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
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
Preliminaries

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

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

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

- Resolution rule: [DP60,R65]

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

\[(\alpha \lor \beta)\]

- Complete proof system for propositional logic
Resolution

- **Resolution rule:**

\[
\begin{array}{c}
(\alpha \lor x) \\
(\beta \lor \bar{x}) \\
\hline \\
(\alpha \lor \beta)
\end{array}
\]

- Complete proof system for propositional logic

\[
\begin{array}{c}
(x \lor a) \\
(\bar{x} \lor a) \\
(\bar{y} \lor \bar{a}) \\
(y \lor \bar{a})
\end{array}
\]

\[
\begin{array}{c}
(a) \\
(\bar{a})
\end{array}
\]

- Extensively used with (CDCL) SAT solvers
Resolution

- Resolution rule:  
  \[
  \frac{(\alpha \lor x)}{(\beta \lor \bar{x})} \quad \frac{(\beta \lor \bar{x})}{(\alpha \lor \beta)}
  \]
  
- Complete proof system for propositional logic

- Extensively used with (CDCL) SAT solvers

- Self-subsuming resolution (with \( \alpha' \subseteq \alpha \)):
  \[
  \frac{(\alpha \lor x)}{(\alpha' \lor \bar{x})} \quad \frac{(\alpha' \lor \bar{x})}{(\alpha)}
  \]

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

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

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

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

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

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

\begin{tabular}{|c|c|c|}
  \hline
  Level & Dec. & Unit Prop. \\
  \hline
  0 & \emptyset & \rightarrow s \\
  1 & w & \rightarrow a \\
  2 & x & \rightarrow b \\
  3 & y & \\
  4 & z & \rightarrow c \rightarrow d \\
  \hline
\end{tabular}
\( \mathcal{F} = (r) \land (\bar{r} \lor s) \land \\
(\bar{w} \lor a) \land (\bar{x} \lor \bar{a} \lor b) \\
(\bar{y} \lor \bar{z} \lor c) \land (\bar{b} \lor \bar{c} \lor d) \)

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

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

Basic Definitions

DPLL Solvers

CDCL Solvers

What Next in CDCL Solvers?
The DPLL Algorithm

- Optional: pure literal rule
The DPLL Algorithm

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

• Optional: pure literal rule
The DPLL Algorithm

Unassigned variables?
- Y: Assign value to variable
  - Satisfiable
  - Unit propagation
  - Conflict?
    - Y: Can undo decision?
      - Y: Backtrack & flip variable
      - N: Unsatisfiable
    - N: Conflict?
      - Y: Can undo decision?
        - Y: Backtrack & flip variable
        - N: Unsatisfiable
  - N: Unit propagation

Optional: pure literal rule

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

Levels:
- 0: $\emptyset$
- 1: $x$
- 2: $y$
- 3: $a \rightarrow b \rightarrow \bot$
The DPLL Algorithm

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

<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>$\bar{a}$</td>
<td>$\bar{b}$</td>
</tr>
</tbody>
</table>

- Optional: pure literal rule
The DPLL Algorithm

- Optional: pure literal rule

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

<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>(\bar{y})</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>(a) (\rightarrow) (b) (\rightarrow) (\perp)</td>
<td></td>
</tr>
</tbody>
</table>
The DPLL Algorithm

- **Optional:** pure literal rule

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

<table>
<thead>
<tr>
<th>Level</th>
<th>Dec.</th>
<th>Unit Prop.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$\emptyset$</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>$x$</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>$\bar{y}$</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>$\bar{a} \rightarrow \bar{b} \rightarrow \bot$</td>
<td></td>
</tr>
</tbody>
</table>

**Diagram:**

1. Level 0: Start with an empty decision list.
2. Level 1: Assign $x$. Continue with $y$.
3. Level 2: Assign $\bar{y}$. Continue with $\bar{a} \rightarrow \bar{b} \rightarrow \bot$.
The DPLL Algorithm

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

- **Level**
  - 0: \( \emptyset \)
  - 1: \( \bar{x} \rightarrow y \)
  - 2: \( a \rightarrow b \rightarrow \bot \)

- **Optional:** pure literal rule

**Diagram:**
- **Level 0:** \( \emptyset \)
- **Level 1:** \( \bar{x} \rightarrow y \)
- **Level 2:** \( a \rightarrow b \rightarrow \bot \)

**Decision Tree:**
- **Level 0:** \( a, \bar{a}, a, \bar{a} \)
- **Level 1:** \( x, \bar{x} \)
- **Level 2:** \( y, \bar{y} \)
- **Level 3:** \( \bot \)
The DPLL Algorithm

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

<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>\bar{x}</td>
<td>y</td>
</tr>
<tr>
<td>2</td>
<td>\bar{a}</td>
<td>\bar{b}</td>
</tr>
</tbody>
</table>

- Optional: pure literal rule

Diagram:

- Unassigned variables? (Y: Satisfiable)
- Unit propagation
- Conflict? (N: Unsatisfiable)
- Can undo decision? (N)
- Backtrack & flip variable

Diagram nodes:
- x
- y
- \bar{y}
- a
- \bar{a}
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)
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>$\emptyset$</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>$z \rightarrow a \rightarrow \perp$</td>
<td>$b$</td>
</tr>
</tbody>
</table>
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: Decision variable & literals assigned at lower decision levels
- Create new clause: \((\overline{x} \lor \overline{z})\)
- Can relate clause learning with resolution
  Learned clauses result from (selected) resolution operations
Clause Learning

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

- Analyze conflict
  - Reasons: x and z
    - Decision variable & literals assigned at lower decision levels
### Clause Learning

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

#### Analyze conflict
- Reasons: \( x \) and \( z \)
  - Decision variable & literals assigned at lower decision levels
- Create new clause: \( (\bar{x} \lor \bar{z}) \)
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>(\overline{a} \lor \overline{b})</td>
</tr>
<tr>
<td>2</td>
<td>y</td>
<td>(\overline{z} \lor b)</td>
</tr>
<tr>
<td>3</td>
<td>z</td>
<td>(\overline{x} \lor \overline{z} \lor 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>((\bar{a} \lor \bar{b})) ((\bar{z} \lor b)) ((\bar{x} \lor \bar{z} \lor a))</td>
</tr>
<tr>
<td>2</td>
<td>(y)</td>
<td>((\bar{a} \lor \bar{z}))</td>
</tr>
<tr>
<td>3</td>
<td>(z)</td>
<td>((\bar{x} \lor \bar{z}))</td>
</tr>
<tr>
<td></td>
<td>(a)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(b)</td>
<td></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
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

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

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

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

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

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

- Learned clauses are always asserting [MSS96,MSS99]
- Backtracking differs from plain DPLL:
  - Always backtrack after a conflict [MMZZM01]

\[
\neg x \land \neg z
\]

is asserting at decision level 1
**Clause Learning – After Backtracking**

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

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

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

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

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

<table>
<thead>
<tr>
<th>Level</th>
<th>Dec.</th>
<th>Unit Prop.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(\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>

Level Dec. Unit Prop.

<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>(\bar{z})</td>
</tr>
</tbody>
</table>

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

References:
- 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>
<tr>
<td></td>
<td></td>
<td>c</td>
</tr>
<tr>
<td></td>
<td></td>
<td>↓</td>
</tr>
<tr>
<td></td>
<td></td>
<td>b</td>
</tr>
<tr>
<td></td>
<td></td>
<td>↓</td>
</tr>
<tr>
<td></td>
<td></td>
<td>⊥</td>
</tr>
</tbody>
</table>
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>$a \rightarrow c$</td>
</tr>
</tbody>
</table>

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

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

- Learn clause \((\overline{w} \lor \overline{x} \lor \overline{y} \lor \overline{z})\)
- But \(a\) is an UIP

\[
\begin{align*}
(\overline{b} \lor \overline{c}) & \quad (\overline{w} \lor c) & \quad (\overline{x} \lor \overline{a} \lor b) & \quad (\overline{y} \lor \overline{z} \lor a) \\
(\overline{w} \lor \overline{b}) & \quad (\overline{w} \lor \overline{x} \lor \overline{a}) & \quad (\overline{w} \lor \overline{x} \lor \overline{y} \lor \overline{z})
\end{align*}
\]
Unique Implication Points (UIPs)

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

- Learn clause \((\bar{w} \lor \bar{x} \lor \bar{y} \lor \bar{z})\)
- But \(a\) is an UIP
- Learn clause \((\bar{w} \lor \bar{x} \lor \bar{a})\)
## Multiple UIPs

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

- **First UIP**: Learn clause \(\_\_\_\_\_\)  
  - But there can be more than 1 UIP

- **Second UIP**: Learn clause \(\_\_\_\_\)  
  - In practice smaller clauses more effective

- Multiple UIPs proposed in GRASP [MSS96]  
  - First UIP learning proposed in Cha [MMZZM01]

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

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

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

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

Multiple UIPs proposed in GRASP [MSS96]

- First UIP learning proposed in Cha [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) (a) (c) (s) (b) (\perp)</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

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

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

- Multiple UIPs proposed in GRASP [MSS96]
- First UIP learning proposed in Cha [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 \((\lnot w \lor \lnot y \lor \lnot a)\)
- But there can be more than 1 UIP
- **Second UIP:**
  - Learn clause \((\lnot x \lor \lnot z \lor a)\)
- In practice smaller clauses more effective
  - Compare with \((\lnot w \lor \lnot x \lor \lnot y \lor \lnot z)\)
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 b ⊥</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
  - First UIP learning proposed in Chaff
- Not used in recent state of the art CDCL SAT solvers

[MSS96]
[MMZZM01]
[SSS12]
Multiple UIPs

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

[MMZZM01]

[SSS12]

[MSS96]
## 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>

Diagram:
- Level 0: Dec. ∅
- Level 1: Dec. x 
  - x → b
- Level 2: Dec. y
  - y → c
  - c → ⊥
- Level 3: Dec. z
  - z → ⊥
  - z → a
  - a → ⊥
Clause Minimization I

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

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

\[
\begin{align*}
(\tilde{a} \lor \tilde{c}) &\quad (\tilde{z} \lor \tilde{b} \lor c) &\quad (\tilde{x} \lor \tilde{y} \lor \tilde{z} \lor a) \\
(\tilde{z} \lor \tilde{b} \lor \tilde{a}) \\
(\tilde{x} \lor \tilde{y} \lor \tilde{z} \lor \tilde{b})
\end{align*}
\]

[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 → b</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>y</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>z → c</td>
<td></td>
</tr>
</tbody>
</table>

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

\[
\begin{align*}
(\bar{a} \lor \bar{c}) & \quad (\bar{z} \lor \bar{b} \lor c) \\
(\bar{z} \lor \bar{b} \lor \bar{a}) & \\
(\bar{x} \lor \bar{y} \lor \bar{z} \lor \bar{b}) & \\
(\bar{x} \lor \bar{y} \lor \bar{z}) & \quad (\bar{x} \lor b)
\end{align*}
\]
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)
• Learn clause \((\overline{x} \lor \overline{y} \lor \overline{z})\)
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>$a$ \rightarrow $c$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$b$</td>
</tr>
<tr>
<td>2</td>
<td>$x$</td>
<td>$e$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$d$ \rightarrow $\bot$</td>
</tr>
</tbody>
</table>

- Cannot apply self-subsuming resolution
- Resolving with reason of $c$ yields $(\overline{w} \_ \overline{x} \_ \overline{a} \_ \overline{b})$
- Can apply recursive minimization
- Learn clause $(\overline{w} \_ \overline{x})$
- Marked nodes: literals in learned clause
- [SB09]
- Trace back from $c$ until marked nodes or new nodes
- Learn clause if only marked nodes visited
# 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 → c, b</td>
</tr>
<tr>
<td>2</td>
<td>x</td>
<td>e → d, ⊥</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>$a$  $c$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$b$  $c$</td>
</tr>
<tr>
<td>2</td>
<td>$x$</td>
<td>$e$  $\perp$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$d$  $\perp$</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Level</th>
<th>Dec.</th>
<th>Unit Prop.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>∅</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>w→a→c</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>x→e</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]
Clause Minimization II

Level | Dec. | Unit Prop. |
--- | --- | --- |
0 | $\emptyset$ | |
1 | $w$ | $a$ | $c$ |
2 | $x$ | $e$ |

- **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></td>
<td>$b$</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>$x$</td>
<td>$e$ $\bot$</td>
</tr>
<tr>
<td></td>
<td>$d$</td>
<td></td>
</tr>
</tbody>
</table>

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

#### Annotation
- **Marked nodes**: literals in learned clause
- **Trace back from** $c$ until marked nodes or new nodes
  - Learn clause if only marked nodes visited

---

[SB09]
Outline

Basic Definitions

DPLL Solvers

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

What Next in CDCL Solvers?
- 10000 runs, branching randomization on industrial instance

- Use **rapid randomized restarts** (search restarts)
Search Restarts II

- Restart search after a number of conflicts

![Diagram showing search restarts with cutoffs and a solution]
Resume de la Recherche des Départs

- Redonner la recherche après un certain nombre de conflits.
- Augmenter les cutoffs après chaque redépart.
  - Garantit la complétude.
  - Différents politiques existent (voir références).

La méthode fonctionne pour les instances SAT et UNSAT.

Pourquoi?
- Les clauses apprises sont efficients après chaque redépart.

Graphique illustrant la recherche des départs avec des cutoffs et une solution.
Search Restarts II

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

- Restart search after a number of conflicts
- Increase cutoff after each restart
  - Guarantees completeness
  - Different policies exist (see refs)
- Works for SAT & UNSAT instances. Why?
- Learned clauses effective after restart(s)
Data Structures Basics

- Each literal should access clauses containing /
  - Why?
Each literal \( l \) should access clauses containing \( l \)
- **Why?** Unit propagation
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
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, $L$
• 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: $O(n)$
Data Structures Basics

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

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

- Number of clause references equals number of literals, \( L \)
  - Clause learning can generate large clauses
    - Worst-case size: \( \mathcal{O}(n) \)
  - Worst-case number of literals: \( \mathcal{O}(m n) \)
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!
Each literal \( l \) should access clauses containing \( l \)

- **Why?** Unit propagation

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

Number of clause references **equals** number of literals, \( L \)

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

Clause learning to be effective requires a more efficient representation:
Data Structures Basics

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

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

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

- Clause learning to be effective requires a more efficient representation: **Watched Literals**
• 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

[Caption: Diagram showing states of a clause]

unresolved

unresolved

unit

satisfied

after backtracking to level 4
- Important states of a clause
- Associate 2 references with each clause
- Deciding unit requires traversing all literals

\[ \text{unresolved} \]

\[ \text{unit} \]

\[ \text{satisfied} \]

\[ \text{after backtracking to level 4} \]
Watched Literals

- Important states of a clause
- Associate 2 references with each clause
- Deciding unit requires traversing all literals
- References unchanged when backtracking
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

[e.g. MMZZM01]
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.
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.

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.

- **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.
• 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 I
Examples of SAT-Based Problem Solving II

- Decision problems in $\Sigma_2^P$
  - 2QBF
  - ...

- Function problems in $FP^{\Sigma_2^P}$
  - (Weighted) Quantified MaxSAT ($Q\text{MaxSAT}$) \cite{IJMS13}
  - Smallest MUS ($SMUS$) \cite{IJMS13}
  - ...

- Decision problems in PSPACE
  - QBF
  - ...

- ...

\cite{IJMS13}
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?
Encoding to CNF

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

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

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 $z = OP(x, y)$, represent in CNF $z \leftrightarrow OP(x, y)$
    - CNF formula for the circuit is the conjunction of CNF formula for each gate

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

\[
F_t = (\overline{r} \lor t) \land (\overline{s} \lor t) \land (r \lor s \lor \overline{t})
\]
Representing Boolean Formulas / Circuits II

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

\[
\begin{array}{ccc|c}
 a & b & c & F_c(a, b, c) \\
 0 & 0 & 0 & 0 \\
 0 & 0 & 1 & 1 \\
 0 & 1 & 0 & 0 \\
 0 & 1 & 1 & 1 \\
 1 & 0 & 0 & 0 \\
 1 & 0 & 1 & 1 \\
 1 & 1 & 0 & 1 \\
 1 & 1 & 1 & 0 \\
\end{array}
\]
- CNF formula for the circuit is the conjunction of the CNF formula for each gate
  - Can specify objectives with additional clauses

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

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

• 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: \( \sum_{j=1}^{n} x_j \bowtie k \), with \( \bowtie \in \{<, \leq, =, \geq, >\} \)

• 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: \( \sum_{j=1}^{n} x_j \bowtie k \), with \( \bowtie \in \{<, \leq, =, \geq, >\} \)

- **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: $\mathcal{O}(n^2)$; No auxiliary variables
  - Sequential counter
    - Clauses: $\mathcal{O}(n)$; Auxiliary variables: $\mathcal{O}(n)$
  - Bitwise encoding
    - Clauses: $\mathcal{O}(n \log n)$; Auxiliary variables: $\mathcal{O}(\log n)$
  - ...
Bitwise Encoding

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

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

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

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

<table>
<thead>
<tr>
<th>$j - 1$</th>
<th>$v_1v_0$</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 $v_0, \ldots, v_{r-1}$; $r = \lfloor \log n \rfloor$ (with $n > 1$)
  - If $x_j = 1$, then $v_0 \ldots v_{r-1} = b_0 \ldots b_{r-1}$, the binary encoding of $j - 1$
    \[
    x_j \rightarrow (v_0 = b_0) \land \ldots \land (v_{r-1} = b_{r-1}) \iff (\bar{x}_j \lor (v_0 = b_0) \land \ldots \land (v_{r-1} = b_{r-1}))
    \]
  - Clauses $(\bar{x}_j \lor (v_i \leftrightarrow b_i)) = (\bar{x}_j \lor l_i)$, $i = 0, \ldots, r - 1$, where
    - $l_i \equiv v_i$, if $b_i = 1$
    - $l_i \equiv \bar{v}_i$, otherwise

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

<table>
<thead>
<tr>
<th></th>
<th>$v_1 \lor \bar{v}_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_1$</td>
<td>00</td>
</tr>
<tr>
<td>$x_2$</td>
<td>01</td>
</tr>
<tr>
<td>$x_3$</td>
<td>10</td>
</tr>
</tbody>
</table>
Bitwise Encoding

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

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

<table>
<thead>
<tr>
<th></th>
<th>$j - 1$</th>
<th>$v_1 v_0$</th>
<th>$(\bar{x}_1 \lor \bar{v}_1) \land (\bar{x}_1 \lor v_0)$</th>
<th>$(\bar{x}_2 \lor \bar{v}_1) \land (\bar{x}_2 \lor v_0)$</th>
<th>$(\bar{x}_3 \lor v_1) \land (\bar{x}_3 \lor \bar{v}_0)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_1$</td>
<td>0</td>
<td>00</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$x_2$</td>
<td>1</td>
<td>01</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$x_3$</td>
<td>2</td>
<td>10</td>
<td></td>
<td></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 \[ \text{[W98]} \]
    - Clauses/Variables: \( O(n) \)
    - Does not guarantee arc-consistency
  - BDDs \[ \text{[ES06]} \]
    - Worst-case exponential number of clauses
  - Polynomial watchdog encoding \[ \text{[BBR09]} \]
    - 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 \[ \text{[ANORC11b]} \]
    - Clauses & aux variables: \( O(n^3 \log(a_{max})) \)
  - ...

Encoding PB Constraints with BDDs I

- Encode $3x_1 + 3x_2 + x_3 \leq 3$
- Construct BDD
  - E.g. analyze variables by decreasing coefficients
- Extract ITE-based circuit from BDD
Encoding PB Constraints with BDDs

- 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,F02,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

- 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 $\sum_{k=1}^{m_i} x_{i,k} \geq 1$

[Wo0]

• If the pair of assignments $x_i = v_i \land x_j = v_j$ is not allowed, add binary clause $(\bar{x}_{i,v_i} \lor \bar{x}_{j,v_j})$
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:
    \[
    \sum_{j \in N} a_{ij} l_j \geq b_i, \quad l_j \in \{x_j, \bar{x}_j\}, x_j \in \{0, 1\}, a_{ij}, b_i \in \mathbb{N}_0^+\]
Pseudo-Boolean 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):
  \[
  \begin{align*}
  \text{minimize} & \quad \sum_{j \in N} c_j \cdot x_j \\
  \text{subject to} & \quad \sum_{j \in N} a_{ij} l_j \geq b_i, \\
  & \quad l_j \in \{x_j, \bar{x}_j\}, x_j \in \{0, 1\}, a_{ij}, b_i, c_j \in \mathbb{N}_0^+ 
  \end{align*}
\]
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):**
  \[
  \text{minimize} \quad \sum_{j \in N} c_j \cdot x_j \\
  \text{subject to} \quad \sum_{j \in N} a_{ij} l_j \geq b_i, \\
  \quad l_j \in \{x_j, \bar{x}_j\}, x_j \in \{0, 1\}, a_{ij}, b_i, c_j \in \mathbb{N}_0^+ 
  \]

- **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 \( (C_i \_ \_a_i) \), 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 \( C_i \), add clause \( (\_a_i) \) to delete \( C_i \).

  - **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:** \( \text{SAT}(F) \)
  2. Compute model: \( (\text{st}, \mu) \).
  3. Compute unsatisfiable cores: \( (\text{st}, \mu, U) \).
  4. Compute proof traces/resolution proof: \( (\text{st}, \mu, T) \).
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*

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

- **Incremental vs. non-incremental SAT**
  - **Incremental SAT:**
    - Replace each clause \((C_i)\) with \((C_i \lor \overline{a_i})\), 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 \(C_i\)

- **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 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 \(C_i\)
  - **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:**
Practical Aspects of Using SAT Oracles

- **Incremental vs. non-incremental SAT** [ES03]
  - **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: $(\text{st}, \text{SAT}(F))$
2. Compute model: $(\text{st}, \mu, \text{SAT}(F))$
3. Compute unsatisfiable cores: $(\text{st}, \mu, U, \text{SAT}(F))$
4. Compute proof traces/resolution proof: $(\text{st}, \mu, T, \text{SAT}(F))$
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 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 \( C_i \)
    - 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 \((C_i \lor \bar{a}_i)\), 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: \((st, \mu) \leftarrow \text{SAT}(\mathcal{F})\)
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 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: \((st) \leftarrow \text{SAT}(\mathcal{F})\)
  2. Compute model: \((st, \mu) \leftarrow \text{SAT}(\mathcal{F})\)
  3. Compute unsatisfiable cores: \((st, \mu, \mathcal{U}) \leftarrow \text{SAT}(\mathcal{F})\)
  4. Compute proof traces/resolution proof: \((st, \mu, \mathcal{T}) \leftarrow \text{SAT}(\mathcal{F})\)
Outline

CNF Encodings

SAT Embeddings

**SAT Oracles**
- MUS Extraction
- MaxSAT
- 2QBF

What Next in SAT-Based Problem Solving?
### Defining MUSes

<table>
<thead>
<tr>
<th></th>
<th>$x_6 \lor x_2$</th>
<th>$\neg x_6 \lor x_2$</th>
<th>$\neg x_2 \lor x_1$</th>
<th>$\neg x_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\neg x_6 \lor x_8$</td>
<td>$x_6 \lor \neg x_8$</td>
<td>$x_2 \lor x_4$</td>
<td>$\neg x_4 \lor x_5$</td>
<td></td>
</tr>
<tr>
<td>$x_7 \lor x_5$</td>
<td>$\neg x_7 \lor x_5$</td>
<td>$\neg x_5 \lor x_3$</td>
<td>$\neg x_3$</td>
<td></td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th>$x_6 \lor x_2$</th>
<th>$\neg x_6 \lor x_2$</th>
<th>$x_2 \lor x_1$</th>
<th>$\neg x_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\neg x_6 \lor x_8$</td>
<td>$x_6 \lor \neg x_8$</td>
<td>$x_2 \lor x_4$</td>
<td>$\neg x_4 \lor x_5$</td>
</tr>
<tr>
<td>$x_7 \lor x_5$</td>
<td>$\neg x_7 \lor x_5$</td>
<td>$\neg x_5 \lor x_3$</td>
<td>$\neg x_3$</td>
</tr>
</tbody>
</table>

- 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
Defining MUSes

\[
\begin{align*}
    x_6 \lor x_2 & \quad \neg x_6 \lor x_2 & \quad \neg x_2 \lor x_1 & \quad \neg x_1 \\
    \neg x_6 \lor x_8 & \quad x_6 \lor \neg x_8 & \quad x_2 \lor x_4 & \quad \neg x_4 \lor x_5 \\
    x_7 \lor x_5 & \quad \neg x_7 \lor x_5 & \quad \neg x_5 \lor x_3 & \quad \neg x_3
\end{align*}
\]

- 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

$\mathcal{M} \leftarrow \mathcal{F}$ \hspace{1cm} // MUS over-approximation

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

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

$\mathcal{M} \leftarrow \mathcal{M} \setminus \{c\}$ \hspace{1cm} // If UNSAT($\mathcal{M} \setminus \{c\}$), then $c \notin \mathcal{M}$

return $\mathcal{M}$ \hspace{1cm} // Final $\mathcal{M}$ is MUS

end

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

// Final $\mathcal{M}$ is MUS

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

(¬x₁ ∨ x₂)
(¬x₃ ∨ x₂)
(x₁ ∨ x₂)
(¬x₃)
(¬x₂)

Hide clause (¬x₁ ∨ x₂)
\[
\begin{align*}
\text{(keep clause)} \\
\neg x_3 \lor x_2 \\
x_1 \lor x_2 \\
\neg x_3 \\
\neg x_2
\end{align*}
\]

\textit{SAT} instance $\rightarrow$ keep clause $\left( \neg x_1 \lor x_2 \right)$
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

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

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

(\neg x_1 \lor x_2)

(x_1 \lor x_2)

(\neg x_3)

(\neg x_2)

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

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

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

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

Hide clause (\neg x_3)
An Example

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

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

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

\((\neg x_3)\)

\((x_1 \lor x_2)\)

\((\neg x_3)\)

\((\neg x_2)\)

Hide clause (\(\neg x_2\))
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 (\(\neg x_2\))
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>$\mathcal{O}(m \times k)$</td>
<td>[SP88]</td>
</tr>
<tr>
<td>Deletion (Default)</td>
<td>$\mathcal{O}(m)$</td>
<td>[CD91,BDTW93]</td>
</tr>
<tr>
<td>QuickXplain</td>
<td>$\mathcal{O}(k \times (1 + \log \frac{m}{k}))$</td>
<td>[J01,J04]</td>
</tr>
<tr>
<td>Dichotomic</td>
<td>$\mathcal{O}(k \times \log m)$</td>
<td>[HLSB06]</td>
</tr>
<tr>
<td>Insertion with Relaxation Variables</td>
<td>$\mathcal{O}(m)$</td>
<td>[MSL11]</td>
</tr>
<tr>
<td>Deletion with Model Rotation</td>
<td>$\mathcal{O}(m)$</td>
<td>[BLMS12,MSL11]</td>
</tr>
<tr>
<td>Progression</td>
<td>$\mathcal{O}(k \times \log(1 + \frac{m}{k}))$</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?
Defining Maximum Satisfiability

- 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. $\sum_{c_i \in \varphi} w_i + 1$; 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
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 \( \sum w_i r_i \leq UB_0 - 1 \)
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
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
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

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

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

\[ x_7 \lor x_5 \quad \neg x_7 \lor x_5 \quad \neg x_5 \lor x_3 \quad \neg x_3 \]
Fu&Malik’s Core-Guided Approach

Formula is **UNSAT**; **OPT ≤ |φ| - 1**; Get unsat core
Fu&Malik’s Core-Guided Approach

\[
\begin{align*}
&x_6 \lor x_2 \\
&\neg x_6 \lor x_2 \\
&\neg x_6 \lor x_8 \\
&x_6 \lor \neg x_8 \\
&x_7 \lor x_5 \\
&\neg x_7 \lor x_5 \\
&\neg x_5 \lor x_1 \lor r_1 \\
&\neg x_2 \lor x_1 \lor r_1 \\
&\neg x_1 \lor r_2 \\
&\neg x_4 \lor x_5 \lor r_4 \\
&\neg x_2 \lor x_4 \lor r_3 \\
&\neg x_4 \lor x_5 \lor r_4 \\
&\neg x_5 \lor x_3 \lor r_5 \\
&\neg x_5 \lor x_3 \lor r_5 \\
&\neg x_3 \lor r_6 \\
&\sum_{i=1}^{6} r_i \leq 1
\end{align*}
\]

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

\[
\begin{align*}
& x_6 \lor x_2 & \neg x_6 \lor x_2 & \neg x_2 \lor x_1 \lor r_1 & \neg x_1 \lor r_2 \\
& \neg x_6 \lor x_8 & x_6 \lor \neg x_8 & x_2 \lor x_4 \lor r_3 & \neg x_4 \lor x_5 \lor r_4 \\
& x_7 \lor x_5 & \neg x_7 \lor x_5 & \neg x_5 \lor x_3 \lor r_5 & \neg x_3 \lor r_6 \\
& \sum_{i=1}^{6} r_i \leq 1
\end{align*}
\]

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

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

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

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

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

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

MaxSAT solution is \(|\varphi| - I = 12 - 2 = 10|\)
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)

[FM06]
**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)

- 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**

References:
- [MHLPMS13]
- [e.g. LBP10]
- [e.g. FM06]
- [MSP07]
- [MSP08]
- [ABL09a, ABL09b, ABL10, ABGL13]
- [HMMS11, MHMS12]
- [DB11, DB13a, DB13b]
### 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 $\forall Y. \phi[X/\tau]$?
Problem Statement

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

<table>
<thead>
<tr>
<th>$\xi$</th>
<th>1</th>
<th>0</th>
<th>...</th>
<th>0</th>
<th>1</th>
<th>...</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></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>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Looking at Assignments

<table>
<thead>
<tr>
<th></th>
<th>$\mu$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X$</td>
<td></td>
</tr>
<tr>
<td>$\xi$</td>
<td>1</td>
</tr>
<tr>
<td>$\tau$</td>
<td>1</td>
</tr>
</tbody>
</table>
# Looking at Assignments

$$
\begin{array}{c|c}
\xi & 1 & 0 & \ldots & 0 & 1 & \ldots & 1 \\
\tau & 1 & 1 & \ldots & 1 & 1 & \ldots & 1 \\
\end{array}
$$

$$
\phi[Y/\mu]
$$
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

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

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

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

Expansion:

$$(x_1 \leftrightarrow 0) \land (x_2 \leftrightarrow 0) \land (\overline{x}_1 \lor \overline{x}_2)$$
$$\land (x_1 \leftrightarrow 0) \land (x_2 \leftrightarrow 1) \land (\overline{x}_1 \lor \overline{x}_2)$$
$$\land (x_1 \leftrightarrow 1) \land (x_2 \leftrightarrow 0) \land (\overline{x}_1 \lor \overline{x}_2)$$
$$\land (x_1 \leftrightarrow 1) \land (x_2 \leftrightarrow 1) \land (\overline{x}_1 \lor \overline{x}_2)$$
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 $\omega \subseteq B^{\lvert Y \rvert}$

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

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

$$\bigwedge_{\mu \in B^{\lvert Y \rvert}} \phi[Y/\mu] \Rightarrow \bigwedge_{\mu \in \omega} \phi[Y/\mu]$$

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 $\omega \subseteq B^{|Y|}$

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

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

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

- 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

\((\text{outc}_1, \tau) \leftarrow \text{SAT}(\wedge_{\mu \in \omega} \phi[Y/\mu]);\) \hspace{1cm} // find a candidate

if \(\text{outc}_1 = \text{false}\) then

\(\text{return } (\text{false},-);\) \hspace{1cm} // no candidate found

end

if “\(\tau\) is a solution”; \hspace{1cm} // solution check
then
\(\text{return } (\text{true}, \tau)\)
else
“Grow \(\omega\)”;
end

end
CEGAR Loop

**input**: \( \exists X \forall Y. \phi \)

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

\((\text{false}, -)\) otherwise

\( \omega \leftarrow \emptyset; \)

**while** true **do**

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

**if** \( \text{outc}_1 = \text{false} \) **then**

| **return** (false, -);  // no candidate found

**end**

**if** “\( \tau \) is a solution”;  // solution check

**then**

| **return** (true, \( \tau \))

**else**

| “Grow \( \omega \)”;

**end**

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

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

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

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

If $\text{SAT}(\lnot \phi[X/\tau])$ 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: \( y_1 = 0, y_2 = 0 \)
  \( y_1 = 0, y_2 = 1 \)
  \( y_1 = 1, y_2 = 0 \)
Refinement
Refinement
Refinement
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

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

**while** true **do**

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

**if** outc$_1 = false$ **then**

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

**end**

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

**if** outc$_2 = false$ **then**

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

**end**

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

**end**
... is a CEGAR-based algorithm for 2QBF [JMS11]
AReQS — Conclusions

- ... is a CEGAR-based algorithm for 2QBF
- ... uses SAT solver as an oracle
AReQS — Conclusions

- ... is a CEGAR-based algorithm for 2QBF
- ... 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)
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
- $FP^{NP}[\log n]$, $FP^{NP}$, ...
- 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>Author(s)</th>
<th>Title</th>
<th>Conference/Proceedings</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>BMS00</td>
<td>L. Baptista, J. Marques-Silva</td>
<td>Using Randomization and Learning to Solve Hard Real-World Instances of Satisfiability.</td>
<td>CP</td>
<td>2000: 489-494</td>
</tr>
<tr>
<td>Reference</td>
<td>Authors</td>
<td>Title</td>
<td>Conference/Journal</td>
<td>Pages</td>
</tr>
<tr>
<td>-----------</td>
<td>---------</td>
<td>-------</td>
<td>--------------------</td>
<td>-------</td>
</tr>
<tr>
<td>GN02</td>
<td>E. Goldberg, Y. Novikov</td>
<td>BerkMin: A Fast and Robust Sat-Solver.</td>
<td>DATE 2002</td>
<td>142-149</td>
</tr>
<tr>
<td>ES03</td>
<td>N. Een, Niklas Sorensson</td>
<td>An Extensible SAT-solver.</td>
<td>SAT 2003</td>
<td>502-518</td>
</tr>
<tr>
<td>PD07</td>
<td>K. Pipatsrisawat, A. Darwiche</td>
<td>A Lightweight Component Caching Scheme for Satisfiability Solvers.</td>
<td>SAT 2007</td>
<td>294-299</td>
</tr>
<tr>
<td>H07</td>
<td>J. Huang</td>
<td>The Effect of Restarts on the Efficiency of Clause Learning.</td>
<td>IJCAI 2007</td>
<td>2318-2323</td>
</tr>
<tr>
<td>B08</td>
<td>A. Biere</td>
<td>PicoSAT Essentials.</td>
<td>JSAT 4(2-4)</td>
<td>75-97 (2008)</td>
</tr>
<tr>
<td>SB09</td>
<td>N. Sorensson, A. Biere</td>
<td>Minimizing Learned Clauses.</td>
<td>SAT 2009</td>
<td>237-243</td>
</tr>
<tr>
<td>VG09</td>
<td>A. Van Gelder</td>
<td>Improved Conflict-Clause Minimization Leads to Improved Propositional Proof Traces.</td>
<td>SAT 2009</td>
<td>141-146</td>
</tr>
<tr>
<td>AS09</td>
<td>G. Audemard, L. Simon</td>
<td>Predicting Learnt Clauses Quality in Modern SAT Solvers.</td>
<td>IJCAI 2009</td>
<td>399-404</td>
</tr>
<tr>
<td>SSS12</td>
<td>A. Sabharwal, H. Samulowitz, M. Sellmann</td>
<td>Learning Back-Clauses in SAT.</td>
<td>SAT 2012</td>
<td>498-499</td>
</tr>
<tr>
<td>Reference</td>
<td>Author(s)</td>
<td>Title</td>
<td>Conference/Journal</td>
<td>Volume/Issue/Number</td>
</tr>
<tr>
<td>-----------</td>
<td>-----------</td>
<td>-------</td>
<td>--------------------</td>
<td>---------------------</td>
</tr>
<tr>
<td>B68</td>
<td>K. Batcher</td>
<td>Sorting Networks and Their Applications.</td>
<td>AFIPS Spring Joint Computing Conference 1968</td>
<td>307-314</td>
</tr>
<tr>
<td>dK89</td>
<td>Johan de Kleer</td>
<td>A Comparison of ATMS and CSP Techniques.</td>
<td>IJCAI 1989</td>
<td>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</td>
<td>180-184</td>
</tr>
<tr>
<td>W00</td>
<td>T. Walsh</td>
<td>SAT v CSP.</td>
<td>CP 2000</td>
<td>441-456</td>
</tr>
<tr>
<td>Reference</td>
<td>Authors</td>
<td>Title</td>
<td>Conference/Journal</td>
<td>Pages</td>
</tr>
<tr>
<td>-----------</td>
<td>---------</td>
<td>-------</td>
<td>--------------------</td>
<td>-------</td>
</tr>
<tr>
<td>FP01</td>
<td>A. Frisch, T. Peugniez</td>
<td>Solving Non-Boolean Satisfiability Problems with Stochastic Local Search</td>
<td>IJCAI 2001</td>
<td>282-290</td>
</tr>
<tr>
<td>FS02</td>
<td>T. Fahle, M. Sellmann</td>
<td>Cost Based Filtering for the Constrained Knapsack Problem</td>
<td>Annals OR 115(1-4)</td>
<td>73-93 (2002)</td>
</tr>
<tr>
<td>S03</td>
<td>M. Sellmann</td>
<td>Approximated Consistency for Knapsack Constraints</td>
<td>CP 2003</td>
<td>679-693</td>
</tr>
<tr>
<td>S05</td>
<td>C. Sinz</td>
<td>Towards an Optimal CNF Encoding of Boolean Cardinality Constraints</td>
<td>CP 2005</td>
<td>827-831</td>
</tr>
<tr>
<td>G07</td>
<td>M. Gavanelli</td>
<td>The Log-Support Encoding of CSP into SAT</td>
<td>CP 2007</td>
<td>815-822</td>
</tr>
<tr>
<td>P07</td>
<td>S. Prestwich</td>
<td>Variable Dependency in Local Search: Prevention Is Better Than Cure</td>
<td>SAT 2007</td>
<td>107-120</td>
</tr>
<tr>
<td>Reference</td>
<td>Authors</td>
<td>Title</td>
<td>Conference/Journal</td>
<td>Pages</td>
</tr>
<tr>
<td>-----------</td>
<td>---------</td>
<td>-------</td>
<td>--------------------</td>
<td>-------</td>
</tr>
<tr>
<td>BBR09</td>
<td>O. Bailleux, Y. Boufkhad, O. Roussel</td>
<td>New Encodings of Pseudo-Boolean Constraints into CNF</td>
<td>SAT 2009</td>
<td>181-194</td>
</tr>
<tr>
<td>TTKB09</td>
<td>Naoyuki Tamura, Akiko Taga, Satoshi Kitagawa, Mutsunori Banbara</td>
<td>Compiling finite linear CSP into SAT</td>
<td>Constraints 14(2)</td>
<td>254-272</td>
</tr>
<tr>
<td>CZI10</td>
<td>M. Codish, M. Zason-Ivry</td>
<td>Pairwise Cardinality Networks</td>
<td>LPAR (Dakar)</td>
<td>154-172</td>
</tr>
<tr>
<td>ANORC11a</td>
<td>R. Asin, R. Nieuwenhuis, A. Oliveras, E. RodrÃ­guez-Carbonell</td>
<td>Cardinality Networks: a theoretical and empirical study</td>
<td>Constraints 16(2)</td>
<td>195-221</td>
</tr>
<tr>
<td>Reference</td>
<td>Author(s)</td>
<td>Title</td>
<td>Source</td>
<td></td>
</tr>
<tr>
<td>-----------</td>
<td>-----------</td>
<td>-------</td>
<td>--------</td>
<td></td>
</tr>
<tr>
<td>S07</td>
<td>R. Sebastiani</td>
<td>Lazy Satisfiability Modulo Theories</td>
<td>JSAT 3(3-4): 141-224 (2007)</td>
<td></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: 825-885</td>
<td></td>
</tr>
</tbody>
</table>
References – MUSes I


<table>
<thead>
<tr>
<th>Reference</th>
<th>Authors/Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reference</td>
<td>Authors</td>
</tr>
<tr>
<td>------------------</td>
<td>----------------------------------------------</td>
</tr>
<tr>
<td>FM06</td>
<td>Z. Fu, S. Malik</td>
</tr>
<tr>
<td>MSP08</td>
<td>J. Marques-Silva, Jordi Planes</td>
</tr>
<tr>
<td>ABL09a</td>
<td>C. Ansotegui, M. Bonet, J. Levy</td>
</tr>
<tr>
<td>ABL10</td>
<td>C. Ansotegui, M. Bonet, J. Levy</td>
</tr>
<tr>
<td>Reference</td>
<td>Authors</td>
</tr>
<tr>
<td>-----------</td>
<td>---------</td>
</tr>
<tr>
<td>DB11</td>
<td>J. Davies, F. Bacchus</td>
</tr>
<tr>
<td>MHMS12</td>
<td>A. Morgado, F. Heras, J. Marques-Silva</td>
</tr>
<tr>
<td>ABGL12</td>
<td>C. Ansotegui, M. Bonet, J. Gabas, J. Levy</td>
</tr>
<tr>
<td>DB13a</td>
<td>J. Davies, F. Bacchus</td>
</tr>
<tr>
<td>ABGL13</td>
<td>C. Ansotegui, M. Bonet, J. Gabas and J. Levy</td>
</tr>
<tr>
<td>DB13b</td>
<td>J. Davies and F. Bacchus</td>
</tr>
<tr>
<td>Reference</td>
<td>Title</td>
</tr>
<tr>
<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>
</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>
</tr>
<tr>
<td>KJMSC13</td>
<td>W. Klieber, M. Janota, J. Marques-Silva, E. Clarke: Solving QBF with Free Variables. CP 2013</td>
</tr>
<tr>
<td>Code</td>
<td>Author(s)</td>
</tr>
<tr>
<td>------</td>
<td>-----------</td>
</tr>
<tr>
<td>C71</td>
<td>S. Cook</td>
</tr>
<tr>
<td>ZM03</td>
<td>L. Zhang, S. Malik</td>
</tr>
<tr>
<td>SP04</td>
<td>S. Subbarayan, D. Pradhan</td>
</tr>
<tr>
<td>EB05</td>
<td>N. Een, A. Biere</td>
</tr>
<tr>
<td>HJB11</td>
<td>M. Heule, M. Jarvisalo, A. Biere</td>
</tr>
<tr>
<td>JHB12</td>
<td>M. Jarvisalo, M. Heule, A. Biere</td>
</tr>
<tr>
<td>LB13</td>
<td>J.-M. Lagniez, A. Biere</td>
</tr>
</tbody>
</table>