CDCL SAT Solvers

Joao Marques-Silva

INESC-ID, IST, ULisbon, Portugal

Theory and Practice of SAT Solving
Dagstuhl Workshop
April 2015
The Success of SAT

- Well-known NP-complete decision problem
The Success of SAT

- Well-known NP-complete decision problem
- In practice, **SAT is a success story of Computer Science**
  - Hundreds (even more?) of practical applications
The Success of SAT

- Well-known NP-complete decision problem
- In practice, SAT is a success story of Computer Science
  - Hundreds (even more?) of practical applications
SAT Solver Improvement

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)
- Lingeling S87f (2011)
This Talk

- Review key ideas in implementing CDCL SAT solvers
  - Review standard concepts
    - Unit propagation
    - Resolution
    - Proof traces
    - ...
  - Outline organization of DPLL/CDCL SAT solvers
  - Survey most effective techniques
    - Clause learning & non-chronological backtracking
    - UIPs
    - Clause minimization
    - Search restarts
    - Several heuristics
    - ...
Outline

Basic Definitions

DPLL Solvers

CDCL Solvers

What Next in CDCL Solvers?

CNF Encodings
Outline

Basic Definitions

DPLL Solvers

CDCL Solvers

What Next in CDCL Solvers?

CNF Encodings
Preliminaries

- **Variables:** $w, x, y, z, a, b, c, \ldots$
- **Literals:** $w, \bar{x}, \bar{y}, a, \ldots$, but also $\neg w, \neg y, \ldots$
- **Clauses:** disjunction of literals or set of literals
- **Formula:** conjunction of clauses or set of clauses
- **Model (satisfying assignment):** partial/total mapping from variables to $\{0, 1\}$ that satisfies formula
- **Formula can be** SAT/UNSAT
Preliminaries

• Variables: \( w, x, y, z, a, b, c, \ldots \)
• Literals: \( w, \overline{x}, \overline{y}, a, \ldots \), but also \( \neg w, \neg y, \ldots \)
• Clauses: disjunction of literals or set of literals
• Formula: conjunction of clauses or set of clauses
• Model (satisfying assignment): partial/total mapping from variables to \( \{0, 1\} \) that satisfies formula
• Formula can be SAT/UNSAT
• Example:

\[
\mathcal{F} \triangleq (r) \land (\overline{r} \lor s) \land (\overline{w} \lor a) \land (\overline{x} \lor b) \land (\overline{y} \lor \overline{z} \lor c) \land (\overline{b} \lor \overline{c} \lor d)
\]

- Example models:
  - \( \{r, s, a, b, c, d\} \)
  - \( \{r, s, \overline{x}, y, \overline{w}, z, \overline{a}, b, c, d\} \)
Resolution

- Resolution rule:

\[
(\alpha \lor x) \quad (\beta \lor \neg x) \\
(\alpha \lor \beta)
\]

- Complete proof system for propositional logic
Resolution

- Resolution rule:

\[(\alpha \lor x) \quad (\beta \lor \bar{x}) \quad \overline{\quad (\alpha \lor \beta) \quad} \]

- Complete proof system for propositional logic

\[
\begin{align*}
(x \lor a) & \quad \bar{x} \lor a & \quad \bar{y} \lor \bar{a} & \quad y \lor \bar{a} \\
(a) & \quad \bar{a} \quad \bot
\end{align*}
\]

- Extensively used with (CDCL) SAT solvers
Resolution

- Resolution rule:

\[
\frac{(\alpha \lor x) \quad (\beta \lor \bar{x})}{(\alpha \lor \beta)}
\]

- Complete proof system for propositional logic

\[
\begin{array}{cccc}
(x \lor a) & (\bar{x} \lor a) & (\bar{y} \lor \bar{a}) & (y \lor \bar{a}) \\
(a) & (\bar{a}) & \\
\perp & \\
\end{array}
\]

- Extensively used with (CDCL) SAT solvers

- Self-subsuming resolution (with \(\alpha' \subseteq \alpha\)):

\[
\frac{(\alpha \lor x) \quad (\alpha' \lor \bar{x})}{(\alpha)}
\]

- \((\alpha)\) subsumes \((\alpha \lor x)\)
Unit Propagation

\[ \mathcal{F} = (r) \land (\overline{r} \lor s) \land (\overline{w} \lor a) \land (\overline{x} \lor \overline{a} \lor b) \land (\overline{y} \lor \overline{z} \lor c) \land (\overline{b} \lor \overline{c} \lor d) \]
Unit Propagation

\[ F = (r) \land (\bar{r} \lor s) \land \\
    (\bar{w} \lor a) \land (\bar{x} \lor \bar{a} \lor b) \land \\
    (\bar{y} \lor \bar{z} \lor c) \land (\bar{b} \lor \bar{c} \lor d) \]

- Decisions / Variable Branchings:
  \[ w = 1, x = 1, y = 1, z = 1 \]
Unit Propagation

\[ \mathcal{F} = (r) \land (\bar{r} \lor s) \land \\
(\bar{w} \lor a) \land (\bar{x} \lor \bar{a} \lor b) \\
(\bar{y} \lor \bar{z} \lor c) \land (\bar{b} \lor \bar{c} \lor d) \]

- **Decisions / Variable Branchings:**
  \[ w = 1, x = 1, y = 1, z = 1 \]

<table>
<thead>
<tr>
<th>Level</th>
<th>Dec.</th>
<th>Unit Prop.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>\emptyset</td>
<td>( r \rightarrow s )</td>
</tr>
<tr>
<td>1</td>
<td>( w )</td>
<td>( a )</td>
</tr>
<tr>
<td>2</td>
<td>( x )</td>
<td>( b )</td>
</tr>
<tr>
<td>3</td>
<td>( y )</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>( z )</td>
<td>( c \rightarrow d )</td>
</tr>
</tbody>
</table>
Unit Propagation

\[ \mathcal{F} = (r) \land (\bar{r} \lor s) \land (\bar{w} \lor a) \land (\bar{x} \lor \bar{a} \lor b) \land (\bar{y} \lor \bar{z} \lor c) \land (\bar{b} \lor \bar{c} \lor d) \]

- Decisions / Variable Branchings:
  \[ w = 1, \ x = 1, \ y = 1, \ z = 1 \]

- Additional definitions:
  - Antecedent (or reason) of an implied assignment
    - \((\bar{b} \lor \bar{c} \lor d)\) for \(d\)
  - Associate assignment with decision levels
    - \(w = 1 @ 1, \ x = 1 @ 2, \ y = 1 @ 3, \ z = 1 @ 4\)
    - \(r = 1 @ 0, \ d = 1 @ 4, \ldots\)
Resolution Proofs

• Refutation of unsatisfiable formula by iterated resolution operations produces resolution proof

• An example:
  \[ F = (\overline{c}) \land (\overline{b}) \land (\overline{a} \lor c) \land (a \lor b) \land (a \lor \overline{d}) \land (\overline{a} \lor \overline{d}) \]

• Resolution proof:

\[
\begin{array}{cccc}
(a \lor b) & (\overline{a} \lor c) \\
(b \lor c) & \\
(\overline{b}) & (b) \\
\big\downarrow & \\
\bot & \\
\end{array}
\]

• A modern SAT solver can generate resolution proofs using clauses learned by the solver

[ZM03]
Unsatisfiable Cores & Proof Traces

• CNF formula:

\[ \mathcal{F} = (\overline{c}) \land (\overline{b}) \land (\overline{a} \lor c) \land (a \lor b) \land (a \lor \overline{d}) \land (\overline{a} \lor \overline{d}) \]

<table>
<thead>
<tr>
<th>Level</th>
<th>Dec.</th>
<th>Unit Prop.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>\emptyset</td>
<td>\overline{b} \rightarrow a</td>
</tr>
</tbody>
</table>

\[ \overline{c} \rightarrow \bot \]

Implication graph with conflict
Unsatisfiable Cores & Proof Traces

- CNF formula:

\[ \mathcal{F} = (\neg c) \land (\neg b) \land (\neg a \lor c) \land (a \lor b) \land (a \lor \neg d) \land (\neg a \lor \neg d) \]

<table>
<thead>
<tr>
<th>Level</th>
<th>Dec.</th>
<th>Unit Prop.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>∅</td>
<td>( \neg b \rightarrow a )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \neg c \rightarrow \bot )</td>
</tr>
</tbody>
</table>

Proof trace \( \bot: (\neg a \lor c) \ (a \lor b) \ (\neg c) \ (\neg b) \)
Unsatisfiable Cores & Proof Traces

- CNF formula:

\[
\mathcal{F} = (\neg c) \land (\neg b) \land (\neg a \lor c) \land (a \lor b) \land (a \lor \neg d) \land (\neg a \lor \neg d)
\]

Resolution proof follows structure of conflicts
Unsatisfiable Cores & Proof Traces

- CNF formula:

\[ \mathcal{F} = (\bar{c}) \land (\bar{b}) \land (\bar{a} \lor c) \land (a \lor b) \land (a \lor \bar{d}) \land (\bar{a} \lor \bar{d}) \]

<table>
<thead>
<tr>
<th>Level</th>
<th>Dec.</th>
<th>Unit Prop.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>\emptyset</td>
<td>\bar{b} \rightarrow a</td>
</tr>
</tbody>
</table>

Unsatisfiable subformula (core): \((\bar{c}), (\bar{b}), (\bar{a} \lor c), (a \lor b)\)
Outline

Basic Definitions

DPLL Solvers

CDCL Solvers

What Next in CDCL Solvers?

CNF Encodings
The DPLL Algorithm

- Optional: pure literal rule
The DPLL Algorithm

\[ F = (x \lor y) \land (a \lor b) \land (\bar{a} \lor b) \land (a \lor \bar{b}) \land (\bar{a} \lor \bar{b}) \]

- Optional: pure literal rule
The DPLL Algorithm

- Optional: pure literal rule

\[ F = (x \lor y) \land (a \lor b) \land (\bar{a} \lor b) \land (a \lor \bar{b}) \land (\bar{a} \lor \bar{b}) \]

<table>
<thead>
<tr>
<th>Level</th>
<th>Dec.</th>
<th>Unit Prop.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( \emptyset )</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>( x )</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>( y )</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>( a \rightarrow b \rightarrow \perp )</td>
<td></td>
</tr>
</tbody>
</table>
The DPLL Algorithm

- Optional: pure literal rule
The DPLL Algorithm

- Optional: pure literal rule

\[ \mathcal{F} = (x \lor y) \land (a \lor b) \land (\overline{a} \lor b) \land (a \lor \overline{b}) \land (\overline{a} \lor \overline{b}) \]

<table>
<thead>
<tr>
<th>Level</th>
<th>Dec.</th>
<th>Unit Prop.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>\emptyset</td>
<td>\nothing</td>
</tr>
<tr>
<td>1</td>
<td>(x)</td>
<td>\nothing</td>
</tr>
<tr>
<td>2</td>
<td>(\overline{y})</td>
<td>\nothing</td>
</tr>
<tr>
<td>3</td>
<td>(a)</td>
<td>(b)</td>
</tr>
</tbody>
</table>

\begin{align*}
\text{Unassigned variables?} & \\
\text{Y} & \rightarrow \text{Satisfiable} \\
\text{No} & \\
\text{Assign value to variable} & \\
\text{Unit propagation} & \\
\text{Conflict?} & \\
\text{Y} & \\
\text{Can undo decision?} & \\
\text{No} & \\
\text{Unsatisfiable} & \\
\text{Backtrack & flip variable} & \end{align*}
The DPLL Algorithm

- Optional: pure literal rule

\[ F = (x \lor y) \land (a \lor b) \land (\bar{a} \lor b) \land (a \lor \bar{b}) \land (\bar{a} \lor \bar{b}) \]

<table>
<thead>
<tr>
<th>Level</th>
<th>Dec.</th>
<th>Unit Prop.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>\emptyset</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>(x)</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>(\bar{y})</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>(\bar{a})</td>
<td>(\bar{b})</td>
</tr>
</tbody>
</table>
The DPLL Algorithm

- Optional: pure literal rule

\[ F = (x \lor y) \land (a \lor b) \land (\bar{a} \lor b) \land (a \lor \bar{b}) \land (\bar{a} \lor \bar{b}) \]
The DPLL Algorithm

- Optional: pure literal rule

\[ \mathcal{F} = (x \lor y) \land (a \lor b) \land (\overline{a} \lor b) \land (a \lor \overline{b}) \land (\overline{a} \lor \overline{b}) \]

<table>
<thead>
<tr>
<th>Level</th>
<th>Dec.</th>
<th>Unit Prop.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>\emptyset</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>\overline{x}</td>
<td>y</td>
</tr>
<tr>
<td>2</td>
<td>\overline{a}</td>
<td>\overline{b}</td>
</tr>
</tbody>
</table>

Level Dec. Unit Prop.
Outline

Basic Definitions

DPLL Solvers

CDCL Solvers

What Next in CDCL Solvers?

CNF Encodings
What is a CDCL SAT Solver?

- Extend **DPLL** SAT solver with:
  - Clause learning & non-chronological backtracking

  - Search restarts
  - Lazy data structures
  - Conflict-guided branching

- ...
What is a CDCL SAT Solver?

- Extend **DPLL SAT** solver with:
  - Clause learning & non-chronological backtracking
    - Exploit UIPs
    - Minimize learned clauses
    - Opportunistically delete clauses
  - Search restarts
  - Lazy data structures
    - Watched literals
  - Conflict-guided branching
    - Lightweight branching heuristics
    - Phase saving
  - ...

References:
- [DP60, DLL62]
- [MSS96, BS97, Z97]
- [MSS96, SSS12]
- [SB09, VG09]
- [MSS96, MSS99, GN02]
- [GSK98, BMS00, H07, B08]
- [MMZZM01]
- [MMZZM01]
- [PD07]
How Significant are CDCL SAT Solvers?

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)
- Lingeling 587f (2011)

GRASP

DPLL
Outline

Basic Definitions

DPLL Solvers

CDCL Solvers
  Clause Learning, UIPs & Minimization
  Search Restarts & Lazy Data Structures

What Next in CDCL Solvers?

CNF Encodings
**Clause Learning**

<table>
<thead>
<tr>
<th>Level</th>
<th>Dec.</th>
<th>Unit Prop.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>∅</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>y</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>z</td>
<td>a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>b</td>
</tr>
</tbody>
</table>

- **Analyze conflict**
  - **Reasons:**
    - $x$ and $z$ → Decision variable & literals assigned at lower decision levels
  - Create new clause: $(\overline{x} \lor \overline{z})$

- Can relate clause learning with resolution
  - Learned clauses result from (selected) resolution operations
# Clause Learning

<table>
<thead>
<tr>
<th>Level</th>
<th>Dec.</th>
<th>Unit Prop.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>∅</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>y</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>z</td>
<td>a</td>
</tr>
</tbody>
</table>

- Analyze conflict

- Reasons:
  - Decision variable & literals assigned at lower decision levels
  - Create new clause: \( \lnot x \lor \lnot z \)

- Can relate clause learning with resolution
  - Learned clauses result from selected resolution operations
### Clause Learning

<table>
<thead>
<tr>
<th>Level</th>
<th>Dec.</th>
<th>Unit Prop.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>∅</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>y</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>z</td>
<td>a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>b</td>
</tr>
<tr>
<td></td>
<td></td>
<td>⊥</td>
</tr>
</tbody>
</table>

- **Analyze conflict**
  - **Reasons:** $x$ and $z$
    - Decision variable & literals assigned at lower decision levels

- Create new clause: $(\neg x \lor \neg z)$
Clause Learning

<table>
<thead>
<tr>
<th>Level</th>
<th>Dec.</th>
<th>Unit Prop</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>∅</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>y</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>z</td>
<td>a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>b</td>
</tr>
</tbody>
</table>

- **Analyze conflict**
  - Reasons: $x$ and $z$
    - Decision variable & literals assigned at lower decision levels
  - Create **new** clause: $(\overline{x} \lor \overline{z})$
Clause Learning

<table>
<thead>
<tr>
<th>Level</th>
<th>Dec.</th>
<th>Unit Prop.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>∅</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>y</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>z</td>
<td>a</td>
</tr>
</tbody>
</table>

(¬a ∨ b) (¬z ∨ b) (¬x ∨ ¬z ∨ a)

• Analyze conflict
  – Reasons: x and z
    ▶ Decision variable & literals assigned at lower decision levels
  – Create new clause: (¬x ∨ ¬z)

• Can relate clause learning with resolution
Clause Learning

<table>
<thead>
<tr>
<th>Level</th>
<th>Dec.</th>
<th>Unit Prop.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>∅</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>y</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>z</td>
<td>a</td>
</tr>
</tbody>
</table>

- Analyze conflict
  - Reasons: $x$ and $z$
    - Decision variable & literals assigned at lower decision levels
  - Create **new** clause: $(\bar{x} \lor \bar{z})$

- Can relate clause learning with resolution

$(\bar{a} \lor \bar{b})$  $(\bar{z} \lor b)$  $(\bar{x} \lor \bar{z} \lor a)$

$(\bar{a} \lor \bar{z})$  $(\bar{x} \lor \bar{z})$
Clause Learning

<table>
<thead>
<tr>
<th>Level</th>
<th>Dec.</th>
<th>Unit Prop.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>∅</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>y</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>z</td>
<td>a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>b</td>
</tr>
</tbody>
</table>

- **Analyze conflict**
  - Reasons: x and z
    - Decision variable & literals assigned at lower decision levels
  - Create **new** clause: $(\overline{x} \lor \overline{z})$

- **Can relate clause learning with resolution**
**Clause Learning**

<table>
<thead>
<tr>
<th>Level</th>
<th>Dec.</th>
<th>Unit Prop.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>∅</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>y</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>z</td>
<td>a</td>
</tr>
</tbody>
</table>

- Analyze conflict
  - Reasons: x and z
    - Decision variable & literals assigned at lower decision levels
  - Create **new** clause: \((\bar{x} \lor \bar{z})\)

- Can relate **clause learning** with **resolution**
  - Learned clauses result from **selected** resolution operations
Clause Learning – After Backtracking

<table>
<thead>
<tr>
<th>Level</th>
<th>Dec.</th>
<th>Unit Prop.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>∅</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>y</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>z</td>
<td>a</td>
</tr>
</tbody>
</table>

- Clause (¬x ∨ ¬z) is asserting at decision level 1
- Learned clauses are always asserting [MSS96, MSS99]
- Backtracking differs from plain DPLL:
  - Always backtrack after a conflict [MMZZM01]
Clause Learning – After Backtracking

<table>
<thead>
<tr>
<th>Level</th>
<th>Dec.</th>
<th>Unit Prop.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>∅</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>y</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>z</td>
<td></td>
</tr>
</tbody>
</table>

- Clause $(\overline{x} \lor \overline{z})$ is **asserting** at decision level 1

Learned clauses are always asserting \([MSS96,MSS99]\)

Backtracking differs from plain DPLL:

- Always backtrack after a conflict \([MMZZM01]\)
Clause Learning – After Backtracking

- Clause \((\bar{x} \lor \bar{z})\) is asserting at decision level 1

Learned clauses are always asserting \([MSS96, MSS99]\)

Backtracking differs from plain DPLL:
- Always backtrack after a conflict \([MMZZM01]\)
Clause Learning – After Backtracking

- Clause \((\overline{x} \lor \overline{z})\) is asserting at decision level 1
- Learned clauses are always asserting
- Backtracking differs from plain DPLL:
  - Always backtrack after a conflict

\[\text{Level} \quad \text{Dec.} \quad \text{Unit Prop.}\]
\[
\begin{array}{ccc}
0 & \emptyset & 0 & \emptyset & \emptyset \\
1 & x & 1 & x & \overline{z} \\
2 & y & 2 & y & \bot \\
3 & z & 3 & z & \bot \\
\end{array}
\]

\[\text{[MSS96,MSS99]}\]

\[\text{[MMZZM01]}\]
Unique Implication Points (UIPs)

<table>
<thead>
<tr>
<th>Level</th>
<th>Dec.</th>
<th>Unit Prop.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$\emptyset$</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>$w$</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>$x$</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>$y$</td>
<td>$a \rightarrow c$</td>
</tr>
<tr>
<td>4</td>
<td>$z \rightarrow a$</td>
<td>$b \rightarrow \bot$</td>
</tr>
</tbody>
</table>
Unique Implication Points (UIPs)

<table>
<thead>
<tr>
<th>Level</th>
<th>Dec.</th>
<th>Unit Prop.</th>
<th>Clause</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>∅</td>
<td>w</td>
<td>$(\bar{b} \lor \bar{c})$</td>
</tr>
<tr>
<td>1</td>
<td>w</td>
<td>w</td>
<td>$(\bar{w} \lor \bar{a} \lor c)$</td>
</tr>
<tr>
<td>2</td>
<td>x</td>
<td>x</td>
<td>$(\bar{x} \lor \bar{a} \lor b)$</td>
</tr>
<tr>
<td>3</td>
<td>y</td>
<td>y</td>
<td>$(\bar{y} \lor \bar{z} \lor a)$</td>
</tr>
<tr>
<td>4</td>
<td>z</td>
<td>a</td>
<td>$(\bar{w} \lor \bar{x} \lor \bar{y} \lor \bar{z})$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>b</td>
<td>$(\bar{w} \lor \bar{x} \lor \bar{a} \lor \bar{b})$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>\bot</td>
<td>$(\bar{w} \lor \bar{x} \lor \bar{a} \lor \bar{b})$</td>
</tr>
</tbody>
</table>

- Learn clause $(\bar{w} \lor \bar{x} \lor \bar{y} \lor \bar{z})$
Unique Implication Points (UIPs)

<table>
<thead>
<tr>
<th>Level</th>
<th>Dec.</th>
<th>Unit Prop.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>∅</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>w</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>y</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>z</td>
<td>a</td>
</tr>
</tbody>
</table>

- Learn clause \((\overline{w} \lor \overline{x} \lor \overline{y} \lor \overline{z})\)
- But \(a\) is an UIP
  - Dominator in DAG for level 4
## Unique Implication Points (UIPs)

<table>
<thead>
<tr>
<th>Level</th>
<th>Dec.</th>
<th>Unit Prop.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>∅</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>( w )</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>( x )</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>( y )</td>
<td>( a )</td>
</tr>
<tr>
<td>4</td>
<td>( z )</td>
<td>( a )</td>
</tr>
</tbody>
</table>

- **Learn clause** \((\bar{w} \lor \bar{x} \lor \bar{y} \lor \bar{z})\)
- **But** \( a \) **is an UIP**
  - Dominator in DAG for level 4
- **Learn clause** \((\bar{w} \lor \bar{x} \lor \bar{a})\)
Multiple UIPs

<table>
<thead>
<tr>
<th>Level</th>
<th>Dec.</th>
<th>Unit Prop.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>∅</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>w</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>y</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>z</td>
<td>r</td>
</tr>
</tbody>
</table>

- First UIP:
  - Learn clause ($\overline{w} \lor \overline{x} \lor \overline{a}$)
- But there can be more than 1 UIP
- Second UIP:
  - Learn clause ($\overline{x} \lor \overline{z} \lor a$)
- In practice smaller clauses more effective
  - Compare with ($\overline{w} \lor \overline{x} \lor \overline{y} \lor \overline{z}$)

Multiple UIPs proposed in GRASP [MSS96]
- First UIP learning proposed in Chaff [MMZZM01]
- Not used in recent state of the art CDCL SAT solvers
- Recent results show it can be beneficial on current instances [SSS12]
Multiple UIPs

- First UIP:
  - Learn clause $(\bar{w} \lor \bar{x} \lor \bar{a})$

```
Level | Dec. | Unit Prop.
----- | ---- | --------
0     | $\emptyset$ | |
1     | $w$   | |
2     | $x$   | |
3     | $y$   | |
4     | $z$   | |
```

- Second UIP:
  - Learn clause $(\bar{x} \lor \bar{z} \lor a)$

- Multiple UIPs proposed in GRASP [MSS96]
- First UIP learning proposed in Chaff [MMZZM01]
- Not used in recent state of the art CDCL SAT solvers
- Recent results show it can be beneficial on current instances [SSS12]
Multiple UIPs

- First UIP:
  - Learn clause \((\bar{w} \lor \bar{x} \lor \bar{a})\)
- But there can be more than 1 UIP

<table>
<thead>
<tr>
<th>Level</th>
<th>Dec.</th>
<th>Unit Prop.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>∅</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>(w)</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>(x)</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>(y)</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>(z)</td>
<td>(r) (\rightarrow) (a) (\rightarrow) (c)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(s) (\rightarrow) (b) (\rightarrow) ⊥</td>
</tr>
</tbody>
</table>
Multiple UIPs

- **First UIP:**
  - Learn clause \((\overline{w} \lor \overline{x} \lor \overline{a})\)
- But there can be more than 1 UIP
- **Second UIP:**
  - Learn clause \((\overline{x} \lor \overline{z} \lor a)\)

In practice, smaller clauses are more effective. Comparing with \((\overline{w} \lor \overline{x} \lor \overline{y} \lor \overline{z})\), multiple UIPs proposed in GRASP [MSS96]

First UIP learning proposed in Chaff [MMZZM01]

Not used in recent state of the art CDCL SAT solvers

Recent results show it can be beneficial on current instances [SSS12]
Multiple UIPs

<table>
<thead>
<tr>
<th>Level</th>
<th>Dec.</th>
<th>Unit Prop.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>∅</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>w</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>y</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>z</td>
<td>r</td>
</tr>
</tbody>
</table>

- **First UIP:**
  - Learn clause \((\bar{w} \lor \bar{x} \lor \bar{a})\)

- But there can be more than 1 UIP

- **Second UIP:**
  - Learn clause \((\bar{x} \lor \bar{z} \lor a)\)

- In practice smaller clauses more effective
  - Compare with \((\bar{w} \lor \bar{x} \lor \bar{y} \lor \bar{z})\)

---

Multiple UIPs proposed in GRASP [MSS96]

- First UIP learning proposed in Chaff [MMZZM01]

- Not used in recent state of the art CDCL SAT solvers

- Recent results show it can be beneficial on current instances [SSS12]
Multiple UIPs

- First UIP:
  - Learn clause \( (\bar{w} \lor \bar{x} \lor \bar{a}) \)
- But there can be more than 1 UIP
- Second UIP:
  - Learn clause \( (\bar{x} \lor \bar{z} \lor a) \)
  - In practice smaller clauses more effective
    - Compare with \( (\bar{w} \lor \bar{x} \lor \bar{y} \lor \bar{z}) \)

- Multiple UIPs proposed in GRASP
  - First UIP learning proposed in Chaff
- Not used in recent state of the art CDCL SAT solvers

[MSS96] [MMZZM01] [SSS12]
## Multiple UIPs

<table>
<thead>
<tr>
<th>Level</th>
<th>Dec.</th>
<th>Unit Prop.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$\emptyset$</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>$w$</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>$x$</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>$y$</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>$z$</td>
<td></td>
</tr>
</tbody>
</table>

- **First UIP:**
  - Learn clause \((\bar{w} \lor \bar{x} \lor \bar{a})\)

- But there can be more than 1 UIP

- **Second UIP:**
  - Learn clause \((\bar{x} \lor \bar{z} \lor a)\)

- In practice smaller clauses more effective
  - Compare with \((\bar{w} \lor \bar{x} \lor \bar{y} \lor \bar{z})\)

- Multiple UIPs proposed in GRASP
  - First UIP learning proposed in Chaff

- Not used in recent state of the art CDCL SAT solvers

- Recent results show it can be beneficial on current instances

---

[MSS96]

[MMZZM01]

[SSS12]
### Clause Minimization I

<table>
<thead>
<tr>
<th>Level</th>
<th>Dec.</th>
<th>Unit Prop.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( \emptyset )</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>( x )</td>
<td>( b )</td>
</tr>
<tr>
<td>2</td>
<td>( y )</td>
<td>( \downarrow )</td>
</tr>
<tr>
<td>3</td>
<td>( z )</td>
<td>( c )</td>
</tr>
</tbody>
</table>

![Diagram](attachment:image.png)
• Learn clause \((\overline{x} \lor \overline{y} \lor \overline{z} \lor \overline{b})\)
• Learn clause \((\bar{x} \lor \bar{y} \lor \bar{z} \lor \bar{b})\)

• Apply self-subsuming resolution (i.e. local minimization) [SB09]
**Clause Minimization I**

<table>
<thead>
<tr>
<th>Level</th>
<th>Dec.</th>
<th>Unit Prop.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>∅</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>x</td>
<td>b</td>
</tr>
<tr>
<td>2</td>
<td>y</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>z</td>
<td>c</td>
</tr>
<tr>
<td></td>
<td></td>
<td>⊥</td>
</tr>
</tbody>
</table>

- Learn clause \((\overline{x} \lor \overline{y} \lor \overline{z} \lor \overline{b})\)
- Apply self-subsuming resolution (i.e. local minimization)
Clause Minimization I

<table>
<thead>
<tr>
<th>Level</th>
<th>Dec.</th>
<th>Unit Prop.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>∅</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>x</td>
<td>b</td>
</tr>
<tr>
<td>2</td>
<td>y</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>z</td>
<td>c</td>
</tr>
</tbody>
</table>

- Learn clause $(\bar{x} \lor \bar{y} \lor \bar{z} \lor \bar{b})$
- Apply self-subsuming resolution (i.e. local minimization)
- Learn clause $(\bar{x} \lor \bar{y} \lor \bar{z})$
<table>
<thead>
<tr>
<th>Level</th>
<th>Dec.</th>
<th>Unit Prop.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$\emptyset$</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>$w$</td>
<td>$a$</td>
</tr>
<tr>
<td></td>
<td>$b$</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>$x$</td>
<td>$e$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$d$</td>
</tr>
</tbody>
</table>

Cannot apply self-subsuming resolution – Resolving with reason of $c$ yields $(\neg w \lor \neg x \lor \neg a \lor \neg b)$.

Can apply recursive minimization.

Marked nodes: literals in learned clause [SB09].

Trace back from $c$ until marked nodes or new decision nodes.

– Drop literal $c$ if only marked nodes visited.

Complexity of recursive minimization?
Learn clause \((\bar{w} \lor \bar{x} \lor \bar{c})\)

- Complexit of recursive minimization?
Clause Minimization II

<table>
<thead>
<tr>
<th>Level</th>
<th>Dec.</th>
<th>Unit Prop.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(\emptyset)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>(w)</td>
<td>(a) → (c)</td>
</tr>
<tr>
<td>2</td>
<td>(x)</td>
<td>(e) ← (d) ← (\bot)</td>
</tr>
</tbody>
</table>

- Learn clause \((\overline{w} \lor \overline{x} \lor \overline{c})\)
- **Cannot** apply self-subsuming resolution
  - Resolving with reason of \(c\) yields \((\overline{w} \lor \overline{x} \lor \overline{a} \lor \overline{b})\)
### Clause Minimization II

<table>
<thead>
<tr>
<th>Level</th>
<th>Dec.</th>
<th>Unit Prop.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$\emptyset$</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>$w$</td>
<td>$a$ $c$</td>
</tr>
<tr>
<td></td>
<td>$b$</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>$x$</td>
<td>$e$</td>
</tr>
<tr>
<td></td>
<td>$d$</td>
<td>$\perp$</td>
</tr>
</tbody>
</table>

- Learn clause $(\bar{w} \lor \bar{x} \lor \bar{c})$
- **Cannot** apply self-subsuming resolution
  - Resolving with reason of $c$ yields $(\bar{w} \lor \bar{x} \lor \bar{a} \lor \bar{b})$
- Can apply **recursive minimization**
Clause Minimization II

<table>
<thead>
<tr>
<th>Level</th>
<th>Dec.</th>
<th>Unit Prop.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>∅</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>w</td>
<td>a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>c</td>
</tr>
<tr>
<td>2</td>
<td>x</td>
<td>e</td>
</tr>
<tr>
<td></td>
<td></td>
<td>d</td>
</tr>
<tr>
<td></td>
<td></td>
<td>⊥</td>
</tr>
</tbody>
</table>

- Learn clause \((\overline{w} \lor \overline{x} \lor \overline{c})\)
- **Cannot** apply self-subsuming resolution
  - Resolving with reason of \(c\) yields \((\overline{w} \lor \overline{x} \lor \overline{a} \lor \overline{b})\)
- Can apply **recursive minimization**

- **Marked nodes**: literals in learned clause

[SB09]
Clause Minimization II

Learn clause \((\bar{w} \lor \bar{x} \lor \bar{c})\)

Cannot apply self-subsuming resolution
- Resolving with reason of \(c\) yields \((\bar{w} \lor \bar{x} \lor \bar{a} \lor \bar{b})\)

Can apply recursive minimization

- Marked nodes: literals in learned clause
- Trace back from \(c\) until marked nodes or new decision nodes
  - Drop literal \(c\) if only marked nodes visited

Marked nodes: literals in learned clause

[SB09]
Clause Minimization II

<table>
<thead>
<tr>
<th>Level</th>
<th>Dec.</th>
<th>Unit Prop.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>∅</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>w</td>
<td>a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>c</td>
</tr>
<tr>
<td>2</td>
<td>x</td>
<td>e</td>
</tr>
</tbody>
</table>
|       |      | d          | \perp

- **Learn clause** \((\overline{w} \lor \overline{x} \lor \overline{c})\)
- **Cannot** apply self-subsuming resolution
  - Resolving with reason of \(c\) yields \((\overline{w} \lor \overline{x} \lor \overline{a} \lor \overline{b})\)
- **Can apply** recursive minimization
- **Learn clause** \((\overline{w} \lor \overline{x})\)

- **Marked nodes**: literals in learned clause
- **Trace back from** \(c\) until marked nodes or new decision nodes
  - Drop literal \(c\) if only marked nodes visited

[SB09]
Learn clause $((\overline{w} \lor \overline{x} \lor \overline{c})$

Cannot apply self-subsuming resolution
- Resolving with reason of $c$ yields $(\overline{w} \lor \overline{x} \lor \overline{a} \lor \overline{b})$

Can apply recursive minimization
Learn clause $(\overline{w} \lor \overline{x})$

Marked nodes: literals in learned clause
Trace back from $c$ until marked nodes or new decision nodes
- Drop literal $c$ if only marked nodes visited
Complexity of recursive minimization?
Outline

Basic Definitions

DPLL Solvers

CDCL Solvers
  Clause Learning, UIPs & Minimization
  Search Restarts & Lazy Data Structures

What Next in CDCL Solvers?

CNF Encodings
Search Restarts I

- Heavy-tail behavior:

- 10000 runs, branching randomization on industrial instance
  - Use rapid randomized restarts (search restarts)
Search Restarts II

- Restart search after a number of conflicts

- Increase cutoff after each restart
  - Guarantees completeness
  - Different policies exist (see refs)

- Works for SAT & UNSAT instances. Why?
  - Learned clauses effective after restart(s)
Search Restarts II

- Restart search after a number of conflicts
- Increase \textit{cutoff} after each restart
  - Guarantees completeness
  - Different policies exist (see refs)

\[\text{solution} \rightarrow \text{cutoff} \rightarrow \text{cutoff} \rightarrow \text{solution}\]
Search Restarts II

- Restart search after a number of conflicts
- Increase **cutoff** after each restart
  - Guarantees completeness
  - Different policies exist (see refs)
- Works for **SAT** & **UNSAT** instances. **Why?**
Search Restarts II

- Restart search after a number of conflicts
- Increase \textit{cutoff} after each restart
  - Guarantees completeness
  - Different policies exist (see refs)
- Works for \textit{SAT} & \textit{UNSAT} instances. Why?
- Learned clauses effective after restart(s)
• Each literal should access clauses containing
  – Why?
Each literal \( l \) should access clauses containing \( l \)
  - Why? Unit propagation
• Each literal \( l \) should access clauses containing \( l \)
  – Why? Unit propagation

• Clause with \( k \) literals results in \( k \) references, from literals to the clause

- Worst-case size: \( O(n) \)
- Worst-case number of literals: \( O(m \cdot n) \)
- In practice, Unit propagation slow-down worse than linear as clauses are learned!
Data Structures Basics

- Each literal \( l \) should access clauses containing \( l \)
  - Why? Unit propagation

- Clause with \( k \) literals results in \( k \) references, from literals to the clause

- Number of clause references equals number of literals, \( L \)
Data Structures Basics

- Each literal \( l \) should access clauses containing \( l \)
  - Why? Unit propagation

- Clause with \( k \) literals results in \( k \) references, from literals to the clause

- Number of clause references equals number of literals, \( L \)
  - Clause learning can generate large clauses
    - Worst-case size: \( \mathcal{O}(n) \)

- Worst-case number of literals: \( \mathcal{O}(m n) \)
- In practice, Unit propagation slow-down worse than linear as clauses are learned!

- Clause learning to be effective requires a more efficient representation:
  - Watched literals are one example of lazy data structures
  - But there are others
Each literal $l$ should access clauses containing $l$
  
  Why? Unit propagation

Clause with $k$ literals results in $k$ references, from literals to the clause

Number of clause references equals number of literals, $L$
  
  Clause learning can generate large clauses
    
    Worst-case size: $O(n)$
  
  Worst-case number of literals: $O(mn)$
Data Structures Basics

• Each literal \( l \) should access clauses containing \( l \)
  – Why? Unit propagation

• Clause with \( k \) literals results in \( k \) references, from literals to the clause

• Number of clause references equals number of literals, \( L \)
  – Clause learning can generate large clauses
    ▶ Worst-case size: \( O(n) \)
  – Worst-case number of literals: \( O(mn) \)
  – In practice,
    Unit propagation slow-down worse than linear as clauses are learned


Data Structures Basics

- Each literal \( l \) should access clauses containing \( l \)
  - Why? Unit propagation

- Clause with \( k \) literals results in \( k \) references, from literals to the clause

- Number of clause references equals number of literals, \( L \)
  - Clause learning can generate large clauses
    - Worst-case size: \( \mathcal{O}(n) \)
  - Worst-case number of literals: \( \mathcal{O}(mn) \)
  - In practice,
    - Unit propagation slow-down worse than linear as clauses are learned

- Clause learning to be effective requires a more efficient representation:
• Each literal \( l \) should access clauses containing \( l \)
  – Why? Unit propagation

• Clause with \( k \) literals results in \( k \) references, from literals to the clause

• Number of clause references equals number of literals, \( L \)
  – Clause learning can generate large clauses
    ▶ Worst-case size: \( O(n) \)
  – Worst-case number of literals: \( O(mn) \)
  – In practice,
    
    Unit propagation slow-down worse than linear as clauses are learned !

• Clause learning to be effective requires a more efficient representation: \textbf{Watched Literals}
Data Structures Basics

- Each literal \( l \) should access clauses containing \( l \)
  - Why? Unit propagation

- Clause with \( k \) literals results in \( k \) references, from literals to the clause

- Number of clause references equals number of literals, \( L \)
  - Clause learning can generate large clauses
    - Worst-case size: \( \mathcal{O}(n) \)
  - Worst-case number of literals: \( \mathcal{O}(mn) \)
  - In practice,
    
    Unit propagation slow-down worse than linear as clauses are learned!

- Clause learning to be effective requires a more efficient representation: Watched Literals
  - Watched literals are one example of lazy data structures
    - But there are others
Watched Literals

- Important states of a clause
Watched Literals

- Important states of a clause
- Associate 2 references with each clause

[Diagram showing different states of a clause]

- unresolved
- unit
- satisfied
- after backtracking to level 4
Watched Literals

- Important states of a clause
- Associate 2 references with each clause
- Deciding unit requires traversing all literals
Watched Literals

- Important states of a clause
- Associate 2 references with each clause
- Deciding unit requires traversing all literals
- References **unchanged** when backtracking
In practice, first two positions of clause are watched.
Watched Literal, in Practice

- In practice, first two positions of clause are watched.
- May require to traverse already assigned literals, multiple times.

Fixed
Assign
watched
Assign
watched
Swap
Swap
• In practice, first two positions of clause are watched
• May require to traverse already assigned literals, multiple times
• Worst-case time of unit propagation is **quadratic** on clause size and so on number of literals
Watched Literals, in Practice

- In practice, first two positions of clause are watched
- May require to traverse already assigned literals, multiple times
- Worst-case time of unit propagation is **quadratic** on clause size and so on number of literals
- In practice, no gains observed from considering alternative implementations (see previous slide)
Additional Key Techniques

- **Lightweight branching** [e.g. MMZZM01]
  - Use conflict to bias variables to branch on, associate score with each variable
  - Prefer recent bias by regularly decreasing variable scores

- Clause deletion policies
  - Not practical to keep all learned clauses
  - Delete larger clauses [e.g. MSS96]
  - Delete less used clauses [e.g. GN02, ES03]

- Proven recent techniques:
  - Phase saving [PD07]
  - Luby restarts [H07]
  - Literal blocks distance [AS09]
Additional Key Techniques

- **Lightweight branching**
  - Use conflict to bias variables to branch on, associate score with each variable
  - Prefer recent bias by regularly decreasing variable scores

- **Clause deletion policies**
  - Not practical to keep all learned clauses
  - Delete larger clauses
  - Delete less used clauses
Additional Key Techniques

- **Lightweight branching**
  - Use conflict to bias variables to branch on, associate score with each variable
  - Prefer recent bias by regularly decreasing variable scores

- **Clause deletion policies**
  - Not practical to keep all learned clauses
  - Delete larger clauses
  - Delete less used clauses

- **Proven recent techniques:**
  - Phase saving
  - Luby restarts
  - Literal blocks distance
Outline

Basic Definitions

DPLL Solvers

CDCL Solvers

What Next in CDCL Solvers?

CNF Encodings
• Clause learning techniques
  – Clause learning is the key technique in CDCL SAT solvers
  – Many recent papers propose improvements to the basic clause learning approach

[e.g. ABHJS08]
CDCL – A Glimpse of the Future

• **Clause learning techniques**
  - Clause learning is the key technique in CDCL SAT solvers
  - Many recent papers propose improvements to the basic clause learning approach

• **Preprocessing & inprocessing**
  - Many recent papers
  - Essential in some applications
• **Clause learning techniques**
  - Clause learning is the key technique in CDCL SAT solvers
  - Many recent papers propose improvements to the basic clause learning approach

• **Preprocessing & inprocessing**
  - Many recent papers
  - Essential in some applications

• **Application-driven improvements**
  - Incremental SAT
    - Handling of assumptions due to MUS extractors
    - Changing SAT solvers to better suit applications
But Also, SAT-Based Problem Solving

Problem Solving with SAT

Encodings
- Eager SMT
- BMC

Embeddings
- PBO
- B&B Search
- Enumeration
- OPT SAT
- Lazy SMT
- LCG

Oracles
- MC: ic3
- MaxSAT
- MUS
- MUS
- CEGAR QBF
- CEGAR SMT
- Enumeration
- Backbones
- Min. Models
- MCS
- MCS
- MCS
Outline

Basic Definitions

DPLL Solvers

CDCL Solvers

What Next in CDCL Solvers?

CNF Encodings
Encoding to CNF

• What to encode?
  – Boolean formulas
    ▶ Tseitin’s encoding
    ▶ Plaisted&Greenbaum’s encoding
    ▶ ...
  – Cardinality constraints
  – Pseudo-Boolean (PB) constraints
  – Can also translate to SAT:
    ▶ Constraint Satisfaction Problems (CSPs)
    ▶ Answer Set Programming (ASP)
    ▶ Model Finding
    ▶ ...

• Key issues:
  – Encoding size
  – Arc-consistency?
Outline

Basic Definitions

DPLL Solvers

CDCL Solvers

What Next in CDCL Solvers?

CNF Encodings
  Boolean Formulas
  Cardinality Constraints
  Pseudo-Boolean Constraints
  Encoding CSPs
Satisfiability problems can be defined on Boolean circuits/formulas. Can represent circuits/formulas as CNF formulas:

- For each (simple) gate, CNF formula encodes the consistent assignments to the gate’s inputs and output.
  - Given \( z = \text{OP}(x, y) \), represent in CNF \( z \leftrightarrow \text{OP}(x, y) \).
  - CNF formula for the circuit is the conjunction of CNF formula for each gate.

\[
\begin{align*}
\mathcal{F}_c &= (a \lor c) \land (b \lor c) \land (\bar{a} \lor \bar{b} \lor \bar{c}) \\
\mathcal{F}_t &= (\bar{r} \lor t) \land (\bar{s} \lor t) \land (r \lor s \lor \bar{t})
\end{align*}
\]
Representing Boolean Formulas / Circuits II

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

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>$F_c(a,b,c)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
• CNF formula for the circuit is the conjunction of the CNF formula for each gate
  - Can specify objectives with additional clauses

\[ F = (a \lor x) \land (b \lor x) \land (\bar{a} \lor \bar{b} \lor \bar{x}) \land \\
    (x \lor \bar{y}) \land (c \lor \bar{y}) \land (\bar{x} \lor \bar{c} \lor y) \land \\
    (\bar{y} \lor z) \land (\bar{d} \lor z) \land (y \lor d \lor \bar{z}) \land (z) \]
• CNF formula for the circuit is the conjunction of the CNF formula for each gate
  – Can specify objectives with additional clauses

\[ F = (a \lor x) \land (b \lor x) \land (\bar{a} \lor \bar{b} \lor \bar{x}) \land \\
  (x \lor \bar{y}) \land (c \lor \bar{y}) \land (\bar{x} \lor \bar{c} \lor y) \land \\
  (\bar{y} \lor z) \land (\bar{d} \lor z) \land (y \lor d \lor \bar{z}) \land (z) \]

• Note: \( z = d \lor (c \land (\neg (a \land b))) \)
  – No distinction between Boolean circuits and formulas
Outline

Basic Definitions

DPLL Solvers

CDCL Solvers

What Next in CDCL Solvers?

CNF Encodings
  Boolean Formulas
  Cardinality Constraints
  Pseudo-Boolean Constraints
  Encoding CSPs
Cardinality Constraints

• How to handle cardinality constraints, $\sum_{j=1}^{n} x_j \leq k$ ?
  - How to handle AtMost1 constraints, $\sum_{j=1}^{n} x_j \leq 1$ ?
  - General form: $\sum_{j=1}^{n} x_j \triangleright k$, with $\triangleright \in \{<, \leq, =, \geq, >\}$

• Solution #1:
  - Use native PB solver, e.g. BSOLO, PBS, Galena, Pueblo, etc.
  - Difficult to keep up with advances in SAT technology
  - For SAT/UNSAT, best solvers already encode to CNF
    - E.g. Minisat+, WBO, etc.
Cardinality Constraints

- How to handle cardinality constraints, \( \sum_{j=1}^{n} x_j \leq k \)?
  - How to handle AtMost1 constraints, \( \sum_{j=1}^{n} x_j \leq 1 \)?
  - General form: \( \sum_{j=1}^{n} x_j \triangleleft k \), with \( \triangleleft \in \{<, \leq, =, \geq, >\} \)

- Solution #1:
  - Use native PB solver, e.g. BSOLO, PBS, Galena, Pueblo, etc.
  - Difficult to keep up with advances in SAT technology
  - For SAT/UNSAT, best solvers already encode to CNF
    - E.g. Minisat+, WBO, etc.

- Solution #2:
  - Encode cardinality constraints to CNF
  - Use SAT solver
Equals 1, AtLeast 1 & AtMost 1 Constraints

- \( \sum_{j=1}^{n} x_j = 1 \): encode with \((\sum_{j=1}^{n} x_j \leq 1) \wedge (\sum_{j=1}^{n} x_j \geq 1)\)

- \( \sum_{j=1}^{n} x_j \geq 1 \): encode with \((x_1 \lor x_2 \lor \ldots \lor x_n)\)

- \( \sum_{j=1}^{n} x_j \leq 1 \) encode with:
  - Pairwise encoding
    - Clauses: \(O(n^2)\) ; No auxiliary variables
  - Sequential counter
    - Clauses: \(O(n)\) ; Auxiliary variables: \(O(n)\) [S05]
  - Bitwise encoding
    - Clauses: \(O(n \log n)\) ; Auxiliary variables: \(O(\log n)\) [P07,FP01]
  - ...
Bitwise Encoding

- Encode $\sum_{j=1}^{n} x_j \leq 1$ with bitwise encoding:

- An example: $x_1 + x_2 + x_3 \leq 1$
Bitwise Encoding

- Encode $\sum_{j=1}^{n} x_j \leq 1$ with bitwise encoding:
  - Auxiliary variables $v_0, \ldots, v_{r-1}$; $r = \lceil \log n \rceil$ (with $n > 1$)
  - If $x_j = 1$, then $v_0 \ldots v_{r-1} = b_0 \ldots b_{r-1}$, the binary encoding of $j - 1$
    \[ x_j \rightarrow (v_0 = b_0) \land \ldots \land (v_{r-1} = b_{r-1}) \Leftrightarrow (\bar{x}_j \lor (v_0 = b_0) \land \ldots \land (v_{r-1} = b_{r-1})) \]

- An example: $x_1 + x_2 + x_3 \leq 1$

<table>
<thead>
<tr>
<th>$j - 1$</th>
<th>$v_1 v_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_1$</td>
<td>0 0</td>
</tr>
<tr>
<td>$x_2$</td>
<td>1 0</td>
</tr>
<tr>
<td>$x_3$</td>
<td>2 1</td>
</tr>
</tbody>
</table>
Bitwise Encoding

- Encode $\sum_{j=1}^{n} x_j \leq 1$ with bitwise encoding:
  - Auxiliary variables $v_0, \ldots, v_{r-1}$; $r = \lceil \log n \rceil$ (with $n > 1$)
  - If $x_j = 1$, then $v_0 \ldots v_{r-1} = b_0 \ldots b_{r-1}$, the binary encoding of $j - 1$
    
    $x_j \rightarrow (v_0 = b_0) \land \ldots \land (v_{r-1} = b_{r-1}) \iff (\overline{x_j} \lor (v_0 = b_0) \land \ldots \land (v_{r-1} = b_{r-1}))$
  - Clauses $(\overline{x_j} \lor (v_i \leftrightarrow b_i)) = (\overline{x_j} \lor l_i)$, $i = 0, \ldots, r - 1$, where
    - $l_i \equiv v_i$, if $b_i = 1$
    - $l_i \equiv \overline{v_i}$, otherwise

- An example: $x_1 + x_2 + x_3 \leq 1$

<table>
<thead>
<tr>
<th></th>
<th>$j - 1$</th>
<th>$v_1 v_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_1$</td>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>$x_2$</td>
<td>1</td>
<td>01</td>
</tr>
<tr>
<td>$x_3$</td>
<td>2</td>
<td>10</td>
</tr>
</tbody>
</table>
Bitwise Encoding

- Encode \( \sum_{j=1}^{n} x_j \leq 1 \) with bitwise encoding:
  - Auxiliary variables \( v_0, \ldots, v_{r-1} \); \( r = \lceil \log n \rceil \) (with \( n > 1 \))
  - If \( x_j = 1 \), then \( v_0 \ldots v_{r-1} = b_0 \ldots b_{r-1} \), the binary encoding of \( j - 1 \)
    \( x_j \rightarrow (v_0 = b_0) \land \ldots \land (v_{r-1} = b_{r-1}) \iff (\bar{x}_j \lor (v_0 = b_0) \land \ldots \land (v_{r-1} = b_{r-1})) \)
  - Clauses \( (\bar{x}_j \lor (v_i \leftrightarrow b_i)) = (\bar{x}_j \lor l_i), \ i = 0, \ldots, r - 1 \), where
    - \( l_i \equiv v_i \), if \( b_i = 1 \)
    - \( l_i \equiv \bar{v}_i \), otherwise
  - If \( x_j = 1 \), assignment to \( v_i \) variables must encode \( j - 1 \)
    - All other \( x \) variables must take value 0
  - If all \( x_j = 0 \), any assignment to \( v_i \) variables is consistent
  - \( \mathcal{O}(n \log n) \) clauses ; \( \mathcal{O}(\log n) \) auxiliary variables

- An example: \( x_1 + x_2 + x_3 \leq 1 \)

<table>
<thead>
<tr>
<th></th>
<th>( j - 1 )</th>
<th>( v_1 v_0 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x_1 )</td>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>( x_2 )</td>
<td>1</td>
<td>01</td>
</tr>
<tr>
<td>( x_3 )</td>
<td>2</td>
<td>10</td>
</tr>
</tbody>
</table>

\( (\bar{x}_1 \lor \bar{v}_1) \land (\bar{x}_1 \lor \bar{v}_0) \)
\( (\bar{x}_2 \lor \bar{v}_1) \land (\bar{x}_2 \lor v_0) \)
\( (\bar{x}_3 \lor v_1) \land (\bar{x}_3 \lor \bar{v}_0) \)
General Cardinality Constraints

- General form: $\sum_{j=1}^{n} x_j \leq k$ (or $\sum_{j=1}^{n} x_j \geq k$)
  - Sequential counters
    - Clauses/Variables: $O(nk)$
  - BDDs
    - Clauses/Variables: $O(nk)$
  - Sorting networks
    - Clauses/Variables: $O(n \log^2 n)$
  - Cardinality Networks:
    - Clauses/Variables: $O(n \log^2 k)$
  - Pairwise Cardinality Networks:
  - ...
Outline

Basic Definitions

DPLL Solvers

CDCL Solvers

What Next in CDCL Solvers?

CNF Encodings
  Boolean Formulas
  Cardinality Constraints
  Pseudo-Boolean Constraints
  Encoding CSPs
Pseudo-Boolean Constraints

- General form: \( \sum_{j=1}^{n} a_j x_j \leq b \)
  - Operational encoding
    - Clauses/Variables: \( O(n) \)
    - Does not guarantee arc-consistency
  - BDDs
    - Worst-case exponential number of clauses
  - Polynomial watchdog encoding
    - Let \( \nu(n) = \log(n) \log(a_{\text{max}}) \)
    - Clauses: \( O(n^3 \nu(n)) \) ; Aux variables: \( O(n^2 \nu(n)) \)
  - Improved polynomial watchdog encoding
    - Clauses & aux variables: \( O(n^3 \log(a_{\text{max}})) \)
  - ...
- Encode $3x_1 + 3x_2 + x_3 \leq 3$
- Construct BDD
  - E.g. analyze variables by decreasing coefficients
- Extract ITE-based circuit from BDD
Encoding PB Constraints with BDDs I

- Encode $3x_1 + 3x_2 + x_3 \leq 3$
- Construct BDD
  - E.g. analyze variables by decreasing coefficients
- Extract ITE-based circuit from BDD
- Encode $3x_1 + 3x_2 + x_3 \leq 3$
- Extract ITE-based circuit from BDD
- Simplify and create final circuit:
More on PB Constraints

- How about $\sum_{j=1}^{n} a_j x_j = k$?
• How about $\sum_{j=1}^{n} a_j x_j = k$?
  - Can use $(\sum_{j=1}^{n} a_j x_j \geq k) \land (\sum_{j=1}^{n} a_j x_j \leq k)$, but...
    - $\sum_{j=1}^{n} a_j x_j = k$ is a subset-sum constraint
      (special case of a knapsack constraint)
More on PB Constraints

- How about $\sum_{j=1}^{n} a_j x_j = k$?
  - Can use $(\sum_{j=1}^{n} a_j x_j \geq k) \land (\sum_{j=1}^{n} a_j x_j \leq k)$, but...
    - $\sum_{j=1}^{n} a_j x_j = k$ is a subset-sum constraint
      (special case of a knapsack constraint)
    - Cannot find all consequences in polynomial time [S03,FS02,T03]
More on PB Constraints

• How about $\sum_{j=1}^{n} a_j x_j = k$?
  – Can use $(\sum_{j=1}^{n} a_j x_j \geq k) \land (\sum_{j=1}^{n} a_j x_j \leq k)$, but...
    ▶ $\sum_{j=1}^{n} a_j x_j = k$ is a subset-sum constraint
      (special case of a knapsack constraint)
    ▶ Cannot find all consequences in polynomial time

[S03, FS02, T03]

• Example:

  $4x_1 + 4x_2 + 3x_3 + 2x_4 = 5$
More on PB Constraints

• How about $\sum_{j=1}^{n} a_j x_j = k$?
  - Can use $(\sum_{j=1}^{n} a_j x_j \geq k) \land (\sum_{j=1}^{n} a_j x_j \leq k)$, but...
    - $\sum_{j=1}^{n} a_j x_j = k$ is a subset-sum constraint
      (special case of a knapsack constraint)
    - Cannot find all consequences in polynomial time

• Example:

  $4x_1 + 4x_2 + 3x_3 + 2x_4 = 5$

  - Replace by $(4x_1 + 4x_2 + 3x_3 + 2x_4 \geq 5) \land (4x_1 + 4x_2 + 3x_3 + 2x_4 \leq 5)$
More on PB Constraints

- How about $\sum_{j=1}^{n} a_j x_j = k$?
  - Can use $(\sum_{j=1}^{n} a_j x_j \geq k) \land (\sum_{j=1}^{n} a_j x_j \leq k)$, but...
    - $\sum_{j=1}^{n} a_j x_j = k$ is a **subset-sum** constraint
      (special case of a **knapsack** constraint)
    - **Cannot** find all consequences in polynomial time

- Example:
  
  \[ 4x_1 + 4x_2 + 3x_3 + 2x_4 = 5 \]

  - Replace by $(4x_1 + 4x_2 + 3x_3 + 2x_4 \geq 5) \land (4x_1 + 4x_2 + 3x_3 + 2x_4 \leq 5)$
  - Let $x_3 = 0$
More on PB Constraints

• How about \( \sum_{j=1}^{n} a_j x_j = k \) ?
  
  – Can use \((\sum_{j=1}^{n} a_j x_j \geq k) \land (\sum_{j=1}^{n} a_j x_j \leq k)\), but...
  
  ▶ \( \sum_{j=1}^{n} a_j x_j = k \) is a subset-sum constraint
    (special case of a knapsack constraint)
  
  ▶ **Cannot** find all consequences in polynomial time

[S03,FS02,T03]

• Example:

\[
4x_1 + 4x_2 + 3x_3 + 2x_4 = 5
\]

– Replace by \((4x_1 + 4x_2 + 3x_3 + 2x_4 \geq 5) \land (4x_1 + 4x_2 + 3x_3 + 2x_4 \leq 5)\)
– Let \(x_3 = 0\)
– Either constraint can still be satisfied, but **not** both
Outline

Basic Definitions

DPLL Solvers

CDCL Solvers

What Next in CDCL Solvers?

CNF Encodings
  Boolean Formulas
  Cardinality Constraints
  Pseudo-Boolean Constraints
  Encoding CSPs
CSP Constraints

• Many possible encodings:

  – Direct encoding \[dK89,GJ96,W00]\n  – Log encoding \[W00]\n  – Support encoding \[K90,G02]\n  – Log-Support encoding \[G07]\n  – Order encoding for finite linear CSPs \[TTKB09]\n  – ...

Direct Encoding for CSP w/ Binary Constraints

- Variable $x_i$ with domain $D_i$, with $m_i = |D_i|

- Represent values of $x_i$ with Boolean variables $x_{i,1}, \ldots, x_{i,m_i}$

- Require $\sum_{k=1}^{m_i} x_{i,k} = 1$
  - Suffices to require $\sum_{k=1}^{m_i} x_{i,k} \geq 1$ [W00]

- If the pair of assignments $x_i = v_i \land x_j = v_j$ is not allowed, add binary clause $(\overline{x_i,v_i} \lor \overline{x_j,v_j})$
Thanks!