CDCL SAT Solvers & SAT-Based Problem Solving

Joao Marques-Silva\textsuperscript{1,2} \& Mikolas Janota\textsuperscript{2}

\textsuperscript{1}University College Dublin, Ireland
\textsuperscript{2}IST/INESC-ID, Lisbon, Portugal

SAT/SMT Summer School 2013
Aalto University, Espoo, Finland
The Success of SAT

- Well-known NP-complete decision problem

[C71]
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

CPU Time (in seconds)
Number of problems solved

- 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)
Overview modern SAT solvers
  – Conflict-Driven Clause Learning (CDCL) SAT solvers
    ▶ Note: Overview for non-experts
This Lecture

- Overview modern SAT solvers
  - **Conflict-Driven Clause Learning (CDCL)** SAT solvers
    - Note: Overview for non-experts

- SAT-based problem solving in practice
  - How to do it?
This Lecture

- Overview modern SAT solvers
  - **Conflict-Driven Clause Learning (CDCL)** SAT solvers
    - *Note:* Overview for non-experts

- SAT-based problem solving in practice
  - How to do it?
    - **Encode** problems to SAT
    - **Embed** SAT solvers in applications
    - Iteratively use a SAT solver (i.e. as an **NP oracle**
Part I

CDCL SAT Solvers
Outline

Basic Definitions

DPLL Solvers

CDCL Solvers

What Next in CDCL Solvers?
Outline

Basic Definitions

- DPLL Solvers
- CDCL Solvers

What Next in CDCL Solvers?
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 $\{0, 1\}$
- **Formula can be** SAT/UNSAT

Example:

$$F \equiv (r) \land (\bar{r} \lor s) \land (\bar{w} \lor a) \land (\bar{x} \lor b) \land (\bar{y} \lor \bar{z} \lor c) \land (\bar{b} \lor \bar{c} \lor d)$$

Example models:

- $\{r, s, a, b, c, d\}$
- $\{r, s, \bar{x}, y, \bar{w}, z, \bar{a}, b, c, d\}$
Preliminaries

- **Variables**: $w, x, y, z, a, b, c, \ldots$
- **Literals**: $w, \neg x, \neg y, a, \ldots$, 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 $\{0, 1\}$
- **Formula can be** SAT/UNSAT
- **Example**:

$$F \triangleq (r) \land (\neg r \lor s) \land (\neg w \lor a) \land (\neg x \lor b) \land (\neg y \lor \neg z \lor c) \land (\neg b \lor \neg c \lor d)$$

- **Example models**:
  - $\{r, s, a, b, c, d\}$
  - $\{r, s, \bar{x}, y, \bar{w}, z, \bar{a}, b, c, d\}$
Resolution

- **Resolution rule:**

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

- Complete proof system for propositional logic
Resolution

- Resolution rule:

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

- Complete proof system for propositional logic

- Extensively used with (CDCL) SAT solvers

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

- **Resolution rule:**
  \[ \frac{(\alpha \lor x) \quad (\beta \lor \bar{x})}{(\alpha \lor \beta)} \]

  - Complete proof system for propositional logic
  \[
  \begin{array}{cccc}
  (x \lor a) & (\bar{x} \lor a) & (\bar{y} \lor \bar{a}) & (y \lor \bar{a}) \\
  \downarrow & \downarrow & \downarrow & \downarrow \\
  (a) & (a) & (\bar{a}) & (\bar{a}) \\
  \downarrow & \downarrow & \downarrow & \downarrow \\
  \bot & \bot & \bot & \bot \\
  \end{array}
  \]

  - 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

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

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

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

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

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

<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>( c \rightarrow d )</td>
</tr>
<tr>
<td>4</td>
<td>( z )</td>
<td>( c \rightarrow d )</td>
</tr>
</tbody>
</table>
Unit Propagation

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

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

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

- Refutation of unsatisfiable formula by iterated resolution operations produces resolution proof.

- An example:
  \[ F = (\neg c) \land (\neg b) \land (\neg a \lor c) \land (a \lor b) \land (a \lor \neg d) \land (\neg a \lor \neg d) \]

- Resolution proof:

  ![Resolution Proof Diagram]

  ![Resolution Proof Diagram](https://via.placeholder.com/150)

  ![Resolution Proof Diagram](https://via.placeholder.com/150)

- 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>
<tr>
<td></td>
<td></td>
<td>\overline{c} \rightarrow \bot</td>
</tr>
</tbody>
</table>

Implication graph with conflict
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>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>\overline{c} \rightarrow \bot</td>
</tr>
</tbody>
</table>

Proof trace \( \bot \): (\overline{a} \lor c) (a \lor b) (\overline{c}) (\overline{b})
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>Ø</td>
<td>( \bar{b} \rightarrow a )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \bar{c} \rightarrow \bot )</td>
</tr>
</tbody>
</table>

Resolution proof follows structure of conflicts
Unsatisfiable Cores & Proof Traces

- CNF formula:

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

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?
The DPLL Algorithm

- Unassigned variables?
  - Yes: Assign value to variable
  - No: Unit propagation

- Conflict?
  - Yes: Can undo decision?
    - Yes: Backtrack & flip variable
    - No: Unassignable
  - No: Satisfiable

- Optional: pure literal rule
The DPLL Algorithm

$$\mathcal{F} = (x \lor y) \land (a \lor b) \land (\bar{a} \lor b) \land (a \lor \bar{b}) \land (\bar{a} \lor \bar{b})$$

- Optional: pure literal rule
The DPLL Algorithm

Optional: pure literal rule
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>(x)</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>(y)</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>(\overline{a})</td>
<td>(\overline{b})</td>
</tr>
</tbody>
</table>
The DPLL Algorithm

- Optional: pure literal rule

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

<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>(x)</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>(\overline{y})</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>(a)</td>
<td>(b) (\perp)</td>
</tr>
</tbody>
</table>

- Unassigned variables? 
  - Y: Assign value to variable
  - N: Unit propagation

- Conflict? 
  - Y: Satisfiable

- Can undo decision? 
  - Y: Backtrack & flip variable
  - N: Unsatisfiable
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>∅</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>(\bar{a})</td>
<td>(\bar{b})</td>
</tr>
</tbody>
</table>
The DPLL Algorithm

- Optional: pure literal rule

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

<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>a</td>
<td>b</td>
</tr>
</tbody>
</table>

Level Dec. Unit Prop.
The DPLL Algorithm

- Optional: pure literal rule

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

Basic Definitions

DPLL Solvers

CDCL Solvers

What Next in CDCL Solvers?
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
  - ...

[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)

CPU Time (in seconds)

Number of problems solved
Outline

Basic Definitions

DPLL Solvers

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

What Next in CDCL Solvers?
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: \( \neg x \lor \neg z \)
- 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
### 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) (\perp)</td>
</tr>
</tbody>
</table>

- **Analyze conflict**
  - Reasons: \(x\) and \(z\)
    - Decision variable & literals assigned at lower decision levels
Clause Learning

- 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>
<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: $(\bar{x} \lor \bar{z})$
- Can relate clause learning with resolution

\[
(\bar{a} \lor \bar{b}) (\bar{z} \lor b) (\bar{x} \lor \bar{z} \lor a)
\]
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: \((\overline{x} \lor \overline{z})\)

• 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: \((\overline{x} \lor \overline{z})\)

- 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>$\emptyset$</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>$z$</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: $(\bar{x} \lor \bar{z})$

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

Level | Dec. | Unit Prop.
--- | --- | ---
0    | ∅   | 0
1    | x   | 1
2    | y   | 2
3    | z   | 3
      | a   | 4
      | b   | 5

Clause \((\neg x \lor \neg z)\) 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 Bracktracking

<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>$z$</td>
<td></td>
</tr>
</tbody>
</table>

- Clause $(\bar{x} \lor \bar{z})$ is asserting at decision level 1
Clause Learning – After Backtracking

• Clause \((\overline{x} \lor \overline{z})\) is asserting at decision level 1
Clause Learning – After Bracktracking

<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>

<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>z</td>
</tr>
</tbody>
</table>

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

[MSS96,MSS99] [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>∅</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>

Graph:
- $w$ connects to $x$.
- $y$ connects to $a$.
- $z$ connects to $a$.
- $a$ connects to $c$.
- $b$ connects to $c$.
- $c$ connects to $\bot$. 
### 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 \rightarrow a \rightarrow c )</td>
<td>( z \rightarrow b \rightarrow \bot )</td>
</tr>
</tbody>
</table>

- Learn clause \((\bar{w} \lor \bar{x} \lor \bar{y} \lor \bar{z})\)

\[\begin{align*}
(\bar{b} \lor c) & \quad (\bar{w} \lor c) & \quad (\bar{x} \lor \bar{a} \lor b) & \quad (\bar{y} \lor \bar{z} \lor a) \\
(\bar{w} \lor \bar{b}) & & \quad (\bar{w} \lor \bar{x} \lor \bar{a}) & \quad (\bar{w} \lor \bar{x} \lor \bar{y} \lor \bar{z}) \\
(\bar{w} \lor \bar{x} \lor \bar{a}) & & & \\
(\bar{w} \lor \bar{x} \lor \bar{y} \lor \bar{z}) & & &
\end{align*}\]
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>$(\bar{w} \lor c)$</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>
</tbody>
</table>

- Learn clause $(\bar{w} \lor \bar{x} \lor \bar{y} \lor \bar{z})$
- But $a$ is an UIP
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>Ø</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**
- **Learn clause** \((\overline{w} \lor \overline{x} \lor \overline{a})\)
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></td>
</tr>
</tbody>
</table>

First UIP:
- Learn clause \( \overline{w} \lor \overline{y} \lor \overline{a} \)

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 \((\bar{w} \lor \bar{y} \lor \bar{a})\)

- 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

<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>$r$</td>
</tr>
</tbody>
</table>

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

- But there can be more than 1 UIP

- 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>$\emptyset$</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>$r$ $s$ $a$ $c$ $b$ ⊥</td>
</tr>
</tbody>
</table>

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

<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, a, c, s</td>
</tr>
</tbody>
</table>

- **First UIP:**
  - Learn clause \((\bar{w} \lor \bar{y} \lor \bar{a})\)
- **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 \((\overline{w} \lor \overline{y} \lor \overline{a})\)
- 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
  - First UIP learning proposed in Chaff
- Not used in recent state of the art CDCL SAT solvers

---

<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>s</td>
<td>b \rightarrow \bot</td>
</tr>
</tbody>
</table>
Multiple UIPs

- **First UIP:**
  - Learn clause \((\bar{w} \lor \bar{y} \lor \bar{a})\)
- 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></td>
</tr>
<tr>
<td>3</td>
<td>$z$</td>
<td>$c \rightarrow \bot$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$a$</td>
</tr>
</tbody>
</table>
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 \((\bar{x} \lor \bar{y} \lor \bar{z} \lor \bar{b})\)

\[(\bar{a} \lor \bar{c}) \quad (\bar{z} \lor \bar{b} \lor c) \quad (\bar{x} \lor \bar{y} \lor \bar{z} \lor a)\]

\[(\bar{z} \lor \bar{b} \lor \bar{a}) \quad (\bar{x} \lor \bar{y} \lor \bar{z} \lor \bar{b})\]

\[
\begin{align*}
\text{(1)} & \quad \text{(2)} \\
\text{(2)} & \quad \text{(3)} \\
\text{(3)} & \quad \text{(4)} \\
\end{align*}
\]
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 $(\overline{x} \lor \overline{y} \lor \overline{z} \lor \overline{b})$
- 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>
</tbody>
</table>

- Learn clause $(\bar{x} \lor \bar{y} \lor \bar{z} \lor \bar{b})$
- 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 \((\bar{x} \lor \bar{y} \lor \bar{z})\)
### 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>$a$</td>
</tr>
<tr>
<td>2</td>
<td>$x$</td>
<td>$e$</td>
</tr>
</tbody>
</table>

**Diagram:**

- **Level 0:** $\emptyset$
- **Level 1:**
  - $w \rightarrow a \rightarrow c$
  - $b$
- **Level 2:**
  - $x \rightarrow e$
  - $d \rightarrow \bot$

**Notes:**
- 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 nodes. Learn clause if only marked nodes visited.

[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></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>⊥</td>
</tr>
</tbody>
</table>

- Learn clause \((\overline{w} \lor \overline{x} \lor \overline{c})\)
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>$a$</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>$\perp$</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Level</th>
<th>Dec.</th>
<th>Unit Prop.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>∅</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>w</td>
<td>a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>b</td>
</tr>
<tr>
<td>2</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>e</td>
</tr>
<tr>
<td></td>
<td></td>
<td>d</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
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]
• **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

• **Trace back from** \(c\) until marked nodes or new nodes
  – Learn clause if only marked nodes visited

---

[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>$\emptyset$</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 $(\overline{w} \lor \overline{x} \lor \overline{a} \lor \overline{b})$
- **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** nodes
  - Learn clause if only **marked** nodes visited
Outline

Basic Definitions

DPLL Solvers

CDCL Solvers

Clause Learning, UIPs & Minimization

Search Restarts & Lazy Data Structures

What Next in CDCL Solvers?
Search Restarts I

- Heavy-tail behavior:

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

[GSK98]
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)
• 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)
• 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 cutoff after each restart
  - Guarantees completeness
  - Different policies exist (see refs)
- Works for SAT & 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
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: \( O(n) \)

Worst-case number of literals: \( O(mn) \)
In practice, Unit propagation slowdown 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 should access clauses containing it
  - Why? Unit propagation

Clause with literals results in 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!

- 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!

- Clause learning to be effective requires a more efficient representation:
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!

• Clause learning to be effective requires a more efficient representation: Watched Literals
• 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: 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 states of a clause]

- unresolved
- 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

[MMZZM01]
Watched Literals

• Important states of a clause
• Associate 2 references with each clause
• Deciding unit requires traversing all literals
• References **unchanged** when backtracking
• **Lightweight branching**
  - Use conflict to bias variables to branch on, associate score with each variable
  - Prefer recent bias by regularly decreasing variable scores

[MMZZM01]

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

[MSS96, GN02, ES03]

• **Proven recent techniques**
  - Phase saving
    - [PD07]
  - 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 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 less used clauses

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

Basic Definitions

DPLL Solvers

CDCL Solvers

What Next in CDCL Solvers?
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

• **Application-driven improvements**
  - Incremental SAT
    - Handling of assumptions due to MUS extractors
Part II

SAT-Based Problem Solving
How to Solve Problems with SAT?

- **CNF encodings**
  - Represent problem as instance of SAT
  - E.g. Eager SMT, Pseudo-Boolean constraints, etc.

- **Embedding of SAT solvers**
  - SAT solver used to implement domain specific algorithm
  - White-box integration
  - E.g. Lazy SMT, Pseudo-Boolean constraints/optimization, etc.

- **SAT solvers as oracles**
  - Algorithm invokes SAT solver as an NP oracle
  - Black-box integration
  - E.g. MaxSAT, MUSes, (2)QBF, etc.

**Note:**
- CNF encodings most often used with either black-box or white-box approaches
- SAT techniques adapted in many other domains: QBF, ASP, ILP, CSP, ...
How to Solve Problems with SAT?

• **CNF encodings**
  - Represent problem as instance of SAT
  - E.g. Eager SMT, Pseudo-Boolean constraints, etc.

• **Embedding of SAT solvers**
  - SAT solver used to implement domain specific algorithm
  - White-box integration
  - E.g. Lazy SMT, Pseudo-Boolean constraints/optimization, etc.

• **Note**: CNF encodings most often used with either black-box or white-box approaches
  SAT techniques adapted in many other domains: QBF, ASP, ILP, CSP, ...
How to Solve Problems with SAT?

- **CNF encodings**
  - Represent problem as instance of SAT
  - E.g. Eager SMT, Pseudo-Boolean constraints, etc.

- **Embedding of SAT solvers**
  - SAT solver used to implement domain specific algorithm
  - **White-box** integration
  - E.g. Lazy SMT, Pseudo-Boolean constraints/optimization, etc.

- **SAT solvers as oracles**
  - Algorithm invokes SAT solver as an NP oracle
  - **Black-box** integration
  - E.g. MaxSAT, MUSes, (2)QBF, etc.
How to Solve Problems with SAT?

- **CNF encodings**
  - Represent problem as instance of SAT
  - E.g. Eager SMT, Pseudo-Boolean constraints, etc.

- **Embedding of SAT solvers**
  - SAT solver used to implement domain specific algorithm
  - White-box integration
  - E.g. Lazy SMT, Pseudo-Boolean constraints/optimization, etc.

- **SAT solvers as oracles**
  - Algorithm invokes SAT solver as an NP oracle
  - Black-box integration
  - E.g. MaxSAT, MUSes, (2)QBF, etc.

- **Note:**
  - CNF encodings most often used with either black-box or white-box approaches
  - SAT techniques adapted in many other domains: QBF, ASP, ILP, CSP, ...
Some apps associated with more than one concept: planning, BMC, lazy clause generation, etc.
Examples of SAT-Based Problem Solving I

- **Function problems in** $\text{FP}^{\text{NP}}[\log n]$
  - Unweighted Maximum Satisfiability ($\text{MaxSAT}$)
  - Minimal Correction Subsets ($\text{MCSes}$)
  - Minimal models
  - ...

- **Function problems in** $\text{FP}^{\text{NP}}$
  - Weighted Maximum Satisfiability ($\text{MaxSAT}$)
  - Minimal Unsatisfiable Subformulas ($\text{MUSes}$)
  - Minimal Equivalent Subformulas ($\text{MESes}$)
  - Prime implicates
  - ...

- **Enumeration problems**
  - Models
  - MUSes
  - MCSes
  - MaxSAT
  - ...

Examples of SAT-Based Problem Solving II

• Decision problems in $\Sigma_2^P$
  – 2QBF
  – ...

• Function problems in $FP_2^\Sigma^P$
  – (Weighted) Quantified MaxSAT ($Q\text{MaxSAT}$) [IJMS13]
  – Smallest MUS ($SMUS$) [IJMS13]
  – ...

• Decision problems in $\text{PSPACE}$
  – QBF
  – ...

• ...

Outline

CNF Encodings

SAT Embeddings

SAT Oracles

What Next in SAT-Based Problem Solving?
Outline

CNF Encodings

SAT Embeddings

SAT Oracles

What Next in SAT-Based Problem Solving?
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

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

**SAT Embeddings**

**SAT Oracles**

**What Next in SAT-Based Problem Solving?**
Representing Boolean Formulas / Circuits I

- 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 $z \iff \text{OP}(x, y)$
    - CNF formula for the circuit is the conjunction of CNF formula for each gate

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

$$F_t = (\bar{r} \lor t) \land (\bar{s} \lor t) \land (r \lor s \lor \bar{t})$$
Representing Boolean Formulas / Circuits II

\[ F_c = (a \lor c) \land (b \lor c) \land (\overline{\bar{a}} \lor \overline{\bar{b}} \lor \overline{\bar{c}}) \]

<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

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

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

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

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

**SAT Embeddings**

**SAT Oracles**

What Next in SAT-Based Problem Solving?
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: \( \sum_{j=1}^{n} x_j \preceq k \), with \( \preceq \in \{<, \leq, =, \geq, >\} \)

• Solution #1:
  - Use PB solver
  - Difficult to keep up with advances in SAT technology
  - For SAT/UNSAT, best solvers already encode to CNF
    - E.g. Minisat+, but also QMaxSat, MSUnCore, (W)PM2
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: $\sum_{j=1}^{n} x_j \triangledown k$, with $\triangledown \in \{<, \leq, =, \geq, >\}$

- **Solution #1:**
  - Use PB solver
  - Difficult to keep up with advances in SAT technology
  - For SAT/UNSAT, best solvers already encode to CNF
    - E.g. Minisat+, but also QMaxSat, MSUnCore, (W)PM2

- **Solution #2:**
  - Encode cardinality constraints to CNF
  - Use SAT solver
Equals1, AtLeast1 & AtMost1 Constraints

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

- \( \sum_{j=1}^{n} x_j \geq 1 \): 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: \(O(n)\)  
  - Bitwise encoding
    - Clauses: \(O(n \log n)\) ; Auxiliary variables: \(O(\log n)\)
  - ...

[S05] [P07,FP01]
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 $v_0 \ldots v_{j-1} = b_0 \ldots b_{j-1}$, the binary encoding of $j-1$ $x_j \rightarrow (v_0 = b_0) \land \ldots \land (v_{j-1} = b_{j-1}) \iff (\overline{x_j} \lor (v_i = b_i)) = (\overline{x_j} \lor l_i), i = 0, \ldots, r-1$, where $l_i \equiv v_i$, if $b_i = 1$, $\overline{l_i} \equiv \overline{v_i}$, 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

  – $O(n \log n)$ clauses; $O(\log n)$ auxiliary variables

• 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 \( v_0 \ldots v_{j-1} = b_0 \ldots b_{j-1} \), the binary encoding of \( j - 1 \)
    \( x_j \rightarrow (v_0 = b_0) \land \ldots \land (v_{j-1} = b_{j-1}) \iff (\bar{x}_j \lor (v_0 = b_0) \land \ldots \land (v_{j-1} = b_{j-1})) \)

- 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</td>
</tr>
<tr>
<td>( x_2 )</td>
<td>1</td>
</tr>
<tr>
<td>( x_3 )</td>
<td>2</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 $v_0 \ldots v_{j-1} = b_0 \ldots b_{j-1}$, the binary encoding of $j - 1$
    $x_j \rightarrow (v_0 = b_0) \land \ldots \land (v_{j-1} = b_{j-1}) \iff (\overline{x_j} \lor (v_0 = b_0) \land \ldots \land (v_{j-1} = b_{j-1}))$
  - 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>$j - 1$</th>
<th>$v_1 v_0$</th>
<th>$\overline{x_1} \lor \overline{v_1} \lor (\overline{x_1} \lor \overline{v_0})$</th>
<th>$\overline{x_2} \lor \overline{v_1} \lor (\overline{x_2} \lor v_0)$</th>
<th>$\overline{x_3} \lor v_1 \lor (\overline{x_3} \lor \overline{v_0})$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_1$</td>
<td>0</td>
<td>00</td>
<td>(\overline{x_1} \lor \overline{v_1}) \land (\overline{x_1} \lor \overline{v_0})</td>
<td>(\overline{x_1} \lor \overline{v_1}) \land (\overline{x_1} \lor v_0)</td>
</tr>
<tr>
<td>$x_2$</td>
<td>1</td>
<td>01</td>
<td>(\overline{x_2} \lor \overline{v_1}) \land (\overline{x_2} \lor v_0)</td>
<td>(\overline{x_2} \lor \overline{v_1}) \land (\overline{x_2} \lor v_0)</td>
</tr>
<tr>
<td>$x_3$</td>
<td>2</td>
<td>10</td>
<td>(\overline{x_3} \lor v_1) \land (\overline{x_3} \lor \overline{v_0})</td>
<td>(\overline{x_3} \lor v_1) \land (\overline{x_3} \lor \overline{v_0})</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 $v_0 \ldots v_{j-1} = b_0 \ldots b_{j-1}$, the binary encoding of $j - 1$
  
  $$x_j \rightarrow (v_0 = b_0) \land \ldots \land (v_{j-1} = b_{j-1}) \iff (\bar{x}_j \lor (v_0 = b_0) \land \ldots \land (v_{j-1} = b_{j-1}))$$
  - 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
  - $O(n \log n)$ clauses ; $O(\log n)$ auxiliary variables

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

<table>
<thead>
<tr>
<th>$j - 1$</th>
<th>$v_1 v_0$</th>
<th>$(\bar{x}_1 \lor \bar{v}_1) \land (\bar{x}_1 \lor \bar{v}_0)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_1$</td>
<td>0 00</td>
<td>$(\bar{x}_2 \lor \bar{v}_1) \land (\bar{x}_2 \lor v_0)$</td>
</tr>
<tr>
<td>$x_2$</td>
<td>1 01</td>
<td>$(\bar{x}_3 \lor v_1) \land (\bar{x}_3 \lor \bar{v}_0)$</td>
</tr>
<tr>
<td>$x_3$</td>
<td>2 10</td>
<td></td>
</tr>
</tbody>
</table>
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

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

SAT Embeddings

SAT Oracles

What Next in SAT-Based Problem Solving?
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 $\nu(n) = \log(n) \log(a_{max})$
    - Clauses: $O(n^3 \nu(n))$; Aux variables: $O(n^2 \nu(n))$
  - Improved polynomial watchdog encoding
    - Clauses & aux variables: $O(n^3 \log(a_{max}))$
  - ...

[W98] [ES06] [BBR09] [ANORC11b]
- 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 $3x_1 + 3x_2 + x_3 \leq 3$
- 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:

  \[ 4x_1 + 3x_2 + 2x_3 = 5 \]
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 [S03,FS02,T03]

- Example:

  \[
  4x_1 + 3x_2 + 2x_3 = 5
  \]
  - Replace by \((4x_1 + 3x_2 + 2x_3 \geq 5) \land (4x_1 + 3x_2 + 2x_3 \leq 5)\)
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:
  
  \[ 4x_1 + 3x_2 + 2x_3 = 5 \]
  - Replace by $(4x_1 + 3x_2 + 2x_3 \geq 5) \land (4x_1 + 3x_2 + 2x_3 \leq 5)$
  - Let $x_2 = 0$
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:

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

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

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

SAT Embeddings

SAT Oracles

What Next in SAT-Based Problem Solving?
CSP Constraints

- Many possible encodings:
  - Direct encoding \[dK89,GJ96,W00\]
  - Log encoding \[W00\]
  - Support encoding \[K90,G02\]
  - Log-Support encoding \[G07\]
  - Order encoding for finite linear CSPs \[TTKB09\]
• 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 $\left( \overline{x}_{i,v_i} \lor \overline{x}_{j,v_j} \right)$
Outline

CNF Encodings

SAT Embeddings

SAT Oracles

What Next in SAT-Based Problem Solving?
Embedding SAT Solvers

- Modify SAT solver to interface problem-specific propagators (or theory solvers)
- Typical interface:
  - SAT solvers communicates assignments/constraints to propagators
  - Retrieve resulting assignments or explanations for inconsistency
- Well-known examples (many more):
  - Branch&bound PB optimization
  - Non-clausal SAT solvers
  - Lazy SMT solving (see later talks)
- Key problem:
  - Keeping up with improvements in SAT solvers
Pseudo-Boolean Constraints & Optimization

- Pseudo-Boolean Constraints:
  - Boolean variables: $x_1, \ldots, x_n$
  - Linear inequalities:
    \[
    \sum_{j \in \mathbb{N}} a_{ij} l_j \geq b_i, \quad l_j \in \{x_j, \bar{x}_j\}, \quad x_j \in \{0, 1\}, \quad a_{ij}, b_i \in \mathbb{N}_0^+
    \]
Pseudo-Boolean Constraints & Optimization

- **Pseudo-Boolean Constraints:**
  - Boolean variables: \( x_1, \ldots, x_n \)
  - Linear inequalities:
    \[
    \sum_{j \in N} a_{ij} l_j \geq b_i, \quad l_j \in \{x_j, \bar{x}_j\}, \; x_j \in \{0, 1\}, \; a_{ij}, b_i \in \mathbb{N}_0^+
    \]

- **Pseudo-Boolean Optimization (PBO):**
  \[
  \text{minimize} \quad \sum_{j \in N} c_j \cdot x_j \\
  \text{subject to} \quad \sum_{j \in N} a_{ij} l_j \geq b_i, \\
  \quad l_j \in \{x_j, \bar{x}_j\}, \; x_j \in \{0, 1\}, \; a_{ij}, b_i, c_j \in \mathbb{N}_0^+
  \]
Pseudo-Boolean Constraints & Optimization

• Pseudo-Boolean Constraints:
  – Boolean variables: \( x_1, \ldots, x_n \)
  – Linear inequalities:
    \[ \sum_{j \in N} a_{ij} l_j \geq b_i, \quad l_j \in \{x_j, \bar{x}_j\}, \quad x_j \in \{0, 1\}, \quad a_{ij}, b_i \in \mathbb{N}_0^+ \]

• Pseudo-Boolean Optimization (PBO):
  
  minimize \( \sum_{j \in N} c_j \cdot x_j \) 

  subject to \( \sum_{j \in N} a_{ij} l_j \geq b_i, \) 

  \( l_j \in \{x_j, \bar{x}_j\}, \quad x_j \in \{0, 1\}, \quad a_{ij}, b_i, c_j \in \mathbb{N}_0^+ \)

• Branch and bound (B&B) PBO algorithm:
  – Extend SAT solver
  – Must develop propagator for PB constraints
  – B&B search for computing optimum cost function value
    ▶ Trivial upper bound: all \( x_j = 1 \)
Limitations with Embeddings

- **B&B MaxSAT solving:**
  - Cannot use unit propagation
  - Cannot learn clauses

- **MUS extraction:**
  - Decision of clauses to include in MUS based on unsatisfiable outcomes
  - No immediate gain from embedding SAT solvers
Outline

CNF Encodings

SAT Embeddings

SAT Oracles

What Next in SAT-Based Problem Solving?
Practical Aspects of Using SAT Oracles

- **Incremental vs. non-incremental SAT**

  - **Incremental SAT**:
    - Replace each clause ($C_i$) with ($C_i \lor \neg a_i$), where $a_i$ is an assumption variable.
    - When calling SAT solver, each assumption can be assigned 1, 0, or left unassigned.
    - $a_i = 1$ to activate clause $C_i$.
    - $a_i = 0$ to deactivate clause $C_i$.
    - Add clause ($\neg a_i$) to delete $C_i$.
    - Note: incremental SAT enables clause reuse.

  - **Non-incremental SAT**:
    - Submit complete formula to SAT solver in each iteration.
    - Note: difficult to instrument clause reuse.

- What does the SAT oracle compute/return?

  1. Yes/No: $(st) \leftarrow \text{SAT}(F)$
  2. Compute model: $(st, \mu) \leftarrow \text{SAT}(F)$
  3. Compute unsatisfiable cores: $(st, \mu, U) \leftarrow \text{SAT}(F)$
  4. Compute proof traces/resolution proof: $(st, \mu, T) \leftarrow \text{SAT}(F)$
Practical Aspects of Using SAT Oracles

• Incremental vs. non-incremental SAT
  
  – Incremental SAT:
    
    ▶ Replace each clause ($C_i$) with ($C_i \lor \overline{a_i}$), where $a_i$ is assumption variable
    
    ▶ When calling SAT solver, each assumption can be assigned 1, 0, or be left unassigned

  
  – Non-incremental SAT:
Practical Aspects of Using SAT Oracles

- **Incremental vs. non-incremental SAT**
  - **Incremental SAT:**
    - Replace each clause \( C_i \) with \( C_i \lor \bar{a}_i \), where \( a_i \) is an assumption variable
    - When calling SAT solver, each assumption can be assigned 1, 0, or be left unassigned
    - \( a_i = 1 \) to activate clause \( C_i \)
    - \( a_i = 0 \) to deactivate clause \( C_i \)
  - **Non-incremental SAT:**

- What does the SAT oracle compute/return?
  1. Yes/No: \( st \leftarrow \text{SAT}(F) \)
  2. Compute model: \( (st, \mu) \leftarrow \text{SAT}(F) \)
  3. Compute unsatisfiable cores: \( (st, \mu, U) \leftarrow \text{SAT}(F) \)
  4. Compute proof traces/resolution proof: \( (st, \mu, T) \leftarrow \text{SAT}(F) \)
Practical Aspects of Using SAT Oracles

- **Incremental vs. non-incremental SAT**
  - Incremental SAT:
    - Replace each clause $(C_i)$ with $(C_i \lor \overline{a_i})$, where $a_i$ is assumption variable.
    - When calling SAT solver, each assumption can be assigned 1, 0, or be left unassigned.
    - $a_i = 1$ to activate clause $C_i$.
    - $a_i = 0$ to deactivate clause $C_i$.
    - Add clause $(\overline{a_i})$ to delete $C_i$.
  - Non-incremental SAT:

---

[ES03]
Practical Aspects of Using SAT Oracles

- **Incremental vs. non-incremental SAT**
  - Incremental SAT:
    - Replace each clause \((C_i)\) with \((C_i \lor \bar{a}_i)\), where \(a_i\) is assumption variable
    - When calling SAT solver, each assumption can be assigned 1, 0, or be left unassigned
    - \(a_i = 1\) to activate clause \(C_i\)
    - \(a_i = 0\) to deactivate clause \(C_i\)
    - Add clause \(\bar{a}_i\) to delete \(C_i\)
    - **Note**: incremental SAT enables clause reuse
  - Non-incremental SAT:

- [ES03]
Practical Aspects of Using SAT Oracles

- **Incremental vs. non-incremental SAT**
  - **Incremental SAT:**
    - Replace each clause \((C_i)\) with \((C_i \lor \overline{a_i})\), where \(a_i\) is assumption variable
    - When calling SAT solver, each assumption can be assigned 1, 0, or be left unassigned
    - \(a_i = 1\) to activate clause \(C_i\)
    - \(a_i = 0\) to deactivate clause \(C_i\)
    - Add clause \((\overline{a_i})\) to delete \(C_i\)
    - **Note:** incremental SAT enables clause reuse
  - **Non-incremental SAT:**
    - Submit complete formula to SAT solver in each iteration
    - **Note:** difficult to instrument clause reuse

What does the SAT oracle compute/return?

1. Yes/No: \((\text{st}) \leftarrow \text{SAT}(F)\)
2. Compute model: \((\text{st}, \mu) \leftarrow \text{SAT}(F)\)
3. Compute unsatisfiable cores: \((\text{st}, \mu, \text{U}) \leftarrow \text{SAT}(F)\)
4. Compute proof traces/resolution proof: \((\text{st}, \mu, \text{T}) \leftarrow \text{SAT}(F)\)
Practical Aspects of Using SAT Oracles

- **Incremental vs. non-incremental SAT**
  - **Incremental SAT:**
    - Replace each clause ($C_i$) with ($C_i \lor \bar{a}_i$), where $a_i$ is assumption variable
    - When calling SAT solver, each assumption can be assigned 1, 0, or be left unassigned
    - $a_i = 1$ to activate clause $C_i$
    - $a_i = 0$ to deactivate clause $C_i$
    - Add clause ($\bar{a}_i$) to delete $C_i$
    - **Note:** incremental SAT enables clause reuse
  - **Non-incremental SAT:**
    - Submit complete formula to SAT solver in each iteration
    - **Note:** difficult to instrument clause reuse

- **What does the SAT oracle compute/return?**
  1. Yes/No: ($st$) $\leftarrow$ SAT($\mathcal{F}$)
Practical Aspects of Using SAT Oracles

- **Incremental vs. non-incremental SAT**
  - Incremental SAT:
    - Replace each clause \((C_i)\) with \((C_i \lor \overline{a_i})\), where \(a_i\) is assumption variable
    - When calling SAT solver, each assumption can be assigned 1, 0, or be left unassigned
    - \(a_i = 1\) to activate clause \(C_i\)
    - \(a_i = 0\) to deactivate clause \(C_i\)
    - Add clause \((\overline{a_i})\) to delete \(C_i\)
    - **Note**: incremental SAT enables clause reuse
  - Non-incremental SAT:
    - Submit complete formula to SAT solver in each iteration
    - **Note**: difficult to instrument clause reuse

- **What does the SAT oracle compute/return?**
  1. Yes/No: \((st) \leftarrow SAT(F)\)
  2. Compute model: \((st, \mu) \leftarrow SAT(F)\)
Practical Aspects of Using SAT Oracles

• **Incremental vs. non-incremental SAT**

  - **Incremental SAT:**
    - Replace each clause \( (C_i) \) with \( (C_i \lor \bar{a}_i) \), where \( a_i \) is assumption variable
    - When calling SAT solver, each assumption can be assigned 1, 0, or be left unassigned
    - \( a_i = 1 \) to activate clause \( C_i \)
    - \( a_i = 0 \) to deactivate clause \( C_i \)
    - Add clause \( (\bar{a}_i) \) to delete \( C_i \)
    - **Note:** incremental SAT enables clause reuse

  - **Non-incremental SAT:**
    - Submit complete formula to SAT solver in each iteration
    - **Note:** difficult to instrument clause reuse

• What does the SAT oracle compute/return?

  1. **Yes/No:** \((st) \leftarrow \text{SAT}(\mathcal{F})\)
  2. Compute **model:** \((st, \mu) \leftarrow \text{SAT}(\mathcal{F})\)
  3. Compute **unsatisfiable cores:** \((st, \mu, \mathcal{U}) \leftarrow \text{SAT}(\mathcal{F})\)
Practical Aspects of Using SAT Oracles

• **Incremental vs. non-incremental SAT**

  - **Incremental SAT**:
    - Replace each clause \((C_i)\) with \((C_i \lor \overline{a_i})\), where \(a_i\) is an assumption variable
    - When calling SAT solver, each assumption can be assigned 1, 0, or left unassigned
    - \(a_i = 1\) to activate clause \(C_i\)
    - \(a_i = 0\) to deactivate clause \(C_i\)
    - Add clause \((\overline{a_i})\) to delete \(C_i\)
    - **Note**: incremental SAT enables clause reuse

  - **Non-incremental SAT**:
    - Submit complete formula to SAT solver in each iteration
    - **Note**: difficult to instrument clause reuse

• What does the SAT oracle compute/return?

  1. Yes/No: \((st) \leftarrow SAT(F)\)
  2. Compute model: \((st, \mu) \leftarrow SAT(F)\)
  3. Compute unsatisfiable cores: \((st, \mu, U) \leftarrow SAT(F)\)
  4. Compute proof traces/resolution proof: \((st, \mu, T) \leftarrow SAT(F)\)
Outline

CNF Encodings

SAT Embeddings

SAT Oracles
  MUS Extraction
  MaxSAT
  2QBF

What Next in SAT-Based Problem Solving?
Defining MUSes

- Formula is **unsatisfiable** but not irreducible
Defining MUSes

- Formula is *unsatisfiable* but not irreducible
- Can remove clauses, and formula still *unsatisfiable*
Defining MUSes

- Formula is **unsatisfiable** but not irreducible
- Can remove clauses, and formula still unsatisfiable
- A **Minimal Unsatisfiable Subformula (MUS)** is an unsatisfiable and irreducible subformula
Defining MUSes

- Formula is unsatisfiable but not irreducible
- Can remove clauses, and formula still unsatisfiable
- A Minimal Unsatisfiable Subformula (MUS) is an unsatisfiable and irreducible subformula
**Defining MUSes**

- Formula is **unsatisfiable** but not irreducible
- Can remove clauses, and formula still **unsatisfiable**
- A **Minimal Unsatisfiable Subformula (MUS)** is an **unsatisfiable** and **irreducible** subformula

<table>
<thead>
<tr>
<th>Clause 1</th>
<th>Clause 2</th>
<th>Clause 3</th>
<th>Clause 4</th>
<th>Clause 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_6 \lor x_2$</td>
<td>$\neg x_6 \lor x_2$</td>
<td>$\neg x_2 \lor x_1$</td>
<td>$\neg x_1$</td>
<td>$x_2 \lor x_4$</td>
</tr>
<tr>
<td>$\neg x_6 \lor x_8$</td>
<td>$x_6 \lor \neg x_8$</td>
<td>$x_4 \lor x_5$</td>
<td>$\neg x_4 \lor x_5$</td>
<td>$\neg x_3 \lor x_3$</td>
</tr>
</tbody>
</table>
Defining MUSes

- Formula is **unsatisfiable** but not irreducible
- Can remove clauses, and formula still **unsatisfiable**
- A **Minimal Unsatisfiable Subformula (MUS)** is an **unsatisfiable** and **irreducible** subformula
- How to compute an MUS?
Deletion-Based MUS Extraction

Input: Unsatisfiable CNF Formula $\mathcal{F}$
Output: MUS $\mathcal{M}$

begin
\[
\mathcal{M} \leftarrow \mathcal{F} \quad \text{// MUS over-approximation}
\]
foreach $c \in \mathcal{M}$ do
\[
\text{if not SAT}(\mathcal{M} \setminus \{c\}) \text{ then}
\]
\[
\quad \mathcal{M} \leftarrow \mathcal{M} \setminus \{c\} \quad \text{// If UNSAT}(\mathcal{M} \setminus \{c\}), \text{ then } c \not\in \mathcal{M}
\]
return $\mathcal{M}$ \quad \text{// Final $\mathcal{M}$ is MUS}
end

- Number of calls to SAT solver: $\mathcal{O}(|\mathcal{F}|)$
Deletion-Based MUS Extraction

**Input**: Unsatisfiable CNF Formula $\mathcal{F}$

**Output**: MUS $\mathcal{M}$

begin

\[ \mathcal{M} \leftarrow \mathcal{F} \]  // MUS over-approximation

foreach $c \in \mathcal{M}$ do

if not SAT($\mathcal{M} \setminus \{c\}$) then

\[ \mathcal{M} \leftarrow \mathcal{M} \setminus \{c\} \]  // Remove $c$ from $\mathcal{M}$

return $\mathcal{M}$

end

- Number of calls to SAT solver: $O(|\mathcal{F}|)$
An Example

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

UNSAT instance
An Example

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

Hide clause \((\neg x_1 \lor x_2)\)
An Example

SAT instance $\rightarrow$ keep clause $(\neg x_1 \lor x_2)$
An Example

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

Hide clause \((\neg x_3 \lor x_2)\)
An Example

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

**UNSAT** instance → **remove** clause \((\neg x_3 \lor x_2)\)
An Example

Hide clause $(x_1 \lor x_2)$
An Example

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

SAT instance → keep clause \((x_1 \lor x_2)\)
An Example

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

Hide clause $(\neg x_3)$
An Example

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

**UNSAT** instance \(\rightarrow\) **remove** clause \((\neg x_3)\)
An Example

Hide clause \((\neg x_2)\)
An Example

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

SAT instance $\rightarrow$ keep clause ($\neg x_2$)
An Example

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

Computed MUS
# More on MUS Extraction

<table>
<thead>
<tr>
<th>Algorithm</th>
<th># Oracle Calls</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertion (Default)</td>
<td>$O(m \times k)$</td>
<td>[SP88]</td>
</tr>
<tr>
<td>Deletion (Default)</td>
<td>$O(m)$</td>
<td>[CD91,BDTW93]</td>
</tr>
<tr>
<td>QuickXplain</td>
<td>$O(k \times (1 + \log \frac{m}{k}))$</td>
<td>[J01,J04]</td>
</tr>
<tr>
<td>Dichotomic</td>
<td>$O(k \times \log m)$</td>
<td>[HLSB06]</td>
</tr>
<tr>
<td>Insertion with Relaxation Variables</td>
<td>$O(m)$</td>
<td>[MSL11]</td>
</tr>
<tr>
<td>Deletion with Model Rotation</td>
<td>$O(m)$</td>
<td>[BLMS12,MSL11]</td>
</tr>
<tr>
<td>Progression</td>
<td>$O(k \times \log(1 + \frac{m}{k}))$</td>
<td>[MSJB13]</td>
</tr>
</tbody>
</table>

- **Additional Techniques:**
  - Restrict formula to unsatisfiable subsets
    - [BDTW93,HLSB06,DHN06,MSL11]
  - Check redundancy condition
    - [vMW08,MSL11,BLMS12]
  - Model rotation, recursive model rotation, etc.
    - [MSL11,BMS11,BLMS12,W12]
More on MUS Extraction

<table>
<thead>
<tr>
<th>Algorithm</th>
<th># Oracle Calls</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertion (Default)</td>
<td>$O(m \times k)$</td>
<td>[SP88]</td>
</tr>
<tr>
<td>Deletion (Default)</td>
<td>$O(m)$</td>
<td>[CD91,BDTW93]</td>
</tr>
<tr>
<td>QuickXplain</td>
<td>$O(k \times (1 + \log \frac{m}{k}))$</td>
<td>[J01,J04]</td>
</tr>
<tr>
<td>Dichotomic</td>
<td>$O(k \times \log m)$</td>
<td>[HLSB06]</td>
</tr>
<tr>
<td>Insertion with Relaxation Variables</td>
<td>$O(m)$</td>
<td>[MSL11]</td>
</tr>
<tr>
<td>Deletion with Model Rotation</td>
<td>$O(m)$</td>
<td>[BLMS12,MSL11]</td>
</tr>
<tr>
<td>Progression</td>
<td>$O(k \times \log(1 + \frac{m}{k}))$</td>
<td>[MSJB13]</td>
</tr>
</tbody>
</table>

- Additional Techniques:
  - Restrict formula to unsatisfiable subsets  [BDTW93,HLSB06,DHN06,MSL11]
  - Check redundancy condition            [vMW08,MSL11,BLMS12]
  - Model rotation, recursive model rotation, etc.  [MSL11,BMS11,BLMS12,W12]
Outline

CNF Encodings

SAT Embeddings

**SAT Oracles**

MUS Extraction
MaxSAT
2QBF

What Next in SAT-Based Problem Solving?
Given **unsatisfiable** formula, find **largest** subset of clauses that is satisfiable
Defining Maximum Satisfiability

- Given unsatisfiable formula, find largest subset of clauses that is satisfiable
- A Minimal Correction Subset (MCS) is an irreducible relaxation of the formula
Defining Maximum Satisfiability

- Given **unsatisfiable** formula, find **largest** subset of clauses that is satisfiable
- A **Minimal Correction Subset (MCS)** is an irreducible relaxation of the formula
- The MaxSAT solution is one of the **smallest** MCSes
MaxSAT Problem(s)

- **MaxSAT:**
  - All clauses are soft
  - Maximize number of satisfied soft clauses
  - Minimize number of unsatisfied soft clauses
MaxSAT Problem(s)

- **MaxSAT:**
  - All clauses are soft
  - Maximize number of satisfied soft clauses
  - Minimize number of unsatisfied soft clauses

- **Partial MaxSAT:**
  - Hard clauses must be satisfied
  - Minimize number of unsatisfied soft clauses
MaxSAT Problem(s)

- **MaxSAT:**
  - All clauses are soft
  - Maximize number of satisfied soft clauses
  - Minimize number of unsatisfied soft clauses

- **Partial MaxSAT:**
  - Hard clauses must be satisfied
  - Minimize number of unsatisfied soft clauses

- **Weighted MaxSAT**
  - Weights associated with (soft) clauses
  - Minimize sum of weights of unsatisfied clauses
MaxSAT Problem(s)

- **MaxSAT**:  
  - All clauses are soft  
  - Maximize number of satisfied soft clauses  
  - Minimize number of unsatisfied soft clauses

- **Partial MaxSAT**:  
  - Hard clauses must be satisfied  
  - Minimize number of unsatisfied soft clauses

- **Weighted MaxSAT**:  
  - Weights associated with (soft) clauses  
  - Minimize sum of weights of unsatisfied clauses

- **Weighted Partial MaxSAT**:  
  - Weights associated with soft clauses  
  - Hard clauses must be satisfied  
  - Minimize sum of weights of unsatisfied soft clauses
Definitions

- **Cost of assignment:**
  - Sum of weights of unsatisfied clauses

- **Optimum solution (OPT):**
  - Assignment with minimum cost

- **Upper Bound (UB):**
  - Assignment with cost not less than OPT
  - E.g. $\sum_{c_i \in \Phi} w_i + 1$; hard clauses may be inconsistent

- **Lower Bound (LB):**
  - No assignment with cost no larger than LB
  - E.g. $-1$; it may be possible to satisfy all soft clauses
Definitions

- **Cost of assignment:**
  - Sum of weights of *unsatisfied* clauses
- **Optimum solution (OPT):**
  - Assignment with *minimum* cost
- **Upper Bound (UB):**
  - Assignment with cost *not less* than OPT
  - E.g. \[\sum_{c_i \in \varphi} w_i + 1\]; hard clauses may be inconsistent
- **Lower Bound (LB):**
  - No assignment with cost *no larger* than LB
  - E.g. \[-1\]; it may be possible to satisfy all soft clauses
Iterative SAT Solving – Refine UB

- Require $\sum w_i r_i \leq UB_0 - 1$

![Diagram showing LB, OPT, and UB0 with the requirement $\sum w_i r_i \leq UB_0 - 1$]
Iterative SAT Solving – Refine UB

- Require $\sum w_i r_i \leq UB_0 - 1$
- While SAT, refine UB
  - New UB given by cost of unsatisfied clauses, i.e. $\sum w_i r_i$
Iterative SAT Solving – Refine UB

- Require $\sum w_i r_i \leq UB_0 - 1$
- While SAT, refine UB
  - New UB given by cost of unsatisfied clauses, i.e. $\sum w_i r_i$
Iterative SAT Solving – Refine UB

- Require $\sum w_i r_i \leq UB_0 - 1$
- While SAT, refine UB
  - New UB given by cost of unsatisfied clauses, i.e. $\sum w_i r_i$
- Repeat until constraint $\sum w_i r_i \leq UB_k - 1$ becomes UNSAT
  - $UB_k$ denotes the optimum value

![Diagram showing the relationship between LB, UB, OPT, UB_k, and UB_n. The diagram includes a line segment with LB at one end and UB at the other, with OPT, UB_k, and UB_n as labeled points along the line.]
Iterative SAT Solving – Refine UB

- Require $\sum w_i r_i \leq UB_0 - 1$
- While SAT, refine UB
  - New UB given by cost of unsatisfied clauses, i.e. $\sum w_i r_i$
- Repeat until constraint $\sum w_i r_i \leq UB_k - 1$ becomes UNSAT
  - $UB_k$ denotes the optimum value

- Worst-case # of iterations \textbf{exponential} on instance size
Iterative SAT Solving – Refine UB

- Require \( \sum w_i r_i \leq UB_0 - 1 \)
- While SAT, refine UB
  - New UB given by cost of unsatisfied clauses, i.e. \( \sum w_i r_i \)
- Repeat until constraint \( \sum w_i r_i \leq UB_k - 1 \) becomes UNSAT
  - \( UB_k \) denotes the optimum value

- Worst-case # of iterations **exponential** on instance size

- Example tools:
  - Minisat+: CNF encoding of constraints
  - SAT4J: native handling of constraints
  - QMaxSat: CNF encoding of constraints
  - ...
Fu&Malik’s Core-Guided Approach

Example CNF formula

\[\begin{align*}
x_6 \lor x_2 & \quad \neg x_6 \lor x_2 & \quad \neg x_2 \lor x_1 & \quad \neg x_1 \\
\neg x_6 \lor x_8 & \quad x_6 \lor \neg x_8 & \quad x_2 \lor x_4 & \quad \neg x_4 \lor x_5 \\
x_7 \lor x_5 & \quad \neg x_7 \lor x_5 & \quad \neg x_5 \lor x_3 & \quad \neg x_3
\end{align*}\]
Fu&Malik’s Core-Guided Approach

Formula is **UNSAT**; \( \text{OPT} \leq |\varphi| - 1 \); Get unsat core
Fu&Malik’s Core-Guided Approach

\[ x_6 \lor x_2 \quad \neg x_6 \lor x_2 \quad \neg x_2 \lor x_1 \lor r_1 \quad \neg x_1 \lor r_2 \]

\[ \neg x_6 \lor x_8 \quad x_6 \lor \neg x_8 \quad x_2 \lor x_4 \lor r_3 \quad \neg x_4 \lor x_5 \lor r_4 \]

\[ x_7 \lor x_5 \quad \neg x_7 \lor x_5 \quad \neg x_5 \lor x_3 \lor r_5 \quad \neg x_3 \lor r_6 \]

\[ \sum_{i=1}^{6} r_i \leq 1 \]

Add relaxation variables and AtMost1 constraint
Fu&Malik’s Core-Guided Approach

Formula is (again) **UNSAT**; $\text{OPT} \leq |\varphi| - 2$; Get unsat core
Fu&Malik’s Core-Guided Approach

\[ x_6 \lor x_2 \lor r_7 \quad \neg x_6 \lor x_2 \lor r_8 \quad \neg x_2 \lor x_1 \lor r_1 \lor r_9 \quad \neg x_1 \lor r_2 \lor r_{10} \]

\[ \neg x_6 \lor x_8 \quad x_6 \lor \neg x_8 \quad x_2 \lor x_4 \lor r_3 \quad \neg x_4 \lor x_5 \lor r_4 \]

\[ x_7 \lor x_5 \lor r_{11} \quad \neg x_7 \lor x_5 \lor r_{12} \quad \neg x_5 \lor x_3 \lor r_5 \lor r_{13} \quad \neg x_3 \lor r_6 \lor r_{14} \]

\[ \sum_{i=1}^{6} r_i \leq 1 \quad \sum_{i=7}^{14} r_i \leq 1 \]

Add new relaxation variables and AtMost1 constraint
Fu&Malik’s Core-Guided Approach

\[ x_6 \lor x_2 \lor r_7 \]
\[ \neg x_6 \lor x_2 \lor r_8 \]
\[ \neg x_2 \lor x_1 \lor r_1 \lor r_9 \]
\[ \neg x_1 \lor r_2 \lor r_{10} \]
\[ \neg x_6 \lor x_8 \]
\[ x_6 \lor \neg x_8 \]
\[ x_2 \lor x_4 \lor r_3 \]
\[ \neg x_4 \lor x_5 \lor r_4 \]
\[ x_7 \lor x_5 \lor r_{11} \]
\[ \neg x_7 \lor x_5 \lor r_{12} \]
\[ \neg x_5 \lor x_3 \lor r_5 \lor r_{13} \]
\[ \neg x_3 \lor r_6 \lor r_{14} \]
\[ \sum_{i=1}^{6} r_i \leq 1 \]
\[ \sum_{i=7}^{14} r_i \leq 1 \]

Instance is now \textit{SAT}
Fu&Malik’s Core-Guided Approach

\[ x_6 \lor x_2 \lor r_7 \]
\[ \neg x_6 \lor x_2 \lor r_8 \]
\[ \neg x_2 \lor x_1 \lor r_1 \lor r_9 \]
\[ \neg x_1 \lor r_2 \lor r_{10} \]
\[ \neg x_6 \lor x_8 \]
\[ x_6 \lor \neg x_8 \]
\[ x_2 \lor x_4 \lor r_3 \]
\[ \neg x_4 \lor x_5 \lor r_4 \]
\[ x_7 \lor x_5 \lor r_{11} \]
\[ \neg x_7 \lor x_5 \lor r_{12} \]
\[ \neg x_5 \lor x_3 \lor r_5 \lor r_{13} \]
\[ \neg x_3 \lor r_6 \lor r_{14} \]

\[ \sum_{i=1}^{6} r_i \leq 1 \]
\[ \sum_{i=7}^{14} r_i \leq 1 \]

MaxSAT solution is \( |\varphi| - \mathcal{I} = 12 - 2 = 10 \)
Organization of Fu\&Malik’s Algorithm

• Clauses characterized as:
  – **Soft**: initial set of soft clauses
  – **Hard**: initially hard, or added during execution of algorithm
    ▶ E.g. clauses from AtMost1 constraints

• While exist unsatisfiable cores
  – Add fresh set $B$ of relaxation variables to soft clauses in core
  – Add new AtMost1 constraint
    \[
    \sum_{b_i \in B} b_i \leq 1
    \]
    ▶ At most 1 relaxation variable from set $B$ can take value 1

• (Partial) MaxSAT solution is $|\varphi| - I$
  – $I$: number of iterations (≡ number of computed unsat cores)
Organization of Fu&Malik’s Algorithm

- Clauses characterized as:
  - **Soft**: initial set of soft clauses
  - **Hard**: initially hard, or added during execution of algorithm
    - E.g. clauses from AtMost1 constraints

- While exist unsatisfiable cores
  - Add fresh set $B$ of relaxation variables to soft clauses in core
  - Add new AtMost1 constraint

  $\sum_{b_i \in B} b_i \leq 1$

  - At most 1 relaxation variable from set $B$ can take value 1

- (Partial) MaxSAT solution is $|\varphi| - I$
  - $I$: number of iterations ($\equiv$ number of computed unsat cores)

- Can be adapted for weighted MaxSAT

  [FM06, ABL09a, MMSP09]
Oracle-Based MaxSAT Solving I

- **Iterative:**
  - Linear search SAT/UNSAT (refine UB)
  - Linear search UNSAT/SAT (refine LB)
  - Binary search
  - Bit-based
  - Mixed linear/binary search

- **Core-Guided:**
  - FM/(W)MSU1.X/WPM1
  - (W)MSU3
  - (W)MSU4
  - (W)PM2
  - Core-guided binary search (w/ disjoint cores)
    - Bin-Core, Bin-Core-Dis, Bin-Core-Dis2

- **Iterative subsetting**
### Oracle MaxSAT Solving II

- A sample of recent algorithms:

<table>
<thead>
<tr>
<th>Algorithm</th>
<th># Oracle Calls</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear search SU</td>
<td>Exponential</td>
<td>[e.g. LP10]</td>
</tr>
<tr>
<td>Binary search</td>
<td>Linear</td>
<td>[e.g. FM06]</td>
</tr>
<tr>
<td>WMSU1/WPM1</td>
<td>Exponential*</td>
<td>[FM06, MSM08, MMSP09, ABL09a, ABGL12]</td>
</tr>
<tr>
<td>WPM2</td>
<td>Exponential*</td>
<td>[ABL10, ABGL13]</td>
</tr>
<tr>
<td>Bin-Core-Dis</td>
<td>Linear</td>
<td>[HMMS11, MHMS12]</td>
</tr>
<tr>
<td>Iterative subsetting</td>
<td>Exponential</td>
<td>[DB11, DB13a, DB13b]</td>
</tr>
</tbody>
</table>

* Weighted case; depends on computed cores

- Example MaxSAT solvers:
  - MSUnCore; WPM1, WPM2; QMaxSAT; SAT4J; etc.
Outline

- CNF Encodings
- SAT Embeddings
- SAT Oracles
  - MUS Extraction
  - MaxSAT
  - 2QBF
- What Next in SAT-Based Problem Solving?
Given: $\exists X \forall Y. \phi$, where $\phi$ is a propositional formula

Question: Is there an assignment $\tau$ to $X$ such that $\forall Y. \phi[X/\tau]$?
Given: $\exists X \forall Y. \phi$, where $\phi$ is a propositional formula

Question: Is there an assignment $\tau$ to $X$ such that $\forall Y. \phi[X/\tau]$?

Example

$\exists x_1, x_2 \forall y_1, y_2. (x_1 \rightarrow y_1) \land (x_2 \rightarrow y_2)$

solution: $x_1 = 0, x_2 = 0$
Motivation

- $\Sigma_2^P$ complete
- interesting problems in this class, e.g. certain nonmonotonic reasoning, aspects of model checking, conformant planning
- separate track at QBF Eval
Looking at Assignments
Looking at Assignments

<table>
<thead>
<tr>
<th>( X )</th>
<th>( \xi )</th>
<th>( \mu )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Diagram:

- \( Y \)
- \( \mu \)
- \( \xi \)
- 1
## Looking at Assignments

<table>
<thead>
<tr>
<th></th>
<th>$Y$</th>
<th>$\mu$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X$</td>
<td><img src="matrix.png" alt="Matrix" /></td>
<td></td>
</tr>
<tr>
<td>$\xi$</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

...
Looking at Assignments

<table>
<thead>
<tr>
<th></th>
<th>Y</th>
<th>μ</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ξ</td>
<td>1 0</td>
<td>0 1</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>τ</td>
<td>1 1</td>
<td>1 1</td>
</tr>
</tbody>
</table>
Looking at Assignments

<table>
<thead>
<tr>
<th></th>
<th>$\phi[Y/\mu]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\xi$</td>
<td>1 0 ... 0 1 ... 1</td>
</tr>
<tr>
<td>$\tau$</td>
<td>1 1 ... 1 1 ... 1</td>
</tr>
<tr>
<td>$X$</td>
<td>$\mu$</td>
</tr>
</tbody>
</table>
Expanding $\exists X \forall Y. \phi$ into SAT

$$\exists X \forall Y. \phi \rightarrow \text{SAT} \left( \bigwedge_{\mu \in B^{|Y|}} \phi[Y/\mu] \right)$$
Expanding $\exists X \forall Y. \phi$ into SAT

$\exists X \forall Y. \phi \rightarrow \text{SAT} \left( \bigwedge_{\mu \in B^{|Y|}} \phi[Y/\mu] \right)$

Example

$\exists x_1, x_2 \forall y_1, y_2. (x_1 \leftrightarrow y_1) \land (x_2 \leftrightarrow y_2) \land (\bar{x}_1 \lor \bar{x}_2)$

Expansion:

$$(x_1 \leftrightarrow 0) \land (x_2 \leftrightarrow 0) \land (\bar{x}_1 \lor \bar{x}_2)$$

$$\land (x_1 \leftrightarrow 0) \land (x_2 \leftrightarrow 1) \land (\bar{x}_1 \lor \bar{x}_2)$$

$$\land (x_1 \leftrightarrow 1) \land (x_2 \leftrightarrow 0) \land (\bar{x}_1 \lor \bar{x}_2)$$

$$\land (x_1 \leftrightarrow 1) \land (x_2 \leftrightarrow 1) \land (\bar{x}_1 \lor \bar{x}_2)$$
Expanding $\exists X \forall Y. \phi$ into SAT

$\exists X \forall Y. \phi \rightarrow \text{SAT} \left( \bigwedge_{\mu \in B^{|Y|}} \phi[Y/\mu] \right)$

Example

$\exists x_1, x_2 \forall y_1, y_2. (x_1 \leftrightarrow y_1) \land (x_2 \leftrightarrow y_2) \land (\bar{x}_1 \lor \bar{x}_2)$

Expansion:

$(x_1 \leftrightarrow 0) \land (x_2 \leftrightarrow 0) \land (\bar{x}_1 \lor \bar{x}_2) \land (x_1 \leftrightarrow 0) \land (x_2 \leftrightarrow 1) \land (\bar{x}_1 \lor \bar{x}_2) \land (x_1 \leftrightarrow 1) \land (x_2 \leftrightarrow 0) \land (\bar{x}_1 \lor \bar{x}_2) \land (x_1 \leftrightarrow 1) \land (x_2 \leftrightarrow 1) \land (\bar{x}_1 \lor \bar{x}_2)$
Abstraction of $\exists X \forall Y. \phi$

- Consider only some set of assignments $\omega \subseteq B^{|Y|}$

$$\bigwedge_{\mu \in \omega} \phi[Y/\mu]$$
Abstraction of $\exists X \forall Y. \phi$

- Consider only some set of assignments $\omega \subseteq B^{|Y|}$
  \[ \bigwedge_{\mu \in \omega} \phi[Y/\mu] \]

- If a solution to the problem is a solution to the abstraction
  \[ \bigwedge_{\mu \in B^{|Y|}} \phi[Y/\mu] \Rightarrow \bigwedge_{\mu \in \omega} \phi[Y/\mu] \]

But not the other way around, a solution to an abstraction is not necessarily a solution to the original problem.
Abstraction of $\exists X \forall Y. \phi$

- Consider only some set of assignments $\omega \subseteq B^{|Y|}$

$$\bigwedge_{\mu \in \omega} \phi[Y/\mu]$$

- If a solution to the problem is a solution to the abstraction

$$\bigwedge_{\mu \in B^{|Y|}} \phi[Y/\mu] \implies \bigwedge_{\mu \in \omega} \phi[Y/\mu]$$

- But not the other way around, a solution to an abstraction is not necessarily a solution to the original problem.
CEGAR Loop

**input**: $\exists X \forall Y. \phi$

**output**: $(\text{true}, \tau)$ if there exists $\tau$ s.t. $\forall Y. \phi[X/\tau]$, $(\text{false}, -)$ otherwise

$\omega \leftarrow \emptyset$

**while** true **do**

- $(\text{outc}_1, \tau) \leftarrow \text{SAT}(\bigwedge_{\mu \in \omega} \phi[Y/\mu])$;  
  // find a candidate
- **if** $\text{outc}_1 = \text{false}$ **then**
  - **return** $(\text{false}, -)$;  
  // no candidate found
- **else**
  - "$\tau$ is a solution";  
    // solution check
  **then**
  - **return** $(\text{true}, \tau)$
  **else**
  - "Grow $\omega$";  
    // refinement
- **end**
- **end**
**CEGAR Loop**

**input**: $\exists X \forall Y. \phi$

**output**: $(\text{true}, \tau)$ if there exists $\tau$ s.t. $\forall Y. \phi[X/\tau]$

(false, −) otherwise

$\omega \leftarrow \emptyset$;

while true do

$(\text{outc}_1, \tau) \leftarrow \text{SAT}(\land_{\mu \in \omega} \phi[Y/\mu])$; // find a candidate

if outc$_1$ = false then
  return (false, −); // no candidate found
endif

if “$\tau$ is a solution”; // solution check
  return (true, $\tau$)
else
  “Grow $\omega$”; // refinement
endif

done
A value $\tau$ is a solution to $\exists X \forall Y. \phi$ iff

$$\forall Y. \phi[X/\tau] \iff \text{UNSAT}(\neg \phi[X/\tau])$$
A value $\tau$ is a solution to $\exists X \forall Y. \phi$ iff

$$\forall Y. \phi[X/\tau] \iff \text{UNSAT}(\neg \phi[X/\tau])$$

If SAT($\neg \phi[X/\tau]$) by some $\mu$, then $\mu$ is a counterexample to $\tau$
A value $\tau$ is a solution to $\exists X \forall Y. \phi$ iff

$$\forall Y. \phi[X/\tau] \iff \text{UNSAT}(\neg \phi[X/\tau])$$

If $\text{SAT}(\neg \phi[X/\tau])$ by some $\mu$, then $\mu$ is a counterexample to $\tau$

Example

$\exists x_1, x_2 \ \forall y_1, y_2. (x_1 \rightarrow y_1) \land (x_2 \rightarrow y_2)$

- candidate: $x_1 = 1, x_2 = 1$
- counterexamples: $y_1 = 0, y_2 = 0$
  $\quad y_1 = 0, y_2 = 1$
  $\quad y_1 = 1, y_2 = 0$
## Refinement

\[ Y \]

\[ \begin{array}{ccc}
\tau_2 & 1 & 1 & 1 & 0 \\
\tau_1 & 1 & 1 & 1 & 0 \\
\tau & 1 & 1 & \ldots & 1 & 0 \\
\omega & \ldots & \ldots & \end{array} \]
Refinement

\[ \begin{array}{cccc}
\tau_2 & 1 & 1 & 1 & 0 \\
\tau_1 & 1 & 1 & 1 & 0 \\
\tau_0 & 1 & 1 & \ldots & 1 & 0 \\
\end{array} \]
Refinement

\[
\begin{array}{cccc}
\chi & Y & \mu \\
\tau_1 & 1 & 1 & 1 & 0 \\
\tau_2 & 1 & 1 & 1 & 0 \\
\vdots & 1 & 1 & \ldots & 1 & 0 & \cdots \\
\omega & & & & & & \\
\omega' & & & & & &
\end{array}
\]
AReQS (Abstraction Refinement-based QBF Solver)

**input**: $\exists X \forall Y. \phi$

**output**: $(true, \tau)$ if there exists $\tau$ s.t. $\forall Y. \phi[X/\tau]$, $(false, -)$ otherwise

\[
\omega \leftarrow \emptyset; \quad \text{// start with the empty expansion}
\]

\[
\text{while true do}
\]

\[
(\text{outc}_1, \tau) \leftarrow \text{SAT}(\bigwedge_{\mu \in \omega} \phi[Y/\mu]); \quad \text{// find a candidate}
\]

\[
\text{if outc}_1 = \text{false then}
\]

\[
\text{return (false, -);} \quad \text{// no candidate found}
\]

\[
\text{end}
\]

\[
(\text{outc}_2, \mu) \leftarrow \text{SAT}(\neg \phi[X/\tau]); \quad \text{// find a counterexample}
\]

\[
\text{if outc}_2 = \text{false then}
\]

\[
\text{return (true, } \tau \text{);} \quad \text{// candidate is a solution}
\]

\[
\text{end}
\]

\[
\omega \leftarrow \omega \cup \{\mu\}; \quad \text{// refine}
\]

\[
\text{end}
\]
AReQS — Conclusions

- ... is a CEGAR-based algorithm for 2QBF

[JMS11]
... is a CEGAR-based algorithm for 2QBF [JMS11]
... uses SAT solver as an oracle
• ... is a CEGAR-based algorithm for 2QBF [JMS11]
• ... uses SAT solver as an oracle
• ... gradually expands given 2QBF into a SAT formula
AReQS — Conclusions

- ... is a CEGAR-based algorithm for 2QBF
- ... uses SAT solver as an oracle
- ... gradually expands given 2QBF into a SAT formula
- Can be extended to arbitrary number of levels by recursion (RAReQS)

[JMS11]

[JKMSC12]
Outline

CNF Encodings

SAT Embeddings

SAT Oracles

What Next in SAT-Based Problem Solving?
Remarkable (and increasing) number of applications of SAT

Can use SAT for solving problems in different complexity classes
- \( \text{FP}^{\text{NP}}[\log n], \text{FP}^{\text{NP}}, \ldots \)
- E.g. tackling problems in the polynomial hierarchy

Many new recent algorithms for concrete problems
- MaxSAT
- MUSes
- MCSes
- Enumeration problems
- ...

Better encodings?

White-box vs. black-box approaches?
- But use of oracles inevitable in many cases
Thank You
<table>
<thead>
<tr>
<th>Reference</th>
<th>Title</th>
</tr>
</thead>
</table>
GN02 E. Goldberg, Y. Novikov: BerkMin: A Fast and Robust Sat-Solver. DATE 2002: 142-149


SB09 N. Sorensson, A. Biere: Minimizing Learned Clauses. SAT 2009: 237-243

VG09 A. Van Gelder: Improved Conflict-Clause Minimization Leads to Improved Propositional Proof Traces. SAT 2009: 141-146

AS09 G. Audemard, L. Simon: Predicting Learnt Clauses Quality in Modern SAT Solvers. IJCAI 2009: 399-404

<table>
<thead>
<tr>
<th>Reference</th>
<th>Author(s)</th>
<th>Title</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>dK89</td>
<td>Johan de Kleer</td>
<td>A Comparison of ATMS and CSP Techniques.</td>
<td>IJCAI 1989: 290-296</td>
</tr>
<tr>
<td>GJ96</td>
<td>R. Genisson, P. Jegou</td>
<td>Davis and Putnam were Already Checking Forward.</td>
<td>ECAI 1996: 180-184</td>
</tr>
<tr>
<td>W00</td>
<td>T. Walsh</td>
<td>SAT v CSP.</td>
<td>CP 2000: 441-456</td>
</tr>
</tbody>
</table>
FP01  A. Frisch, T. Peugniez: Solving Non-Boolean Satisfiability Problems with Stochastic Local Search. IJCAI 2001: 282-290


S05   C. Sinz: Towards an Optimal CNF Encoding of Boolean Cardinality Constraints. CP 2005: 827-831


G07   M. Gavanelli: The Log-Support Encoding of CSP into SAT. CP 2007: 815-822


BBR09  O. Bailleux, Y. Boufkhad, O. Roussel: New Encodings of Pseudo-Boolean Constraints into CNF. SAT 2009: 181-194


CZI10  M. Codish, M. Zazon-Ivry: Pairwise Cardinality Networks. LPAR (Dakar) 2010: 154-172


<table>
<thead>
<tr>
<th>Reference</th>
<th>Authors</th>
<th>Title</th>
<th>Journal</th>
<th>Volume</th>
<th>Issue</th>
<th>Pages</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>S07</td>
<td>R. Sebastiani</td>
<td>Lazy Satisability Modulo Theories</td>
<td>JSAT</td>
<td>3</td>
<td>3-4</td>
<td>141-224</td>
<td>2007</td>
</tr>
<tr>
<td>BSST09</td>
<td>C. Barrett, R. Sebastiani, S. Seshia, C. Tinelli</td>
<td>Satisfiability Modulo Theories</td>
<td>Handbook of Satisfiability 2009</td>
<td></td>
<td></td>
<td>825-885</td>
<td></td>
</tr>
<tr>
<td>LBP10</td>
<td>D. Le Berre, A. Parrain</td>
<td>The Sat4j library, release 2.2</td>
<td>JSAT</td>
<td>7</td>
<td>2-3</td>
<td>59-6</td>
<td>2010</td>
</tr>
<tr>
<td>KZFH12</td>
<td>M. Koshimura, T. Zhang, H. Fujita, R. Hasegawa</td>
<td>QMaxSAT: A Partial Max-SAT Solver</td>
<td>JSAT</td>
<td>8</td>
<td>1/2</td>
<td>95-100</td>
<td>2012</td>
</tr>
<tr>
<td>Reference</td>
<td>Authors</td>
<td>Title</td>
<td>Conference/Journal Details</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-----------</td>
<td>---------</td>
<td>-------</td>
<td>-----------------------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BDTW93</td>
<td>R. R. Bakker, F. Dikker, F. Tempelman, P. M. Wognum</td>
<td>Diagnosing and Solving Over-Determined Constraint Satisfaction Problems</td>
<td>IJCAI 1993: 276-281</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reference Code</td>
<td>Authors</td>
<td>Title</td>
<td>Conference/Volume/Issue</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>----------------</td>
<td>--------------------------------</td>
<td>----------------------------------------------------------------------</td>
<td>------------------------------------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MSL11</td>
<td>J. Marques-Silva, I. Lynce</td>
<td>On Improving MUS Extraction Algorithms.</td>
<td>SAT 2011: 159-173</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BMS11</td>
<td>A. Belov, J. Marques-Silva</td>
<td>Accelerating MUS extraction with recursive model rotation.</td>
<td>FMCAD 2011: 37-40</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>W12</td>
<td>Siert Wieringa</td>
<td>Understanding, Improving and Parallelizing MUS Finding Using Model Rotation.</td>
<td>CP 2012: 672-687</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MSJB13</td>
<td>J. Marques-Silva, M. Janota, A. Belov</td>
<td>Minimal Sets over Monotone Predicates in Boolean Formulae.</td>
<td>CAV 2013</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reference</td>
<td>Authors &amp; Title</td>
<td>Conference/Journal</td>
<td>Page(s)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-----------</td>
<td>----------------</td>
<td>--------------------</td>
<td>---------</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FM06</td>
<td>Z. Fu, S. Malik: On Solving the Partial MAX-SAT Problem</td>
<td>SAT 2006</td>
<td>252-265</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MSP08</td>
<td>J. Marques-Silva, Jordi Planes: Algorithms for Maximum Satisfiability using Unsatisfiable Cores</td>
<td>DATE 2008</td>
<td>408-413</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ABL09a</td>
<td>C. Ansotegui, M. Bonet, J. Levy: Solving (Weighted) Partial MaxSAT through Satisfiability Testing</td>
<td>SAT 2009</td>
<td>427-440</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ABL09b</td>
<td>C. Ansotegui, M. L. Bonet, J. Levy: On Solving MaxSAT Through SAT</td>
<td>CCIA 2009</td>
<td>284-292</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ABL10</td>
<td>C. Ansotegui, M. Bonet, J. Levy: A New Algorithm for Weighted Partial MaxSAT</td>
<td>AAAI 2010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reference</td>
<td>Authors</td>
<td>Title</td>
<td>Conference/Proceedings</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>------------</td>
<td>-------------------------------------------------------------------------</td>
<td>-------------------------------------------------------------------------------------------</td>
<td>-----------------------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DB11</td>
<td>J. Davies, F. Bacchus</td>
<td>Solving MAXSAT by Solving a Sequence of Simpler SAT Instances.</td>
<td>CP 2011: 225-239</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MHMS12</td>
<td>A. Morgado, F. Heras, J. Marques-Silva</td>
<td>Improvements to Core-Guided Binary Search for MaxSAT.</td>
<td>SAT 2012.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ABGL12</td>
<td>C. Ansotegui, M. Bonet, J. Gabas, J. Levy</td>
<td>Improving SAT-Based Weighted MaxSAT Solvers.</td>
<td>CP 2012: 86-101</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DB13a</td>
<td>J. Davies, F. Bacchus</td>
<td>Exploiting the Power of MIP Solvers in MaxSAT.</td>
<td>SAT 2013: 166-181</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ABGL13</td>
<td>C. Ansotegui, M. Bonet, J. Gabas and J. Levy</td>
<td>Improving WPM2 for (Weighted) Partial MaxSAT.</td>
<td>CP 2013</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DB13b</td>
<td>J. Davies and F. Bacchus</td>
<td>Postponing Optimization to Speed Up MaxSAT Solving.</td>
<td>CP 2013</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reference</td>
<td>Authors and Title</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>------------</td>
<td>------------------------------------------------------------------------------------------------------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JMS11</td>
<td>M. Janota, J. Marques-Silva: Abstraction-Based Algorithm for 2QBF. SAT 2011: 230-244</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JKMSC12</td>
<td>M. Janota, W. Klieber, J. Marques-Silva, E. Clarke: Solving QBF with Counterexample Guided Refinement. SAT 2012: 114-128</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KJMSC13</td>
<td>W. Klieber, M. Janota, J. Marques-Silva, E. Clarke: Solving QBF with Free Variables. CP 2013</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
References – Additional References

C71  S. Cook: The Complexity of Theorem-Proving Procedures. STOC 1971: 151-158
SP04 S. Subbarayan, D. Pradhan: NiVER: Non-increasing Variable Elimination Resolution for Preprocessing SAT Instances. SAT 2004: 276-291
EB05 N. Een, A. Biere: Effective Preprocessing in SAT Through Variable and Clause Elimination. SAT 2005: 61-75
HJB11 M. Heule, M. Jarvisalo, A. Biere: Efficient CNF Simplification Based on Binary Implication Graphs. SAT 2011: 201-215