE**
- For larger $k$, the problems can be either **SAT** or **UNSAT**.
- Complete SAT solvers are needed!
SAT is important in theory ...

- Canonical NP-Complete problem [Cook, 1971]
- Threshold phenomenon on randomly generated $k$-SAT instances [Mitchell, Selman, Levesque, 1992]

Source:
... and in practice:

Computer Aided Verification Award 2009

awarded to

Conor F. Madigan
Sharad Malik
Joao Marques-Silva
Matthew Moskewicz
Karem Sakallah
Lintao Zhang
Ying Zhao

for

fundamental contributions to the development of high-performance Boolean satisfiability solvers.

Authors of GRASP SAT solver
Authors of CHAFF SAT solver
Where can we find SAT technology today?

- Formal methods:
  - Hardware model checking; Software model checking; Termination analysis of term-rewrite systems; Test pattern generation (testing of software & hardware); etc.

- Artificial intelligence:
  - Planning; Knowledge representation; Games (n-queens, sudoku, social golpher’s, etc.)

- Bioinformatics:
  - Haplotype inference; Pedigree checking; Analysis of Genetic Regulatory Networks; etc.

- Design automation:
  - Equivalence checking; Delay computation; Fault diagnosis; Noise analysis; etc.

- Security:
  - Cryptanalysis; Inversion attacks on hash functions; etc.
Where can we find SAT technology today? II

- Computationally hard problems:
  - Graph coloring; Traveling salesperson; etc.
- Mathematical problems:
  - van der Waerden numbers; Quasigroup open problems; etc.
- Core engine for other solvers: 0-1 ILP/Pseudo Boolean; QBF; \#SAT; SMT; MAXSAT; ...
- Integrated into theorem provers: HOL; Isabelle; ...
- Integrated into widely used software:
  - Suse 10.1 dependency manager based on a custom SAT solver.
  - Eclipse provisioning system based on a Pseudo Boolean solver.
Introduction to SAT

Early approaches to tackle SAT problems

The CDCL framework

From GRASP to CHAFF

Anatomy of a modern CDCL SAT solver

Practicing SAT

Some results from the SAT Competition 2009
Boolean Formulas

- Boolean formula \( \varphi \) is defined over a set of propositional variables \( x_1, \ldots, x_n \), using the standard propositional connectives \( \neg, \land, \lor, \rightarrow, \leftrightarrow \), and parenthesis
  
  - The domain of propositional variables is \( \{T, F\} \)
  
  - Example: \( \varphi(x_1, \ldots, x_3) = (\neg x_1 \land x_2) \lor \neg x_2 \lor x_3 ) \land (\neg x_2 \lor x_3 ) \)

- A formula \( \varphi \) in conjunctive normal form (CNF) is a conjunction of disjunctions (clauses) of literals, where a literal is a variable or its complement
  
  - Example: \( \varphi(x_1, \ldots, x_3) \equiv \)

- A formula \( \varphi \) in disjunctive normal form (DNF) is a disjunction of conjunctions (terms) of literals
  
  - Example: \( \varphi(x_1, \ldots, x_3) \equiv \)

- Can encode any Boolean formula into Normal Form
Boolean Formulas

- Boolean formula $\varphi$ is defined over a set of propositional variables $x_1, \ldots, x_n$, using the standard propositional connectives $\neg$, $\land$, $\lor$, $\rightarrow$, $\leftrightarrow$, and parenthesis.
  - The domain of propositional variables is $\{T, F\}$
  - Example: $\varphi(x_1, \ldots, x_3) = ((\neg x_1 \land x_2) \lor x_3) \land (\neg x_2 \lor x_3)$

- A formula $\varphi$ in conjunctive normal form (CNF) is a conjunction of disjunctions (clauses) of literals, where a literal is a variable or its complement.
  - Example: $\varphi(x_1, \ldots, x_3) \equiv (\neg x_1 \lor x_3) \land (x_2 \lor x_3) \land (\neg x_2 \lor x_3)$

- A formula $\varphi$ in disjunctive normal form (DNF) is a disjunction of conjunctions (terms) of literals.
  - Example: $\varphi(x_1, \ldots, x_3)$

- Can encode any Boolean formula into Normal Form
Boolean Formulas

- Boolean formula $\varphi$ is defined over a set of propositional variables $x_1, \ldots, x_n$, using the standard propositional connectives $\neg$, $\land$, $\lor$, $\rightarrow$, $\leftrightarrow$, and parenthesis
  - The domain of propositional variables is $\{T, F\}$
  - Example: $\varphi(x_1, \ldots, x_3) = ((\neg x_1 \land x_2) \lor x_3) \land (\neg x_2 \lor x_3)$

- A formula $\varphi$ in conjunctive normal form (CNF) is a conjunction of disjunctions (clauses) of literals, where a literal is a variable or its complement
  - Example: $\varphi(x_1, \ldots, x_3) \equiv (\neg x_1 \lor x_3) \land (x_2 \lor x_3) \land (\neg x_2 \lor x_3)$

- A formula $\varphi$ in disjunctive normal form (DNF) is a disjunction of conjunctions (terms) of literals
  - Example:
    $\varphi(x_1, \ldots, x_3) \equiv (\neg x_1 \land x_2 \land \neg x_2) \lor (x_3 \land \neg x_2) \lor (\neg x_1 \land x_2 \land x_3) \lor x_3$

- Can encode any Boolean formula into Normal Form
The resolution principle and classical simplification rules


**resolution:**

\[
\frac{x_1 \lor x_2 \lor x_3 \quad x_1 \lor \neg x_2 \lor x_4}{x_1 \lor x_1 \lor x_3 \lor x_4}
\]

**merging:**

\[
\frac{x_1 \lor x_1 \lor x_3 \lor x_4}{x_1 \lor x_3 \lor x_4}
\]

**subsumption:**

\[
\frac{\alpha \lor \beta \quad \alpha}{\alpha}
\]
The resolution principle and classical simplification rules


Resolution: $x_1 \lor x_2 \lor x_3 \lor x_4$ \[
\begin{array}{c}
\hline
x_1 \lor x_1 \lor x_3 \lor x_4 \\
\hline
\end{array}
\]

Merging: $x_1 \lor x_1 \lor x_3 \lor x_4$ \[
\begin{array}{c}
\hline
x_1 \lor x_3 \lor x_4 \\
\hline
\end{array}
\]

Subsumption: $\alpha \lor \beta \quad \alpha$ \[
\begin{array}{c}
\hline
\alpha \\
\hline
\end{array}
\]

What happens if we apply resolution between $\neg x_1 \lor x_2 \lor x_3$ and $x_1 \lor \neg x_2 \lor x_4$?
The resolution principle and classical simplification rules

\[
\text{resolution:} \quad \begin{array}{c}
  x_1 \lor x_2 \lor x_3 \\
  x_1 \lor \neg x_2 \lor x_4
\end{array} \\
\hline
  x_1 \lor x_1 \lor x_3 \lor x_4
\]

\[
\text{merging:} \quad \begin{array}{c}
  x_1 \lor x_1 \lor x_3 \lor x_4
\end{array} \\
\hline
  x_1 \lor x_3 \lor x_4
\]

\[
\text{subsumption:} \quad \begin{array}{c}
  \alpha \lor \beta
\end{array} \\
\hline
  \alpha
\]

What happens if we apply resolution between \(\neg x_1 \lor x_2 \lor x_3\) and \(x_1 \lor \neg x_2 \lor x_4\)?

A tautology: \(x_2 \lor \neg x_2 \lor x_3 \lor x_4\).
Apply resolution to decide satisfiability

- Apply resolution between clauses with exactly one opposite literal
- possible outcome:
  - a new clause is derived: removed subsumed clauses
  - the resolvent is subsumed by an existing clause
- until empty clause derived or no new clause derived
- Main issues of the approach:
  - In which order should the resolution steps be performed?
  - huge memory consumption!
The Davis and Putnam procedure: basic idea


Resolution used for variable elimination: \((A \lor x) \land (B \lor \neg x) \land R\) is satisfiable iff \((A \lor B) \land R\) is satisfiable.

- Iteratively apply the following steps:
  - Select variable \(x\)
  - Apply resolution between every pair of clauses of the form \((x \lor \alpha)\) and \((\neg x \lor \beta)\)
  - Remove all clauses containing either \(x\) or \(\neg x\)
- Terminate when either the **empty clause** or the **empty formula** is derived

Proof system: ordered resolution
Variable elimination – An Example

\((x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_2 \vee x_3) \wedge (x_3 \vee x_4) \wedge (x_3 \vee \neg x_4)\) ⊨
Variable elimination – An Example

\((x_1 \lor \neg x_2 \lor \neg x_3) \land (\neg x_1 \lor \neg x_2 \lor \neg x_3) \land (x_2 \lor x_3) \land (x_3 \lor x_4) \land (x_3 \lor \neg x_4)\) \models \\
\((\neg x_2 \lor \neg x_3) \land (x_2 \lor x_3) \land (x_3 \lor x_4) \land (x_3 \lor \neg x_4)\) \models
Variable elimination – An Example

\[
\begin{align*}
(x_1 \lor \neg x_2 \lor \neg x_3) & \land (\neg x_1 \lor \neg x_2 \lor \neg x_3) \land (x_2 \lor x_3) \land (x_3 \lor x_4) \land (x_3 \lor \neg x_4) \models \\
(\neg x_2 \lor \neg x_3) & \land (x_2 \lor x_3) \land (x_3 \lor x_4) \land (x_3 \lor \neg x_4) \models \\
(x_3 \lor \neg x_3) & \land (x_3 \lor x_4) \land (x_3 \lor \neg x_4) \models
\end{align*}
\]
Variable elimination – An Example

\[ (x_1 \lor \neg x_2 \lor \neg x_3) \land (\neg x_1 \lor \neg x_2 \lor \neg x_3) \land (x_2 \lor x_3) \land (x_3 \lor x_4) \land (x_3 \lor \neg x_4) \models \]

\[ (\neg x_2 \lor \neg x_3) \land (x_2 \lor x_3) \land (x_3 \lor x_4) \land (x_3 \lor \neg x_4) \models \]

\[ (x_3 \lor \neg x_3) \land (x_3 \lor x_4) \land (x_3 \lor \neg x_4) \models \]

\[ (x_3 \lor x_4) \land (x_3 \lor \neg x_4) \]
Variable elimination – An Example

\[
(x_1 \lor \neg x_2 \lor \neg x_3) \land (\neg x_1 \lor \neg x_2 \lor \neg x_3) \land (x_2 \lor x_3) \land (x_3 \lor x_4) \land (x_3 \lor \neg x_4) \\
\models \\
(x_2 \lor \neg x_3) \land (x_2 \lor x_3) \land (x_3 \lor x_4) \land (x_3 \lor \neg x_4) \\
\models \\
(x_3 \lor \neg x_3) \land (x_3 \lor x_4) \land (x_3 \lor \neg x_4) \\
\models \\
(x_3 \lor x_4) \land (x_3 \lor \neg x_4) \\
\models \\
x_3
\]
Variable elimination – An Example

\[(x_1 \lor \neg x_2 \lor \neg x_3) \land (\neg x_1 \lor \neg x_2 \lor \neg x_3) \land (x_2 \lor x_3) \land (x_3 \lor x_4) \land (x_3 \lor \neg x_4) \vdash\]

\[(\neg x_2 \lor \neg x_3) \land (x_2 \lor x_3) \land (x_3 \lor x_4) \land (x_3 \lor \neg x_4) \vdash\]

\[(x_3 \lor \neg x_3) \land (x_3 \lor x_4) \land (x_3 \lor \neg x_4) \vdash\]

\[(x_3 \lor x_4) \land (x_3 \lor \neg x_4) \vdash\]

\[x_3 \vdash\]

\[\top\]

- Formula is SAT
The Davis and Putnam procedure: the refinements

Add specific cases to order variable elimination steps

- Iteratively apply the following steps:
  - Apply the pure literal rule and unit propagation
  - Select variable $x$
  - Apply resolution between every pair of clauses of the form $(x \lor \alpha)$ and $(\neg x \lor \beta)$
  - Remove all clauses containing either $x$ or $\neg x$
- Terminate when either the empty clause or the empty formula is derived
A literal is pure if only occurs as a positive literal or as a negative literal in a CNF formula.

Example:
\[ \varphi = (\neg x_1 \lor x_2) \land (x_3 \lor \neg x_2) \land (x_4 \lor \neg x_5) \land (x_5 \lor \neg x_4) \]

\( \neg x_1 \) and \( x_3 \) are pure literals.

 Pure literal rule: eliminate first pure literals because no resolvant are produced!

applying a variable elimination step on a pure literal strictly reduces the number of clauses!

Preserve satisfiability, not logical equivalency!
Specific case of resolution: only shorten clauses.

unit resolution:

\[
\begin{align*}
x_1 \lor x_2 \lor x_3 & \lor \neg x_2 \\
x_1 \lor x_3 & \equiv (x_1 \lor x_2) \land \neg x_2
\end{align*}
\]

Preserve logical equivalency:

\[(x_1 \lor x_2 \lor x_3) \land \neg x_2 \equiv (x_1 \lor x_3) \land \neg x_2\]

Since clauses are shortened, new unit clauses may appear. Empty clauses also!

Unit propagation: apply unit resolution while new unit clauses are produced.
The approach runs easily out of memory.
Even recent attempts using a ROBDD representation [Simon and Chatalic 2000] does not scale well.

The solution: using backtrack search!
Propositional variables can be assigned value False or True.

- In some contexts variables may be unassigned.

A clause is satisfied if at least one of its literals is assigned value true.

\((x_1 \lor \neg x_2 \lor \neg x_3)\)

A clause is unsatisfied if all of its literals are assigned value false.

\((x_1 \lor \neg x_2 \lor \neg x_3)\)

A clause is unit if it contains one single unassigned literal and all other literals are assigned value False.

\((x_1 \lor \neg x_2 \lor \neg x_3)\)

A formula is satisfied if all of its clauses are satisfied.

A formula is unsatisfied if at least one of its clauses is unsatisfied.
Standard backtrack search

DPLL(F):
  - Apply unit propagation
  - If conflict identified, return UNSAT
  - Apply the pure literal rule
  - If F is satisfied (empty), return SAT
  - Select decision variable x
    - If DPLL(F ∧ x)=SAT return SAT
    - return DPLL(F ∧ ¬x)

Proof system: tree resolution
Pure Literals in backtrack search

★ Pure literal rule:
Clauses containing pure literals can be removed from the formula (i.e. just satisfy those pure literals)

★ Example:
ϕ = (¬x₁ ∨ x₂) ∧ (x₃ ∨ ¬x₂) ∧ (x₄ ∨ ¬x₅) ∧ (x₅ ∨ ¬x₄)

★ The resulting formula becomes:
ϕ₋₁,₃ = (x₄ ∨ ¬x₅) ∧ (x₅ ∨ ¬x₄)

★ if 𝑙 is a pure literal in Σ, then Σᵢ ⊂ Σ
★ Preserve satisfiability, not logical equivalency!
Unit Propagation in backtrack search

- **Unit clause rule in backtrack search**: Given a unit clause, its only unassigned literal must be assigned value True for the clause to be satisfied.

- **Example**: for unit clause \((x_1 \lor \neg x_2 \lor \neg x_3)\), \(x_3\) must be assigned value False.

- **Unit propagation**: Iterated application of the unit clause rule.

\[
(x_1 \lor \neg x_2 \lor \neg x_3) \land (\neg x_1 \lor \neg x_3 \lor x_4) \land (\neg x_1 \lor \neg x_2 \lor x_4)
\]
Unit Propagation in backtrack search

- **Unit clause rule in backtrack search**: Given a unit clause, its only unassigned literal must be assigned value True for the clause to be satisfied.

- **Example**: for unit clause \( (x_1 \lor \neg x_2 \lor \neg x_3) \), \( x_3 \) must be assigned value False.

- **Unit propagation**: Iterated application of the unit clause rule.
  
  \[
  (x_1 \lor \neg x_2 \lor \neg x_3) \land (\neg x_1 \lor \neg x_3 \lor x_4) \land (\neg x_1 \lor \neg x_2 \lor x_4)
  \]
Unit Propagation in backtrack search

- **Unit clause rule in backtrack search**: Given a unit clause, its only unassigned literal must be assigned value True for the clause to be satisfied.

- **Example**: for unit clause \((x_1 \lor \neg x_2 \lor \neg x_3)\), \(x_3\) must be assigned value False.

- **Unit propagation**: Iterated application of the unit clause rule

\[
(x_1 \lor \neg x_2 \lor \neg x_3) \land (\neg x_1 \lor \neg x_3 \lor x_4) \land (\neg x_1 \lor \neg x_2 \lor x_4)
\]
Unit Propagation in backtrack search

- **Unit clause rule in backtrack search**: Given a unit clause, its only unassigned literal must be assigned value True for the clause to be satisfied.

- **Example**: for unit clause \((x_1 \lor \neg x_2 \lor \neg x_3)\), \(x_3\) must be assigned value False.

- **Unit propagation**: Iterated application of the unit clause rule

\[
(x_1 \lor \neg x_2 \lor \neg x_3) \land (\neg x_1 \lor \neg x_3 \lor x_4) \land (\neg x_1 \lor \neg x_2 \lor x_4)
\]

\[
(x_1 \lor \neg x_2 \lor \neg x_3) \land (\neg x_1 \lor \neg x_3 \lor x_4) \land (\neg x_1 \lor \neg x_2 \lor \neg x_4)
\]
Unit Propagation in backtrack search

- **Unit clause rule in backtrack search**: Given a unit clause, its only unassigned literal must be assigned value True for the clause to be satisfied.

- **Example**: for unit clause \((x_1 \lor \neg x_2 \lor \neg x_3)\), \(x_3\) must be assigned value False.

- **Unit propagation**: Iterated application of the unit clause rule.

\[
(x_1 \lor \neg x_2 \lor \neg x_3) \land (\neg x_1 \lor \neg x_3 \lor x_4) \land (\neg x_1 \lor \neg x_2 \lor x_4)
\]

\[
(x_1 \lor \neg x_2 \lor \neg x_3) \land (\neg x_1 \lor \neg x_3 \lor x_4) \land (\neg x_1 \lor \neg x_2 \lor \neg x_4)
\]
Unit Propagation in backtrack search

- **Unit clause rule in backtrack search**: Given a unit clause, its only unassigned literal must be assigned value True for the clause to be satisfied.

- **Example**: for unit clause \((x_1 \lor \neg x_2 \lor \neg x_3)\), \(x_3\) must be assigned value False.

- **Unit propagation**: Iterated application of the unit clause rule.

\[
(x_1 \lor \neg x_2 \lor \neg x_3) \land (\neg x_1 \lor \neg x_3 \lor x_4) \land (\neg x_1 \lor \neg x_2 \lor x_4)
\]

\[
(x_1 \lor \neg x_2 \lor \neg x_3) \land (\neg x_1 \lor \neg x_3 \lor x_4) \land (\neg x_1 \lor \neg x_2 \lor \neg x_4)
\]
Unit Propagation in backtrack search

Unit clause rule in backtrack search:
Given a unit clause, its only unassigned literal must be assigned value True for the clause to be satisfied

Example: for unit clause \((x_1 \lor \neg x_2 \lor \neg x_3)\), \(x_3\) must be assigned value False

Unit propagation
Iterated application of the unit clause rule

\[
(x_1 \lor \neg x_2 \lor \neg x_3) \land (\neg x_1 \lor \neg x_3 \lor x_4) \land (\neg x_1 \lor \neg x_2 \lor x_4)
\]

Unit propagation can satisfy clauses but can also unsatisfy clauses (i.e., conflicts)
An Example of DPLL

\[ \varphi = (a \lor \neg b \lor d) \land (a \lor \neg b \lor e) \land \\
(\neg b \lor \neg d \lor \neg e) \land \\
(a \lor b \lor c \lor d) \land (a \lor b \lor c \lor \neg d) \land \\
(a \lor b \lor \neg c \lor e) \land (a \lor b \lor \neg c \lor \neg e) \]
An Example of DPLL

\[ \varphi = (a \lor \neg b \lor d) \land (a \lor \neg b \lor e) \land \\
(\neg b \lor \neg d \lor \neg e) \land \\
(a \lor b \lor c \lor d) \land (a \lor b \lor c \lor \neg d) \land \\
(a \lor b \lor \neg c \lor e) \land (a \lor b \lor \neg c \lor \neg e) \]
An Example of DPLL

\[ \varphi = (a \lor \neg b \lor d) \land (a \lor \neg b \lor e) \land (\neg b \lor \neg d \lor \neg e) \land (a \lor b \lor c \lor d) \land (a \lor b \lor c \lor \neg d) \land (a \lor b \lor \neg c \lor e) \land (a \lor b \lor \neg c \lor \neg e) \]
An Example of DPLL

ϕ = (a ∨ ¬b ∨ d) ∧ (a ∨ ¬b ∨ e) ∧
    (¬b ∨ ¬d ∨ ¬e) ∧
    (a ∨ b ∨ c ∨ d) ∧ (a ∨ b ∨ c ∨ ¬d) ∧
    (a ∨ b ∨ ¬c ∨ e) ∧ (a ∨ b ∨ ¬c ∨ ¬e)
An Example of DPLL

\[ \varphi = (a \lor \neg b \lor d) \land (a \lor \neg b \lor e) \land \\
(\neg b \lor \neg d \lor \neg e) \land \\
(a \lor b \lor c \lor d) \land (a \lor b \lor c \lor \neg d) \land \\
(a \lor b \lor \neg c \lor e) \land (a \lor b \lor \neg c \lor \neg e) \]
An Example of DPLL

\[ \varphi = (a \lor \neg b \lor d) \land (a \lor \neg b \lor e) \land \\
(\neg b \lor \neg d \lor \neg e) \land \\
(a \lor b \lor c \lor d) \land (a \lor b \lor c \lor \neg d) \land \\
(a \lor b \lor \neg c \lor e) \land (a \lor b \lor \neg c \lor \neg e) \]

conflict
An Example of DPLL

\[ \varphi = (a \lor \neg b \lor d) \land (a \lor \neg b \lor e) \land (\neg b \lor \neg d \lor \neg e) \land (a \lor b \lor c \lor d) \land (a \lor b \lor c \lor \neg d) \land (a \lor b \lor \neg c \lor e) \land (a \lor b \lor \neg c \lor \neg e) \]
An Example of DPLL

\[
\varphi = (a \lor \neg b \lor d) \land (a \lor \neg b \lor e) \land \\
(\neg b \lor \neg d \lor \neg e) \land \\
(a \lor b \lor c \lor d) \land (a \lor b \lor c \lor \neg d) \land \\
(a \lor b \lor \neg c \lor e) \land (a \lor b \lor \neg c \lor \neg e)
\]
DP, DLL or DPLL?

- DPLL = DP + DLL
- Acknowledge the principles in DP60 and their memory efficient implementation in DP62
- DPLL commonly used to denote complete solvers for SAT: no longer true for modern complete SAT solvers.
- The focus of researchers in the 90’s was mainly to improve the heuristics to select the variables to branch on on randomly generated formulas.
- Introduction of non chronological backtracking and learning to solve structured/real world formulas.
Clause Learning

- During backtrack search, for each conflict learn new clause, which explains and prevents repetition of the same conflict

\[ \varphi = (a \lor b) \land (\neg b \lor c \lor d) \land (\neg b \lor e) \land (\neg d \lor \neg e \lor f) \ldots \]
During backtrack search, for each conflict learn new clause, which explains and prevents repetition of the same conflict

\[ \varphi = (a \lor b) \land (\neg b \lor c \lor d) \land (\neg b \lor e) \land (\neg d \lor \neg e \lor f) \ldots \]

Assume decisions \( c = False \) and \( f = False \)
During backtrack search, for each conflict learn new clause, which explains and prevents repetition of the same conflict:

\[ \varphi = (a \lor b) \land (\neg b \lor c \lor d) \land (\neg b \lor e) \land (\neg d \lor \neg e \lor f) \ldots \]

- Assume decisions \( c = \text{False} \) and \( f = \text{False} \)
- Assign \( a = \text{False} \) and imply assignments
During backtrack search, for each conflict learn new clause, which explains and prevents repetition of the same conflict

\[ \varphi = (a \lor b) \land (\neg b \lor c \lor d) \land (\neg b \lor e) \land (\neg d \lor \neg e \lor f) \ldots \]

- Assume decisions \(c = False\) and \(f = False\)
- Assign \(a = False\) and imply assignments
During backtrack search, for each conflict learn new clause, which explains and prevents repetition of the same conflict

$$\varphi = (a \lor b) \land (\neg b \lor c \lor d) \land (\neg b \lor e) \land (\neg d \lor \neg e \lor f) \ldots$$

- Assume decisions $c = False$ and $f = False$
- Assign $a = False$ and imply assignments
- A conflict is reached: $\neg d \lor \neg e \lor f$ is unsatisfied
Clause Learning

- During backtrack search, for each conflict learn new clause, which explains and prevents repetition of the same conflict

\[ \varphi = (a \lor b) \land (\neg b \lor c \lor d) \land (\neg b \lor e) \land (\neg d \lor \neg e \lor f) \ldots \]

- Assume decisions \( c = False \) and \( f = False \)
- Assign \( a = False \) and imply assignments
- A conflict is reached: \((\neg d \lor \neg e \lor f)\) is unsatisfied
- \( \varphi \land \neg a \land \neg c \land \neg f \Rightarrow \bot \)
Clause Learning

- During backtrack search, for each conflict learn new clause, which explains and prevents repetition of the same conflict

\[ \varphi = (a \lor b) \land (\neg b \lor c \lor d) \land (\neg b \lor e) \land (\neg d \lor \neg e \lor f) \ldots \]

- Assume decisions \( c = False \) and \( f = False \)
- Assign \( a = False \) and imply assignments
- A conflict is reached: \( (\neg d \lor \neg e \lor f) \) is unsatisfied
- \( \varphi \land \neg a \land \neg c \land \neg f \Rightarrow \bot \)
- \( \varphi \Rightarrow a \lor c \lor f \)
During backtrack search, for each conflict learn new clause, which explains and prevents repetition of the same conflict

\[ \varphi = (a \lor b) \land (\neg b \lor c \lor d) \land (\neg b \lor e) \land (\neg d \lor \neg e \lor f) \ldots \]

- Assume decisions \( c = False \) and \( f = False \)
- Assign \( a = False \) and imply assignments
- A conflict is reached: \((\neg d \lor \neg e \lor f)\) is unsatisfied
- \( \varphi \land \neg a \land \neg c \land \neg f \Rightarrow \bot \)
- \( \varphi \Rightarrow a \lor c \lor f \)

- Learn new clause \((a \lor c \lor f)\)
During backtrack search, for each conflict backtrace to one of the causes of the conflict

\[ \varphi = (a \lor b) \land (\neg b \lor c \lor d) \land (\neg b \lor e) \land (\neg d \lor \neg e \lor f) \land (a \lor c \lor f) \land (\neg a \lor g) \land (\neg g \lor b) \land (\neg h \lor j) \land (\neg i \lor k) \]
Non-Chronological Backtracking

- During backtrack search, for each conflict backtrack to one of the causes of the conflict

\[
\varphi = (a \lor b) \land (\neg b \lor c \lor d) \land (\neg b \lor e) \land (\neg d \lor \neg e \lor f) \land \\
(a \lor c \lor f) \land (\neg a \lor g) \land (\neg g \lor b) \land (\neg h \lor j) \land (\neg i \lor k)
\]

- Assume decisions \(c = \text{False}, f = \text{False}, h = \text{False}\) and \(i = \text{False}\)
During backtrack search, for each conflict backtrack to one of the causes of the conflict

\[ \varphi = (a \lor b) \land (\neg b \lor c \lor d) \land (\neg b \lor e) \land (\neg d \lor \neg e \lor f) \land (a \lor c \lor f) \land (\neg a \lor g) \land (\neg g \lor b) \land (\neg h \lor j) \land (\neg i \lor k) \]

Assume decisions \( c = \text{False}, f = \text{False}, h = \text{False} \) and \( i = \text{False} \)

Assignment \( a = \text{False} \) caused conflict \( \Rightarrow \) learnt clause \((a \lor c \lor f)\) implies \( a\)
During backtrack search, for each conflict backtrack to one of the causes of the conflict

\[ \varphi = (a \lor b) \land (\neg b \lor c \lor d) \land (\neg b \lor e) \land (\neg d \lor \neg e \lor f) \land (a \lor c \lor f) \land (\neg a \lor g) \land (\neg g \lor b) \land (\neg h \lor j) \land (\neg i \lor k) \]

Assume decisions \( c = False, f = False, h = False \) and \( i = False \)

Assignment \( a = False \) caused conflict \( \Rightarrow \) learnt clause \( (a \lor c \lor f) \) implies \( a \)
Non-Chronological Backtracking

- During backtrack search, for each conflict back track to one of the causes of the conflict

\[ \varphi = (a \lor b) \land (\neg b \lor c \lor d) \land (\neg b \lor e) \land (\neg d \lor \neg e \lor f) \land (a \lor c \lor f) \land (\neg a \lor g) \land (\neg g \lor b) \land (\neg h \lor j) \land (\neg i \lor k) \]

- Assume decisions \( c = False, f = False, h = False \) and \( i = False \)
- Assignment \( a = False \) caused conflict \( \Rightarrow \) learnt clause \((a \lor c \lor f)\) implies \(a\)
Non-Chronological Backtracking

During backtrack search, for each conflict backtrack to one of the causes of the conflict

\[ \varphi = (a \lor b) \land (\neg b \lor c \lor d) \land (\neg b \lor e) \land (\neg d \lor \neg e \lor f) \land (a \lor c \lor f) \land (\neg a \lor g) \land (\neg g \lor b) \land (\neg h \lor j) \land (\neg i \lor k) \]

Assume decisions \( c = False, f = False, h = False \) and \( i = False \)

Assignment \( a = False \) caused conflict \( \Rightarrow \) learnt clause \( (a \lor c \lor f) \) implies \( a \)

A conflict is again reached: \( (\neg d \lor \neg e \lor f) \) is unsatisfied
Non-Chronological Backtracking

- During backtrack search, for each conflict backtrack to one of the causes of the conflict

\[ \varphi = (a \lor b) \land (\neg b \lor c \lor d) \land (\neg b \lor e) \land (\neg d \lor \neg e \lor f) \land (a \lor c \lor f) \land (\neg a \lor g) \land (\neg g \lor b) \land (\neg h \lor j) \land (\neg i \lor k) \]

- Assume decisions \( c = False \), \( f = False \), \( h = False \) and \( i = False \)
- Assignment \( a = False \) caused conflict \( \Rightarrow \) learnt clause \((a \lor c \lor f) \) implies \( a \)

A conflict is again reached : \((\neg d \lor \neg e \lor f) \) is unsatisfied
- \( \varphi \land \neg c \land \neg f \Rightarrow \bot \)
Non-Chronological Backtracking

During backtrack search, for each conflict backtrack to one of the causes of the conflict

\[ \varphi = (a \lor b) \land (\neg b \lor c \lor d) \land (\neg b \lor e) \land (\neg d \lor \neg e \lor f) \land (a \lor c \lor f) \land (\neg a \lor g) \land (\neg g \lor b) \land (\neg h \lor j) \land (\neg i \lor k) \]

- Assume decisions \( c = False, f = False, h = False \) and \( i = False \)
- Assignment \( a = False \) caused conflict \( \Rightarrow \) learnt clause \((a \lor c \lor f)\) implies \( a \)
- A conflict is again reached : \((\neg d \lor \neg e \lor f)\) is unsatisfied
- \( \varphi \land \neg c \land \neg f \Rightarrow \bot \)
- \( \varphi \Rightarrow c \lor f \)
Non-Chronological Backtracking

During backtrack search, for each conflict backtrack to one of the causes of the conflict

\[ \varphi = (a \lor b) \land (\neg b \lor c \lor d) \land (\neg b \lor e) \land (\neg d \lor \neg e \lor \textcolor{red}{f}) \land (a \lor c \lor \textcolor{red}{f}) \land (\neg a \lor g) \land (\neg g \lor b) \land (\neg h \lor j) \land (\neg i \lor k) \]

- Assume decisions \( c = \text{False} \), \( f = \text{False} \), \( h = \text{False} \) and \( i = \text{False} \)
- Assignment \( a = \text{False} \) caused conflict \( \Rightarrow \) learnt clause \((a \lor c \lor f)\) implies \( a \)
- A conflict is again reached : \((\neg d \lor \neg e \lor f)\) is unsatisfied
- \( \varphi \land \neg c \land \neg f \Rightarrow \bot \)
- \( \varphi \Rightarrow c \lor f \)
- Learn new clause \((c \lor f)\)
Non-Chronological Backtracking

\[(a + c + f) \quad (c + f)\]
Non-Chronological Backtracking

- Learnt clause: \((c \lor f)\)
- Need to backtrack, given new clause
- Backtrack to most recent decision: \(f = False\)

Clause learning and non-chronological backtracking are hallmarks of modern SAT solvers.
How to implement NCB and Learning? Resolution!

Perform resolution steps in reverse order of the assignments.
Propagations deriving from a : g,b,d, e

\[ \varphi = (a \lor b) \land (\neg b \lor c \lor d) \land (\neg b \lor e) \land (\neg d \lor \neg e \lor f) \land (a \lor c \lor f) \land (\neg a \lor g) \land (\neg g \lor b) \land (\neg h \lor j) \land (\neg i \lor k) \]

Learned: \((a \lor c \lor f)\)

\((\neg d \lor \neg e \lor f)\)
Perform resolution steps in reverse order of the assignments. Propagations deriving from $a : g, b, d, e$

$$\varphi = (a \lor b) \land (\neg b \lor c \lor d) \land (\neg b \lor e) \land (\neg d \lor \neg e \lor f) \land (a \lor c \lor f) \land (\neg a \lor g) \land (\neg g \lor b) \land (\neg h \lor j) \land (\neg i \lor k)$$

Learned : $(a \lor c \lor f)$

$$(\neg b \lor \neg d \lor f)$$
Perform resolution steps in reverse order of the assignments. Propagations deriving from $a : g, b, d, e$.

$$\varphi = (a \lor b) \land (\neg b \lor c \lor d) \land (\neg b \lor e) \land (\neg d \lor \neg e \lor f) \land (a \lor c \lor f) \land (\neg a \lor g) \land (\neg g \lor b) \land (\neg h \lor j) \land (\neg i \lor k)$$

Learned: $(a \lor c \lor f)$

$$(\neg b \lor c \lor f)$$
Perform resolution steps in reverse order of the assignments. 

Propagations deriving from $a$: $g$, b, d, e

$$\varphi = (a \lor b) \land (\neg b \lor c \lor d) \land (\neg b \lor e) \land (\neg d \lor \neg e \lor f) \land (a \lor c \lor f) \land (\neg a \lor g) \land (\neg g \lor b) \land (\neg h \lor j) \land (\neg i \lor k)$$

Learned: $(a \lor c \lor f)$

$$(\neg g \lor c \lor f)$$
Perform resolution steps in reverse order of the assignments.
Propagations deriving from \( a : g, b, d, e \)

\[ \varphi = (a \lor b) \land (\neg b \lor c \lor d) \land (\neg b \lor e) \land (\neg d \lor \neg e \lor f) \land (a \lor c \lor f) \land (\neg a \lor g) \land (\neg g \lor b) \land (\neg h \lor j) \land (\neg i \lor k) \]

Learned: \( (a \lor c \lor f) \)

\[ (\neg a \lor c \lor f) \]
Perform resolution steps in reverse order of the assignments. Propagations deriving from $a : g,b,d,e$

\[
\varphi = (a \lor b) \land (\neg b \lor c \lor d) \land (\neg b \lor e) \land (\neg d \lor \neg e \lor f) \land \\
(a \lor c \lor f) \land (\neg a \lor g) \land (\neg g \lor b) \land (\neg h \lor j) \land (\neg i \lor k)
\]

Learned : $(a \lor c \lor f)$

$(c \lor f)$
Implementation of NCB and Learning for SAT

- Two approaches developed independently in two different research communities:
  - **GRASP/EDA** by Marques-Silva and Sakallah (1996)
    - Resolution graph seen as a circuit
    - Conflict analysis thought as detecting faults in a circuit
    - Other sophisticated conflict analysis methods based on truth maintenance systems
  - **RELSAT/CSP** by Bayardo and Schrag (1997)
    - Introduction of CSP based techniques into a SAT solver
    - Conflict Directed Backjumping aka non chronological backtracking [Prosser 93]
    - Size based and relevance based learning schemes

- Main difference: in GRASP’s framework, the conflict analysis drives the search, while in RELSAT it is the heuristics (more later).
Introduction to SAT

Early approaches to tackle SAT problems

The CDCL framework

From GRASP to CHAFF

Anatomy of a modern CDCL SAT solver

Practicing SAT

Some results from the SAT Competition 2009
GRASP architecture

Role of the boolean propagator

- Perform unit propagation on the set of clauses.
- Detect conflicts
- Backtrack according to a specific clause provided by the conflict analyzer
Conflict analyzer

- Must produce a clause that becomes a unit clause after backtracking (asserting clause)
- Introduction of the notion of Unique Implication Point (UIP), as a reference to Unique Sensitization Points in ATPG.
  - Find a literal that need to be propagated before reaching a conflict
  - Based on the notion of decision level, i.e. the number of assumptions made so far.
  - Syntactical: apply resolution until only one literal from current decision level appears in the clause.
  - Decision variables are always UIP: at least one UIP exists for each conflict!
- Backtracking level computed as the lowest decision level of the literals of the clause
Conflict graph for assumption $a=False$

\[
\phi = (a \lor b) \land (\neg b \lor c \lor d) \land (\neg b \lor e) \land (\neg d \lor \neg e \lor f) \ldots
\]
Conflict graph after learning $a \lor c \lor f$ and backjumping

\[
\varphi = (a \lor b) \land (\neg b \lor c \lor d) \land (\neg b \lor e) \land (\neg d \lor \neg e \lor f) \land (a \lor c \lor f) \land (\neg a \lor g) \land (\neg g \lor b) \land (\neg h \lor j) \land (\neg i \lor k)
\]
Some remarks about UIPs

- There are many possibilities to derive a clause using UIP.
- RELSAT can be seen as applying Decision UIP.
- Decision UIP always flip the decision variable truth value: the search is thus driven by the heuristics.
- Using other UIP scheme, the value of any of the literal propagated at the current decision level may be flipped. The search is thus driven by the conflict analysis.
- Generic name for GRASP approach: Conflict Driven Clause Learning (CDCL) solver [Ryan 2004].
Decision heuristics

- Pick an unassigned variable
- Many sophisticated decision heuristics available in the literature for random formulas (MOMS, JW, etc).
- GRASP uses dynamic largest individual sum (DLIS) : select the literal with the maximum occurrences in unresolved clauses.
- Sophisticated heuristics require an exact representation of the state of the CNF after unit propagation!
Putting everything together: the CDCL approach

- **UNSAT**
  - **top level**
  - **analyze**
    - **learn**

- **propagate**
  - **conflict found**
  - **assume**
  - **no propagation**

- **SAT**
  - **all assigned**
  - **decide**
Agenda

Introduction to SAT

Early approaches to tackle SAT problems

The CDCL framework

From GRASP to CHAFF

Anatomy of a modern CDCL SAT solver

Practicing SAT

Some results from the SAT Competition 2009
Some key insights in the design of SAT solvers were discovered when trying to solve real problems by translation into SAT.

Huge interest on SAT after the introduction of Bounded Model Checking [Biere et al 99] from the EDA community.

The design of SAT solver becomes more pragmatic
Application 1: Planning as satisfiability


- Input: a set of actions, an initial state and a goal state
- Output: a sequence of actions to reach the goal state from the initial state
- One of the first applications of SAT in Artificial Intelligence
- A key application for the adoption of SAT in EDA later on
- The instances are supposed to be SAT
- Those instances are too big for complete solvers based on DPLL
1992 - Planning As Satisfiability

\[
PAS(S, I, T, G, k) = I(s_0) \land \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \land \bigvee_{i=0}^{k} G(s_i)
\]

ou : 

- \(S\) the set of possible states \(s_i\)
- \(I\) the initial state
- \(T\) transitions between states
- \(G\) goal state
- \(k\) bound

If the formula is satisfiable, then there is a plan of length at most \(k\).
function GSAT(CNF c, int maxtries, int maxflips) {
    // DIVERSIFICATION STEP
    for (int i = 0; i < maxtries; i++) {
        m = randomAssignment();
        // INTENSIFICATION STEP
        for (int j = 0; j < maxflips; j++) {
            if (m satisfies c)
                return SAT;
            flip(m);
        }
    }
    return UNKNOWN;
}
Lessons learned from GSAT

- The decision procedure is **very simple to implement and very fast**!
- Efficiency depends on which literal to flip, and the values of the parameters.
- Problem with local minima: use of Random Walks!
- Main drawback: incomplete, cannot answer UNSAT!
- Lesson 1: An agile (fast) SAT solver sometimes better than a clever one!
Application 2 : Quasigroup (Latin Square) open problems

- $S$ a set and $*$ a binary operator. $|S|$ is the order of the group.
- $a*b=c$ has a unique solution when fixing any pair of variables.
- Equivalent to fill in a $|S| \times |S|$ square with elements of $S$ unique in each row and column.
- Looking for the existence of QG of a given order with additional constraints, e.g.:

\[
\begin{align*}
\text{QG1} & \quad x \ast y = u, \quad z \ast w = u, \quad v \ast y = x, \quad v \ast w = z \Rightarrow \\
& \quad x = z, \quad y = w \\
\text{QG2} & \quad x \ast y = u, \quad z \ast w = u, \quad y \ast v = x, \quad w \ast v = z \Rightarrow \\
& \quad x = z, \quad y = w
\end{align*}
\]

- First open QG problems solved by MGTP (Fujita, Slaney, Benett 93)
- QG1(12), QG2(14), QG2(15) solved by SATO in 1996.
CNF resulting for QG problems have a huge amount of clauses: 10K to 150K!

Encoding of real problems into SAT can lead to very large clauses

Truth value propagation cost in eager data structure depends on the number of propagation to perform, thus on the size of the clauses

How to limit the cost of numerous and long clauses during propagation?

Answer: use a lazy data structure to detect only unit propagation and falsified literals.
The Head/Tail data structure

Initially, put a head (resp. tail) pointer to the first (resp. last) element of the clause.

During propagation, move heads or tails pointing to the negation of the propagated literal. Easy identification of unit and falsified clauses.

During backtracking, move back the pointers to their previous location.
Unit propagation with Adjacency lists
Unit propagation with Head / Tail
Pro and Cons of the H/T data structure

advantage reduces the cost of unit propagation

drawback the solver has no longer a complete picture of the reduced CNF!

Lesson 2 : data structure matters!
High variability of SAT solvers runtime!


- SAT solvers exhibits on some problems a high runtime variability
- Decision heuristics need to break ties, often randomly
- The solver are sensible to syntactical input changes:
  - Shuffled industrial benchmarks harder than original ones for most solvers
  - The “lisa syndrome” during the SAT 2003 competition
- An explanation: **Heavy tailed distribution**
Example of variability: SAT4J GreedySolver on QGH
Example of variability: SAT4J GreedySolver on QGH
Example of variability: SAT4J GreedySolver on QGH

Behavior of Mean cost value for SAT4J GreedySolver on qwh.35.405.shuffled-as.sat03-165
Heavy Tailed distribution

- Introduced by the economist Pareto in the context of income distribution
- Widely used in many areas: stock market analysis, weather forecast, earthquake prediction, time delays on the WWW.
- Those distributions have infinite mean and infinite variance
- Some SAT solvers exhibit an Heavy Tailed distribution on Quasigroup Completion with Holes problems.
- What does it mean in practice?
  - In rare occasion, the solver can get trapped on a very long run
  - while most of the time the run could be short
- the solution: **restarts**!
Restarting in SAT solvers

- Stop the search after a given number of conflicts/decisions/propagation is achieved (cutoff).
- Start again the search [with increased cutoff to be complete]
- Requires some variability in the solver behavior between two runs
- Problem: how to choose the cutoff value?
- In theory, an optimal strategy exists [Luby 93].
- Lesson 3: introduce restarts to make the solver more robust
The killer app: Bounded Model Checking


\[ BMC(S, I, T, p, k) = I(s_0) \land \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \land \bigvee_{i=0}^{k} \neg p(s_i) \]

where:

- **S** the set of possible states \( s_i \)
- **I** the initial state
- **T** transitions between states
- **p** is an invariant property
- **k** a bound

If the formula is satisfiable, then there is a counter-example reachable within \( k \) steps.
Some model checking problems out of reach of BDD checkers can be solved thanks to a reduction to SAT.

The behavior of SAT solvers is less dependent of the form of the input than BDD solvers.

But the SAT solvers are not powerful enough yet for industrial use...
The breakthrough : Chaff


- 2 order of magnitude speedup on unsat instances compared to existing approaches on BMC (Velev) benchmarks.
- Immediate speedup for SAT based tools : BlackBox “Supercharged with Chaff”
- Based on careful analysis of GRASP internals
- 3 key features :
  - New lazy data structure : Watched literals
  - New adaptative heuristic : Variable State Independent Decaying Sum
  - New conflict analysis approach : First UIP
- Taking into account randomization
The watched literals data structure

Initially watch two arbitrary literals in the clause
During propagation move watchers pointers in clauses containing the negation of the propagated literal.
During backtracking do nothing!

Advantage cost free data structure when backtracking
Issue pointers can move in both directions.
Variable State Independent Decaying Sum

- compatible with Lazy Data Structures
- each literal has a score
- score based on the number of occurrences of the literals in the formula
- score updated whenever a new clause is learned
- pick the unassigned literal with the highest score, tie broken randomly
- regularly (every 256 conflicts), divided the scores by a constant (2)
The idea is to **quickly** compute a reason for the conflict

- Stop the resolution process as soon as an UIP is detected
- First UIP Shown to be optimal in terms of backtrack level compared to the other possible UIPs [Audemard et al 08].
Learning does not degrade solver performance because the use of the watched literals.

The VSIDS heuristics does not need a complete picture of the reduced formula, i.e. is compatible with the lazy data structure.

VSIDS take advantage of the conflict analysis to spot important literals.

VSIDS provides different orders of literals at each restart.

VSIDS adapt itself to the instance!
The reason of the success?

- Better engineering (level 2 cache awareness)?
The reason of the success?

- Better engineering (level 2 cache awareness)?
- Better tradeoff between speed and intelligence?
The reason of the success?

- Better engineering (level 2 cache awareness)?
- Better tradeoff between speed and intelligence?
- Instance-based auto adaptation?
The reason of the success?

- Better engineering (level 2 cache awareness)?
- Better tradeoff between speed and intelligence?
- Instance-based auto adaptation?
- ...

The reason of the success?

- Better engineering (level 2 cache awareness)?
- Better tradeoff between speed and intelligence?
- Instance-based auto adaptation?
- ...

All those reasons are correct. **There is a more fundamental reason too ...**
CDCL has a better proof system than DPLL! Proof theory strikes back!

- ... thanks to many others before ...
CDCL has a better proof system than DPLL!
Proof theory strikes back!

**Definition**

p-simulation Proof system $S$ p-simulates proof system $T$, if, for every unsatisfiable formula $\varphi$, the shortest refutation proof of $\varphi$ in $S$ is at most polynomially longer than the shortest refutation proof of $\varphi$ in $T$. 
CDCL has a better proof system than DPLL!  
Proof theory strikes back!

Definition

p-simulation Proof system $S$ p-simulates proof system $T$, if, for every unsatisfiable formula $\varphi$, the shortest refutation proof of $\varphi$ in $S$ is at most polynomially longer than the shortest refutation proof of $\varphi$ in $T$.

Theorem 1 [Pipatsrisawat, Darwiche 09]. CLR with any asserting learning scheme p-simulates general resolution.
Since Chaff ...

- The international SAT competition/SAT race is organized every year
- A huge number of CDCL solvers have been developed, and made available to the community
- SAT has integrated the engineer toolbox to solve combinatorial problems
- Many papers published on the design of efficient SAT solvers
Since Chaff ...

- The international SAT competition/SAT race is organized every year.
- A huge number of CDCL solvers have been developed, and made available to the community.
- SAT has integrated the engineer toolbox to solve combinatorial problems.
- Many papers published on the design of efficient SAT solvers.
- ... but a big part of the knowledge still lies in source code!
Introduction to SAT

Early approaches to tackle SAT problems

The CDCL framework

From GRASP to CHAFF

Anatomy of a modern CDCL SAT solver

Practicing SAT

Some results from the SAT Competition 2009
Minisat: the minimalist CDCL SAT solver

- very simple implementation of a Chaff-like solver
- resulting from the lessons learned from designing Satzoo (SAT 2003 Winner) and SATnick
- with implementation improvements (Watched Literals, Heuristics, Priority Queue (2005), etc.)
- ready for generic constraints (cardinality, linear pseudo boolean, etc.).
- published description of the design

Reduced the entry level required to experiment with CDCL SAT solvers
The watched literals data structure improved

[mChaff, vanGelder02, Minisat]

Initially watch the two first literals in the clause during propagation move falsified literal in second position. Exchange it with an unassigned literal is any. Easy identification of unit and falsified clauses.

during backtracking do nothing!

Advantage cost free data structure when backtracking
The watched literals data structure improved
[mChaff, vanGelder02, Minisat]

initially watch the two first literals in the clause
during propagation move falsified literal in second position.
Exchange it with an unassigned literal is any. Easy identification of unit and falsified clauses.
during backtracking do nothing!

advantage cost free data structure when backtracking
Moving literals instead of pointers in HT data structure also provides cost free backtracking!
Berkmin style heuristic

Ideas:

▸ force the heuristic to satisfy recently learned clauses to be more reactive than VSIDS

▸ sophisticated phase selection strategy based on an estimate of the unit propagations to result from the selection (a la SATZ [Li Anbulagan 97]).

▸ take into account literals met during the conflict analysis

Berkmin performed quite well during SAT 2002 (despite a stupid bug) and it’s successor Forklift won in 2003.