Lazy Clause Generation: Combining the best of SAT and CP (and MIP?) solving

Peter J. Stuckey
with help from Timo Berthold, Geoffrey Chu, Michael Codish, Thibaut Feydy, Graeme Gange, Olga Ohrimenko, Andreas Schutt, and Mark Wallace

June 2010
Repeatedly run *propagators*

- Propagators change variable domains by:
  - removing values
  - changing upper and lower bounds
  - fixing to a value

- Run until fixpoint.

**KEY INSIGHT:**

- Changes in domains are really the fixing of *Boolean variables* representing domains.
- Propagation is just the generation of clauses on these variables.
- FD solving is just SAT solving: conflict analysis for **FREE!**
1 Finite Domain Propagation
   - FD Example

2 SAT Solving
   - SAT Example

3 Lazy Clause Generation
   - Original Lazy Clause Generation
   - Lazier Clause Generation
   - Global Constraints
   - Search

4 Related Work

5 Conclusion
Outline

1 Finite Domain Propagation
   - FD Example

2 SAT Solving
   - SAT Example

3 Lazy Clause Generation
   - Original Lazy Clause Generation
   - Lazier Clause Generation
   - Global Constraints
   - Search

4 Related Work

5 Conclusion
Terminology

- **domain** $D$ maps variable $x$ to set of possible values $D(x)$
- **propagator** $f_c : D \mapsto D$ for constraint $c$
  - monotonic decreasing function
  - removes values from the domain which cannot be part of a solution.
- **Problem** set of propagators $F$ and initial domain $D_0$
- **propagation solver** $solv(F, D) = D'$ where $D'$ is the greatest mutual fixpoint of all $f \in F$.
- **FD solving** interleaves propagation with search: (for simplicity binary)
  - Add new search constraint $c$. $D' = solv(F \cup \{f_c\}, D)$
  - On failure add backtrack and add $\neg c$. $D' = solv(F \cup \{f_{\neg c}\}, D)$
  - Repeat until all variables fixed
Outline

1 Finite Domain Propagation
   - FD Example

2 SAT Solving
   - SAT Example

3 Lazy Clause Generation
   - Original Lazy Clause Generation
   - Lazier Clause Generation
   - Global Constraints
   - Search

4 Related Work

5 Conclusion
Finite Domain Propagation Example

Consider the problem with:

**Domain** \( D_0 \):

\[
D_0(x_1) = D_0(x_2) = D_0(x_3) = D_0(x_4) = D_0(x_5) = [1..4]
\]

**\( F \) propagators for:**

\( x_2 \leq x_5, \text{alldifferent}([x_1, x_2, x_3, x_4]), x_1 + x_2 + x_3 + x_4 \leq 9. \)

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( x_1 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( x_2 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( x_3 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( x_4 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( x_5 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Finite Domain Propagation Example

Consider the problem with:

**Domain \( D_0 \):**

\[
D_0(x_1) = D_0(x_2) = D_0(x_3) = D_0(x_4) = D_0(x_5) = [1..4]
\]

**\( F \) propagators for:**

\[
x_2 \leq x_5, \text{alldifferent}([x_1, x_2, x_3, x_4]), x_1 + x_2 + x_3 + x_4 \leq 9.
\]

<table>
<thead>
<tr>
<th>( x_1 = 1 )</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( x_1 )</td>
<td>1</td>
</tr>
<tr>
<td>( x_2 )</td>
<td>[1..4]</td>
</tr>
<tr>
<td>( x_3 )</td>
<td>[1..4]</td>
</tr>
<tr>
<td>( x_4 )</td>
<td>[1..4]</td>
</tr>
<tr>
<td>( x_5 )</td>
<td>[1..4]</td>
</tr>
</tbody>
</table>
Finite Domain Propagation Example

Consider the problem with:

**Domain** $D_0$:

\[ D_0(x_1) = D_0(x_2) = D_0(x_3) = D_0(x_4) = D_0(x_5) = [1..4] \]

**F propagators for**:

\[ x_2 \leq x_5, \text{alldifferent}([x_1, x_2, x_3, x_4]), x_1 + x_2 + x_3 + x_4 \leq 9. \]

<table>
<thead>
<tr>
<th>$x_1 = 1$</th>
<th>alldiff</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_1$</td>
<td>1</td>
</tr>
<tr>
<td>$x_2$</td>
<td>[1..4]</td>
</tr>
<tr>
<td>$x_3$</td>
<td>[1..4]</td>
</tr>
<tr>
<td>$x_4$</td>
<td>[1..4]</td>
</tr>
<tr>
<td>$x_5$</td>
<td>[1..4]</td>
</tr>
</tbody>
</table>
Finite Domain Propagation Example

Consider the problem with:

**Domain** $D_0$:

$$D_0(x_1) = D_0(x_2) = D_0(x_3) = D_0(x_4) = D_0(x_5) = [1..4]$$

**$F$ propagators for**:

$$x_2 \leq x_5, \text{alldifferent}([x_1, x_2, x_3, x_4]), x_1 + x_2 + x_3 + x_4 \leq 9.$$ 

<table>
<thead>
<tr>
<th></th>
<th>$x_1 = 1$</th>
<th>alldiff</th>
<th>$x_2 \leq x_5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_1$</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$x_2$</td>
<td>[1..4]</td>
<td>[2..4]</td>
<td>[2..4]</td>
</tr>
<tr>
<td>$x_3$</td>
<td>[1..4]</td>
<td>[2..4]</td>
<td>[2..4]</td>
</tr>
<tr>
<td>$x_4$</td>
<td>[1..4]</td>
<td>[2..4]</td>
<td>[2..4]</td>
</tr>
<tr>
<td>$x_5$</td>
<td>[1..4]</td>
<td>[1..4]</td>
<td>[2..4]</td>
</tr>
</tbody>
</table>

$D_1$
Finite Domain Propagation Example

Consider the problem with:

**Domain** $D_0$:

$$D_0(x_1) = D_0(x_2) = D_0(x_3) = D_0(x_4) = D_0(x_5) = [1..4]$$

**F propagators** for:

$$x_2 \leq x_5, \text{alldifferent}([x_1, x_2, x_3, x_4]), x_1 + x_2 + x_3 + x_4 \leq 9.$$

<table>
<thead>
<tr>
<th>(x_1) = 1</th>
<th>\text{alldiff}</th>
<th>(x_2 \leq x_5)</th>
<th>(x_5 \leq 2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x_1)</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>(x_2)</td>
<td>[1..4]</td>
<td>[2..4]</td>
<td>[2..4]</td>
</tr>
<tr>
<td>(x_3)</td>
<td>[1..4]</td>
<td>[2..4]</td>
<td>[2..4]</td>
</tr>
<tr>
<td>(x_4)</td>
<td>[1..4]</td>
<td>[2..4]</td>
<td>[2..4]</td>
</tr>
<tr>
<td>(x_5)</td>
<td>[1..4]</td>
<td>[1..4]</td>
<td>[2..4]</td>
</tr>
</tbody>
</table>

\(D_1\)
Consider the problem with:

**Domain** $D_0$:

$$D_0(x_1) = D_0(x_2) = D_0(x_3) = D_0(x_4) = D_0(x_5) = [1..4]$$

**F propagators for:**

$$x_2 \leq x_5, \text{alldifferent}([x_1, x_2, x_3, x_4]), x_1 + x_2 + x_3 + x_4 \leq 9.$$

<table>
<thead>
<tr>
<th></th>
<th>$x_1 = 1$</th>
<th>alldiff</th>
<th>$x_2 \leq x_5$</th>
<th>$x_5 \leq 2$</th>
<th>$x_2 \leq x_5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_1$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$x_2$</td>
<td>[1..4]</td>
<td>[2..4]</td>
<td>[2..4]</td>
<td>[2..4]</td>
<td>2</td>
</tr>
<tr>
<td>$x_3$</td>
<td>[1..4]</td>
<td>[2..4]</td>
<td>[2..4]</td>
<td>[2..4]</td>
<td>[2..4]</td>
</tr>
<tr>
<td>$x_4$</td>
<td>[1..4]</td>
<td>[2..4]</td>
<td>[2..4]</td>
<td>[2..4]</td>
<td>[2..4]</td>
</tr>
<tr>
<td>$x_5$</td>
<td>[1..4]</td>
<td>[1..4]</td>
<td>[2..4]</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

$D_1$
Finite Domain Propagation Example

Consider the problem with:

**Domain** $D_0$:
\[
D_0(x_1) = D_0(x_2) = D_0(x_3) = D_0(x_4) = D_0(x_5) = [1..4]
\]

**F propagators for:**
\[
x_2 \leq x_5, \text{alldifferent}([x_1, x_2, x_3, x_4]), x_1 + x_2 + x_3 + x_4 \leq 9.
\]

<table>
<thead>
<tr>
<th></th>
<th>$x_1 = 1$</th>
<th>alldiff</th>
<th>$x_2 \leq x_5$</th>
<th>$x_5 \leq 2$</th>
<th>alldiff</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_1$</td>
<td>1</td>
<td></td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>$x_2$</td>
<td>[1..4]</td>
<td>[2..4]</td>
<td>[2..4]</td>
<td>[2..4]</td>
<td>2</td>
</tr>
<tr>
<td>$x_3$</td>
<td>[1..4]</td>
<td>[2..4]</td>
<td>[2..4]</td>
<td>[2..4]</td>
<td>[3..4]</td>
</tr>
<tr>
<td>$x_4$</td>
<td>[1..4]</td>
<td>[2..4]</td>
<td>[2..4]</td>
<td>[2..4]</td>
<td>[3..4]</td>
</tr>
<tr>
<td>$x_5$</td>
<td>[1..4]</td>
<td>[1..4]</td>
<td>[2..4]</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

$D_1$
Consider the problem with:

**Domain** $D_0$:

$$D_0(x_1) = D_0(x_2) = D_0(x_3) = D_0(x_4) = D_0(x_5) = [1..4]$$

**$F$ propagators for**:

$$x_2 \leq x_5, \text{alldifferent}([x_1, x_2, x_3, x_4]), x_1 + x_2 + x_3 + x_4 \leq 9.$$
Finite Domain Propagation Example

Consider the problem with:

Domain $D_0$:

\[
D_0(x_1) = D_0(x_2) = D_0(x_3) = D_0(x_4) = D_0(x_5) = [1..4]
\]

$F$ propagators for:

\[x_2 \leq x_5, \text{alldifferent}([x_1, x_2, x_3, x_4]), x_1 + x_2 + x_3 + x_4 \leq 9.\]

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2 \leq x_5$</th>
<th>$x_5 \leq 2$</th>
<th>$x_2 \leq x_5$</th>
<th>$\text{alldiff}$</th>
<th>$\sum \leq 9$</th>
<th>$\text{alldiff}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>[1..4]</td>
<td>[2..4]</td>
<td>[2..4]</td>
<td>[2..4]</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>[1..4]</td>
<td>[2..4]</td>
<td>[2..4]</td>
<td>[2..4]</td>
<td>[3..4]</td>
<td>3</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>[1..4]</td>
<td>[2..4]</td>
<td>[2..4]</td>
<td>[2..4]</td>
<td>[3..4]</td>
<td>3</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>[1..4]</td>
<td>[1..4]</td>
<td>[2..4]</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

$D_1$

Backtrack

\[\text{fail}\]
Finite Domain Propagation Example

<table>
<thead>
<tr>
<th>$x_1 = 1$</th>
<th>$alldiff$</th>
<th>$x_2 \leq x_5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_1$</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$x_2$</td>
<td>[1..4]</td>
<td>[2..4]</td>
</tr>
<tr>
<td>$x_3$</td>
<td>[1..4]</td>
<td>[2..4]</td>
</tr>
<tr>
<td>$x_4$</td>
<td>[1..4]</td>
<td>[2..4]</td>
</tr>
<tr>
<td>$x_5$</td>
<td>[1..4]</td>
<td>[2..4]</td>
</tr>
</tbody>
</table>

$D_1$
Finite Domain Propagation Example

<table>
<thead>
<tr>
<th></th>
<th>$x_1 = 1$</th>
<th>alldiff</th>
<th>$x_2 \leq x_5$</th>
<th>$x_5 &gt; 2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_1$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$x_2$</td>
<td>[1..4]</td>
<td>[2..4]</td>
<td>[2..4]</td>
<td>[2..4]</td>
</tr>
<tr>
<td>$x_3$</td>
<td>[1..4]</td>
<td>[2..4]</td>
<td>[2..4]</td>
<td>[2..4]</td>
</tr>
<tr>
<td>$x_4$</td>
<td>[1..4]</td>
<td>[2..4]</td>
<td>[2..4]</td>
<td>[2..4]</td>
</tr>
<tr>
<td>$x_5$</td>
<td>[1..4]</td>
<td>[1..4]</td>
<td>[2..4]</td>
<td>[3..4]</td>
</tr>
</tbody>
</table>

$D_1$ $D_2$
Finite Domain Propagation Example

<table>
<thead>
<tr>
<th></th>
<th>$x_1 = 1$</th>
<th>alldiff</th>
<th>$x_2 \leq x_5$</th>
<th>$x_5 &gt; 2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_1$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$x_2$</td>
<td>[1..4]</td>
<td>2..4</td>
<td>2..4</td>
<td>2..4</td>
</tr>
<tr>
<td>$x_3$</td>
<td>[1..4]</td>
<td>2..4</td>
<td>2..4</td>
<td>2..4</td>
</tr>
<tr>
<td>$x_4$</td>
<td>[1..4]</td>
<td>2..4</td>
<td>2..4</td>
<td>2..4</td>
</tr>
<tr>
<td>$x_5$</td>
<td>[1..4]</td>
<td>1..4</td>
<td>2..4</td>
<td>3..4</td>
</tr>
</tbody>
</table>

$D_1$ $D_2$

$D_0$

$D_1$

$D_2$

fail
Strengths and Weaknesses of FD solving

- **Strengths**
  - high level modelling
  - specialized global propagators
  - programmable search

- **Weaknesses**
  - Search often needs programming (weak autonomous search)
  - Optimization by repeated satisfaction search
Outline

1. Finite Domain Propagation
   - FD Example

2. SAT Solving
   - SAT Example

3. Lazy Clause Generation
   - Original Lazy Clause Generation
   - Lazier Clause Generation
   - Global Constraints
   - Search

4. Related Work

5. Conclusion
Terminology

- **literal** $l = b$ or $l = \neg b$ where $b$ is a Boolean
- **clause** $l_1 \lor \cdots \lor l_n$ (or set of literals $\{l_1, \ldots, l_n\}$) also
  \[ \neg l_1 \land \cdots \land \neg l_{n-1} \rightarrow l_n \]
- **CNF** set of clauses $C$
- **assignment** $A$ is a set of literals $\{b, \neg b\} \subseteq A$
- **unit propagation** $up(C, A) = A'$
  - foreach clause $l_1 \lor \cdots \lor l_{n-1} \lor l_n$ where $\{\neg l_1, \ldots, \neg l_{n-1}\} \subseteq A$ add $l_n$ to $A$.
  - continue to fixpoint
- **SAT solving**
  - Choose a literal $l$: $A' := up(C, A \cup \{l\})$
  - On failure determine a nogood $c \subseteq A$ and add it to $C$, backjump
  - Repeat until all variables fixed
Outline

1 Finite Domain Propagation
   - FD Example

2 SAT Solving
   - SAT Example

3 Lazy Clause Generation
   - Original Lazy Clause Generation
   - Lazier Clause Generation
   - Global Constraints
   - Search

4 Related Work

5 Conclusion
Decision \( e_{11} \)

Resolving clauses: \( \neg e_{11} \lor \neg e_{21}, \neg e_{11} \lor \neg e_{31}, \neg e_{11} \lor \neg e_{41} \).
Decision $e_{11}$

Resolving clauses: $e_{21} \lor \neg b_{21}$, $e_{31} \lor \neg b_{31}$, $e_{41} \lor \neg b_{41}$. 


Decision $e_{11}$
Resolving clause: $b_{21} \lor \neg b_{51}$
Unit fixpoint
New Decision $b_{52}$

Resolving clauses: $b_{51} \lor \neg b_{52} \lor e_{52}, \neg b_{52} \lor b_{22}$
Decision $b_{52}$

Resolving clauses many

Conflict detected!
Initial nogood ($\neg e_{33} \vee \neg e_{43}$)

$$e_{33} \land e_{43} \rightarrow false$$
Resolving $b_{42} \lor \neg b_{43} \lor e_{43}$ gives

$$\neg b_{42} \land b_{43} \land e_{33} \rightarrow false$$
Resolving \( b_{32} \lor \neg b_{33} \lor e_{33} \) gives
\[
\neg b_{32} \land \neg b_{42} \land b_{33} \land b_{43} \rightarrow false
\]
Resolving $b_{21} \lor b_{42} \lor b_{33}$ and $b_{21} \lor b_{32} \lor b_{43}$ gives

$$\neg b_{21} \land \neg b_{32} \land \neg b_{42} \rightarrow false$$
Resolving $b_{31} \lor e_{32} \lor \neg b_{32}$ and $b_{41} \lor e_{42} \lor \neg b_{42}$ gives

$$\neg b_{21} \land \neg b_{31} \land \neg b_{41} \land \neg e_{32} \land \neg e_{42} \rightarrow \text{false}$$
Resolving $\neg e_{22} \lor \neg e_{32}$ and $\neg e_{22} \lor \neg e_{42}$ gives

$\neg b_{21} \land \neg b_{31} \land \neg b_{41} \land e_{22} \rightarrow false$

The 1UIP nogood! $b_{21} \lor b_{31} \lor b_{41} \lor \neg e_{22}$
Backjump
Apply nogood: $b_{21} \lor b_{31} \lor b_{41} \lor \neg e_{22}$
Continue to unit fixpoint
SAT Implication Graph

Continue to unit fixpoint

Resolving clauses $b_{21} \lor \neg b_{22} \lor e_{22}, \neg b_{52} \lor b_{22}$

Unit fixpoint
SAT engineering

- Cornerstones of modern SAT solvers
  - Watched literals: efficient implementation of unit propagation
  - 1UIP nogoods: record effective nogoods (efficiently)
  - Activity-based search: concentrate on variables involved in recent failures
  - Re starts

- Other features
  - Deep backjumping
  - Activity based forgetting of nogoods
  - Retry last used value for a variable
Strengths and Weaknesses of SAT solving

**Strengths**
- Learning avoids repeating the same subsearch
- Can deal with (low) millions of variables and clauses
- Strong autonomous search

**Weaknesses**
- Optimization by repeated satisfaction search
- Have to model entirely in clauses/Booleans (can definitely blow the limits above)
Outline

1 Finite Domain Propagation
   - FD Example

2 SAT Solving
   - SAT Example

3 Lazy Clause Generation
   - Original Lazy Clause Generation
   - Lazier Clause Generation
   - Global Constraints
   - Search

4 Related Work

5 Conclusion
Outline

1. Finite Domain Propagation
   - FD Example

2. SAT Solving
   - SAT Example

3. Lazy Clause Generation
   - Original Lazy Clause Generation
   - Lazier Clause Generation
   - Global Constraints
   - Search

4. Related Work

5. Conclusion
Representing Integer and Set Variables

- Integer variable $x$: represented using Booleans
  - $[x = d], d \in [l..u] = D_0(x)$,
  - $[x \leq d], l \leq d < u$.

- Clauses to maintain consistency: $\text{DOM}$
  - $[x \leq d] \rightarrow [x \leq d + 1] \quad l \leq d < u - 1$
  - $[x = d] \leftrightarrow [x \leq d] \land \neg[x \leq d - 1] \quad l < d \leq u$

- **Unary arithmetic** representation (linear in size)

- **One to one correspondence** domains $D$ and assignment $A$ unit fixpoints of $\text{DOM} \ A = up(\text{DOM}, A)$
Atomic Constraints

- **atomic constraints** define changes in domains
  - Fixing variable: $x_i = d$
  - Removing value: $x_i \neq d$
  - Bounding variable: $x_i \leq d, x_i \geq d$

Atomic constraints are just Boolean literals!

\[
\begin{align*}
  x_i = d & \equiv [x_i = d] \\
  x_i \neq d & \equiv \neg [x_i = d] \\
  x_i \leq d & \equiv [x_i \leq d] \\
  x_i \geq d & \equiv \neg [x_i \leq d - 1]
\end{align*}
\]
When \( f(D) \neq D \) (new information)
- Propagator explains each atomic constraint change
- What part of the current domain \( D \) created the new inference!
  - \( D(x_1) = \{1\}, D(x_2) = D(x_3) = D(x_4) = [1..4], \)
  - \( \text{alldifferent}([x_1, x_2, x_3, x_4]) \)
  - \( f_{\text{alldiff}}(D) \) implies \( x_2 \neq 1, x_3 \neq 1, x_4 \neq 1 \)
  - explanations \( x_1 = 1 \rightarrow x_2 \neq 1, x_1 = 1 \rightarrow x_3 \neq 1, x_1 = 1 \rightarrow x_4 \neq 1, \)
- Adds explanation as clauses, unit propagate on Booleans
- Propagator similarly explains failure.
  - \( D(x_3) = \{3\}, D(x_4) = \{3\}, \text{alldifferent}([x_1, x_2, x_3, x_4]) \)
  - \( f_{\text{alldiff}}(D) \) gives a false domain
  - explanation \( x_3 = 3 \land x_4 = 3 \rightarrow \text{fail} \)
Finite Domain Propagation Example Redux

Consider the problem with:

**Domain** $D_0$:

$$D_0(x_1) = D_0(x_2) = D_0(x_3) = D_0(x_4) = D_0(x_5) = [1..4]$$

**$F$ propagators for:**

$$x_2 \leq x_5, \text{alldifferent}([x_1, x_2, x_3, x_4]), x_1 + x_2 + x_3 + x_4 \leq 9.$$
Lazy Clause Generation example

\[ \text{alldiff} \]

\[
\begin{align*}
&x_1 = 1 \\
&x_2 \neq 1 \\
&x_3 \neq 1 \\
&x_4 \neq 1 \\
\end{align*}
\]

Search: \( x_1 = 1 \)

\[ D(x_1) = \{1\}, \quad D(x_2) = D(x_3) = D(x_4) = D(x_5) = [1..4], \]

Propagate \text{alldifferent}([x_1, x_2, x_3, x_4]) on \text{D}

Determines \( x_2 \neq 1, \ x_3 \neq 1, \ x_4 \neq 1 \)

Explanations \( x_1 = 1 \rightarrow x_2 \neq 1, \ x_1 = 1 \rightarrow x_3 \neq 1, \ x_1 = 1 \rightarrow x_4 \neq 1 \)
Lazy Clause Generation example

\[ \text{alldiff} \]

\[
\begin{align*}
  x_1 &= 1 \\
  x_2 \neq 1 &\rightarrow x_2 \geq 2 \\
  x_3 \neq 1 &\rightarrow x_3 \geq 2 \\
  x_4 \neq 1 &\rightarrow x_4 \geq 2
\end{align*}
\]

Propagate DOM clauses: \( x_2 \neq 1 \rightarrow x_2 \geq 2, \ldots \)

Ignoring DOM clauses: \( x_1 = 1 \rightarrow x_1 \neq 2, x_1 = 1 \rightarrow x_1 \leq 3, \ldots \)

Domain
\[ D(x_1) = \{1\}, D(x_2) = D(x_3) = D(x_4) = [2..4], D(x_5) = [1..4] \]
Lazy Clause Generation example

\textit{alldiff} \quad x_2 \leq x_5

\begin{itemize}
  \item \text{\texttt{\texttt{x1 = 1}}}
  \item \text{\texttt{x2 \neq 1}} \quad \Rightarrow \quad x_2 \geq 2
  \item \text{\texttt{x3 \neq 1}} \quad \Rightarrow \quad x_3 \geq 2
  \item \text{\texttt{x4 \neq 1}} \quad \Rightarrow \quad x_4 \geq 2
  \item \text{\texttt{x5 \geq 2}}
\end{itemize}

\textbf{Propagate} \quad x_2 \leq x_5

Determines \( x_5 \geq 2 \) with explanation \( x_2 \geq 2 \rightarrow x_5 \geq 2 \)

\textbf{FIXPOINT:}
\[ D_1(x_1) = \{1\}, \quad D_1(x_2) = D_1(x_3) = D_1(x_4) = D_1(x_5) = [2..4] \]
Lazy Clause Generation example

\[
\text{alldiff} \quad x_2 \leq x_5
\]

Search \(x_5 \leq 2\)

Domain constraints determine \(x_5 = 2\) with explanation

\[x_5 \geq 2 \land x_5 \leq 2 \rightarrow x_5 = 2\]
Lazy Clause Generation example

\texttt{alldiff} \quad x_2 \leq x_5 \quad x_2 \leq x_5

\begin{align*}
x_1 &= 1 \\
x_2 &\neq 1 \quad x_2 \geq 2 \\
x_3 &\neq 1 \quad x_3 \geq 2 \\
x_4 &\neq 1 \quad x_4 \geq 2 \\
x_5 &\geq 2 \\
\text{Propagate } x_2 \leq x_5
\end{align*}

Determine \( x_2 \leq 2 \) with explanation \( x_5 \leq 2 \Rightarrow x_2 \leq 2 \)
Lazy Clause Generation example

\text{alldiff} \quad x_2 \leq x_5 \quad x_2 \leq x_5

\begin{align*}
\text{x}_1 &= 1 \\
\text{x}_2 \neq 1 &\implies \text{x}_2 \geq 2 \\
\text{x}_3 \neq 1 &\implies \text{x}_3 \geq 2 \\
\text{x}_4 \neq 1 &\implies \text{x}_4 \geq 2 \\
\text{x}_5 \geq 2 &\implies \text{x}_5 \leq 2 \quad \text{x}_5 = 2
\end{align*}

Domain constraints determine $x_2 = 2$ with explanation

$x_2 \geq 2 \land x_2 \leq 2 \rightarrow x_2 = 2$

Domain:

$D(x_1) = \{1\}, \ D(x_2) = \{2\}, \ D(x_3) = D(x_4) = [2..4], \ D(x_5) = \{2\}$
Lazy Clause Generation example

Propagate alldifferent([x₁, x₂, x₃, x₄])

Determines $x₃ \neq 2$ and $x₄ \neq 2$

with explanations $x₂ = 2 \rightarrow x₃ \neq 2$, $x₂ = 2 \rightarrow x₄ \neq 2$, 

$x₅ \leq 2$
Lazy Clause Generation example

\textbf{alldiff} \quad x_2 \leq x_5 \quad x_2 \leq x_5 \quad \textbf{alldiff}

\begin{align*}
&x_1 = 1 \\
&x_2 \neq 1 \rightarrow x_2 \geq 2 \\
&x_3 \neq 1 \rightarrow x_3 \geq 2 \\
&x_4 \neq 1 \rightarrow x_4 \geq 2 \\
&x_5 \geq 2 \rightarrow x_5 \leq 2 \\
&x_2 \leq 2 \rightarrow x_2 = 2 \\
x_3 \neq 2 \rightarrow x_3 \geq 3 \\
x_4 \neq 2 \rightarrow x_4 \geq 3 \\
x_5 \leq 2 \rightarrow x_5 = 2 \\
\end{align*}

Domain constraints determine $x_3 \geq 3$ and $x_4 \geq 3$

\textbf{Domain}

\[ D(x_1) = \{1\}, \ D(x_2) = \{2\}, \ D(x_3) = D(x_4) = [3..4], \ D(x_5) = \{2\} \]

Peter J. Stuckey () Lazy Clause Generation June 2010 42 / 87
Lazy Clause Generation example

\begin{align*}
alldiff & \quad x_2 \leq x_5 \quad x_2 \leq x_5 \quad alldiff \quad \sum \leq 9 \\
\{x_1 = 1\} & \\
x_2 & \neq 1 \quad x_2 \geq 2 \\
x_3 & \neq 1 \quad x_3 \geq 2 \\
x_4 & \neq 1 \quad x_4 \geq 2 \\
x_5 & \geq 2 \\
x_2 & \leq 2 \quad x_2 = 2 \\
x_3 & \neq 2 \quad x_3 \geq 3 \\
x_4 & \neq 2 \quad x_4 \geq 3 \\
x_5 & \leq 2 \quad x_5 = 2 \\
x_3 & \leq 3 \\
x_4 & \leq 3 \\
x_3 & \leq 3 \\
x_4 & \leq 3
\end{align*}

Propagate $x_1 + x_2 + x_3 + x_4 \leq 9$

Determines $x_3 \leq 3$ and $x_4 \leq 3$

with explanations $x_2 \geq 2 \land x_4 \geq 3 \rightarrow x_3 \leq 3$ and similar
Lazy Clause Generation example

\[
\text{alldiff} \quad x_2 \leq x_5 \quad x_2 \leq x_5 \quad \text{alldiff} \quad \sum_i \leq 9
\]

Domain constraints determine \( x_3 = 3 \) and \( x_4 = 3 \)

With explanations \( x_3 \geq 3 \land x_3 \leq 3 \rightarrow x_3 = 3, \ x_4 \geq 3 \land x_4 \leq 3 \rightarrow x_4 = 3 \)

Domain \( D(x_1) = \{1\}, \ D(x_2) = \{2\}, \ D(x_3) = D(x_4) = \{3\}, \ D(x_5) = \{2\} \)
Propagate `alldifferent([x_1, x_2, x_3, x_4])`

Failure detected: explanation $x_3 = 3 \land x_4 = 3 \rightarrow false$
### Lazy Clause Generation Example

<table>
<thead>
<tr>
<th>$x_1 = 1$</th>
<th>alldiff</th>
<th>$x_2 \leq x_5$</th>
<th>$x_5 \leq 2$</th>
<th>$x_2 \leq x_5$</th>
<th>alldiff</th>
<th>$\sum \leq 9$</th>
<th>alldiff</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_1$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$x_2$</td>
<td>[1..4]</td>
<td>[2..4]</td>
<td>[2..4]</td>
<td>[2..4]</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>$x_3$</td>
<td>[1..4]</td>
<td>[2..4]</td>
<td>[2..4]</td>
<td>[2..4]</td>
<td>[3..4]</td>
<td>3</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$x_4$</td>
<td>[1..4]</td>
<td>[2..4]</td>
<td>[2..4]</td>
<td>[2..4]</td>
<td>[3..4]</td>
<td>3</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$x_5$</td>
<td>[1..4]</td>
<td>[1..4]</td>
<td>[2..4]</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

$D_1$  

\textit{fail}
The initial nogood

\[ x_3 = 3 \land x_4 = 3 \rightarrow \text{false} \]
Lazy Clause Generation Explanation

\[ \text{alldiff} \quad x_2 \leq x_5 \quad x_2 \leq x_5 \quad \text{alldiff} \quad \sum \leq 9 \quad \text{alldiff} \]

Resolving

\[ x_4 \geq 3 \land x_4' \leq 3 \land x_3 = 3 \rightarrow \text{false} \]
Resolving

\[ x_3 \geq 3 \land x_4 \geq 3 \land x_3 \leq 3 \land x_4 \leq 3 \rightarrow false \]
Lazy Clause Generation Explanation

Resolving

\[ x_2 \geq 2 \land x_3 \geq 3 \land x_4 \geq 3 \rightarrow false \]
Lazy Clause Generation Explanation

\[ \text{alldiff} \quad x_2 \leq x_5 \quad x_2 \leq x_5 \quad \text{alldiff} \quad \sum \leq 9 \quad \text{alldiff} \]

\[
\begin{align*}
\text{Resolving} & \quad x_2 \geq 2 \land x_3 \geq 2 \land x_4 \geq 2 \land x_3 \neq 2 \land x_4 \neq 2 \rightarrow \text{false} \\
\end{align*}
\]
Lazy Clause Generation Explanation

\[ \text{alldiff} \quad x_2 \leq x_5 \quad x_2 \leq x_5 \quad \text{alldiff} \quad \sum \leq 9 \quad \text{alldiff} \]

Resolving

\[ x_2 \geq 2 \land x_3 \geq 2 \land x_4 \geq 2 \land x_2 = 2 \rightarrow \text{false} \]

Simplify!

\[ x_3 \geq 2 \land x_4 \geq 2 \land x_2 = 2 \rightarrow \text{false} \]
**Lazy Clause Generation Example**

\[ \text{alldiff} \quad x_2 \leq x_5 \quad \text{nogood} \]

\[ x_1 = 1 \]
\[ x_2 \neq 1 \quad x_2 \geq 2 \quad x_2 \neq 2 \]
\[ x_3 \neq 1 \quad x_3 \geq 2 \]
\[ x_4 \neq 1 \quad x_4 \geq 2 \]
\[ x_5 \geq 2 \]

**Backjump**

**Propagate** \[ x_3 \geq 2 \land x_4 \geq 2 \rightarrow x_2 \neq 2 \]
Lazy Clause Generation Example

\textit{alldiff} \quad x_2 \leq x_5 \quad \textit{nogood} \quad x_2 \leq x_5

Domain constraints determine \(x_2 \geq 3\)

\textbf{Propagate} \(x_2 \leq x_5\) determines \(x_5 \geq 3\)

\textbf{Different Domain}

\(D'_2(x_1) = \{1\}, \quad D'_2(x_2) = D'_2(x_5) = [3..4], \quad D'_2(x_3) = D'_2(x_4) = [2..4]\)
What’s Really Happening

- A high level “Boolean” model of the problem
- Clausal representation of the Boolean model is generated “as we go”
- All generated clauses are redundant and can be removed at any time
- We can control the size of the active “Boolean” model

Comparing with SAT on Tai open shop scheduling: (averages)
SAT generates the full Boolean model before starting solving

<table>
<thead>
<tr>
<th></th>
<th>Time</th>
<th>solve only</th>
<th>Fails</th>
<th>Max Clauses Generated</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAT</td>
<td>318</td>
<td>(89)</td>
<td>3597</td>
<td>13.17</td>
</tr>
<tr>
<td>LCG</td>
<td>62</td>
<td></td>
<td>6611</td>
<td>1.00</td>
</tr>
</tbody>
</table>
Strengths and Weaknesses of Lazy Clause Generation

**Strengths**
- High level modelling
- Learning avoids repeating the same subsearch
- Strong autonomous search
- Programmable search
- Specialized global propagators (but requires work)

**Weaknesses**
- Optimization by repeated satisfaction search
- Overhead compared to FD when nogoods are useless
Outline

1. Finite Domain Propagation
   - FD Example

2. SAT Solving
   - SAT Example

3. Lazy Clause Generation
   - Original Lazy Clause Generation
   - Lazier Clause Generation
   - Global Constraints
   - Search

4. Related Work

5. Conclusion
Lazy Boolean Variable Creation

- Many Boolean variables are **never used**
- Create them on demand

**Array encoding**
- Create bounds variables initially $x \leq d$
- Only create equality variables $x = d$ on demand
  Add $x \geq d \land x \leq d \rightarrow x = d$

**List encoding**
- Create bounds variables on demand $x \leq d$
  Add $x \leq d' \rightarrow x \leq d$, $x \leq d \rightarrow x \leq d''$ where $d'$ ($d''$) is next lowest (highest) existing bound
- At most $2 \times$ bounds clauses
- Create equality variables on demand as before
Lazy Boolean Variable Creation Tradeoffs

- List versus array
- List always works! Array may require too many variables
- Implementation complexity
- List hampers learning

Tai open shop scheduling: 15x15 (average of 10 problems)

<table>
<thead>
<tr>
<th></th>
<th>AverageTime</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td>13.38</td>
</tr>
<tr>
<td>list</td>
<td>56.66</td>
</tr>
</tbody>
</table>
- **View** is a pseudo variable defined by a “bijective” function to another variable
  - $x = \alpha y + \beta$
  - $x = \text{bool2int}(y)$
  - $x = \neg y$

- The view variable $x$, does not exist, operations on it are mapped to $y$

- **More important** for lazy clause generation
  - Reduce Boolean variable representation
  - Improve nogoods (reduce search)

Constrained path covering problems: Average of 5 problems

<table>
<thead>
<tr>
<th></th>
<th>Time</th>
<th>Fails</th>
</tr>
</thead>
<tbody>
<tr>
<td>views</td>
<td>0.71</td>
<td>950</td>
</tr>
<tr>
<td>no views</td>
<td>1.12</td>
<td>1231</td>
</tr>
</tbody>
</table>
Explanation Deletion

- Explanations only really needed for nogood learning
  - **Forward** add explanations as they are generated
  - **Backward** delete explanations as we backtrack past them
- **Smaller set of clauses**
- Can hamper search “Reprioritization”

Tai open shop scheduling:

<table>
<thead>
<tr>
<th></th>
<th>15x15</th>
<th>20x20</th>
</tr>
</thead>
<tbody>
<tr>
<td>deletion</td>
<td>13.38</td>
<td>39.96</td>
</tr>
<tr>
<td>no deletion</td>
<td>20.58</td>
<td>95.88</td>
</tr>
</tbody>
</table>

But RCPSP worse with deletion!
Lazy Explanation

- Explanations only needed for nogood learning
  - **Forward** record propagator causing each atomic constraint
  - **Backward** ask propagator to explain atomic constraint (if required)
- Standard for SAT extensions (MiniSAT 1.14) [See Gent et al. PADL2010]
- Only create needed explanations!
- Harder implementation

Social Golfers Problems: using an MDD propagator
(each explanation as expensive as running entire propagator)

<table>
<thead>
<tr>
<th></th>
<th>Time</th>
<th>Reasons</th>
<th>Fails</th>
</tr>
</thead>
<tbody>
<tr>
<td>lazy explanation</td>
<td>2.38</td>
<td>14347</td>
<td>2751</td>
</tr>
<tr>
<td>eager explanation</td>
<td>4.92</td>
<td>78177</td>
<td>5126</td>
</tr>
</tbody>
</table>
Lazy Clause Generation Explanation

\[ \text{alldiff} \quad x_2 \leq x_5 \quad x_2 \leq x_5 \quad \text{alldiff} \quad \sum \leq 9 \quad \text{alldiff} \]

\[ x_1 = 1 \]

\[ x_2 \neq 1 \rightarrow x_2 \geq 2 \quad x_2 \leq 2 \rightarrow x_2 = 2 \]

\[ x_3 \neq 1 \rightarrow x_3 \geq 2 \quad x_3 \neq 2 \rightarrow x_3 \geq 3 \quad x_3 \leq 3 \rightarrow x_3 = 3 \]

\[ x_4 \neq 1 \rightarrow x_4 \geq 2 \quad x_4 \neq 2 \rightarrow x_4 \geq 3 \quad x_4 \leq 3 \rightarrow x_4 = 3 \rightarrow \text{fail} \]

\[ x_5 \geq 2 \rightarrow x_5 \leq 2 \rightarrow x_5 = 2 \]

Dotted boxes explained by above propagator.
Initial nogood

\[ x_3 = 3 \land x_4 = 3 \rightarrow \text{fail} \]
Lazy Clause Generation Explanation

\[
\begin{align*}
\text{alldiff} & \quad x_2 \leq x_5 & \quad x_2 \leq x_5 & \quad \text{alldiff} & \quad \sum \leq 9 & \quad \text{alldiff} \\
\hline
[x_1 = 1] \\
\quad x_2 \neq 1 & \rightarrow x_2 \geq 2 & \rightarrow x_2 \leq 2 & \rightarrow x_2 = 2 \\
\quad x_3 \neq 1 & \rightarrow x_3 \geq 2 \\
\quad x_4 \neq 1 & \rightarrow x_4 \geq 2 \\
\quad x_5 \geq 2 & \rightarrow x_5 \leq 2 & \rightarrow x_5 = 2 \\
\end{align*}
\]

Resolving \( x_3 \geq 3 \land x_3 \leq 3 \rightarrow x_3 = 3 \) and \( x_4 \geq 3 \land x_4 \leq 3 \rightarrow x_4 = 3 \)

\[
x_3 \geq 3 \land x_4 \geq 3 \land x_3 \leq 3 \land x_4 \leq 3 \rightarrow \text{fail}
\]

Request \( x_1 + x_2 + x_3 + x_4 \leq 9 \) to explain \( x_4 \leq 3 \)
Lazy Clause Generation Explanation

\[
\text{alldiff} \quad x_2 \leq x_5 \quad x_2 \leq x_5 \quad \text{alldiff} \quad \sum \leq 9 \quad \text{alldiff}
\]

\[
[x_1 = 1]
\]

\[
x_2 \neq 1 \rightarrow x_2 \geq 2
\]
\[
x_3 \neq 1 \rightarrow x_3 \geq 2
\]
\[
x_4 \neq 1 \rightarrow x_4 \geq 2
\]
\[
x_5 \geq 2 \rightarrow x_5 = 2
\]

Lazy Explanation \( x_2 \geq 2 \land x_3 \geq 3 \rightarrow x_4 \leq 3 \)

Resolving on this gives

\[
x_2 \geq 2 \land x_3 \geq 3 \land x_4 \geq 3 \land x_3 \leq 3 \rightarrow \text{fail}
\]
Lazy Clause Generation Explanation

\[ \text{alldiff} \quad x_2 \leq x_5 \quad x_2 \leq x_5 \quad \text{alldiff} \quad \sum \leq 9 \quad \text{alldiff} \]

\[ x_1 = 1 \]

\[ x_2 \neq 1 \rightarrow x_2 \geq 2 \]
\[ x_2 \leq 2 \rightarrow x_2 = 2 \]

\[ x_3 \neq 1 \rightarrow x_3 \geq 2 \]

\[ x_4 \neq 1 \rightarrow x_4 \geq 2 \]

\[ x_5 \geq 2 \rightarrow x_5 = 2 \]

Final 1UIP nogood

\[ x_2 \geq 2 \land x_3 \geq 2 \land x_4 \geq 2 \land x_2 = 2 \rightarrow \text{false} \]

Note 5 unexplained atomic constraints remain!
Outline

1 Finite Domain Propagation
   - FD Example

2 SAT Solving
   - SAT Example

3 Lazy Clause Generation
   - Original Lazy Clause Generation
   - Lazier Clause Generation
   - Global Constraints
   - Search

4 Related Work

5 Conclusion
The Globality of Explanation

- Nogoods extract **global information** from the problem
- Can overcome **weaknesses** of local propagators

**Example**
- \( D(x_1) = D(x_2) = \lfloor 0 .. 100000 \rfloor \ x_2 \geq x_1 \land (b \leftrightarrow x_1 > x_2) \)
- Set \( b = true \) and 200000 propagations later **failure**. YIKES
- A global difference logic propagator immediately sets \( b = false \)!
- Lazy clause generation learns \( b = false \) after 200000 propagations
  - But **never tries it again**!
Globals by Decomposition

- Globals defined by decomposition
  - Don’t require implementation
  - Automatically incremental
  - Allow partial state relationships to be “learned”
  - Much more attractive with lazy clause generation

- When propagation is not hampered, and size does not blowout:
  - can be good enough!

Resource constrained project scheduling problems: (cumulative by decomposition) closed 62 open problems % solved to optimality in time

<table>
<thead>
<tr>
<th></th>
<th>J60</th>
<th>J90</th>
<th>J120</th>
</tr>
</thead>
<tbody>
<tr>
<td>Laborie</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LCG</td>
<td>85.2</td>
<td>88.1</td>
<td>89.4</td>
</tr>
<tr>
<td></td>
<td>79.8</td>
<td>81.3</td>
<td>82.5</td>
</tr>
<tr>
<td></td>
<td>42.5</td>
<td>44.8</td>
<td>45.3</td>
</tr>
</tbody>
</table>
Which Decomposition?

- Different decompositions interact better or worse with lazy clause generation.
- alldifferent
  - diseq: $O(n^2)$ disequations
  - bnd: Bound consistent decomposition of Bessiere et al IJCAI09
  - bnd+: Bound consistent decomp. replacing $x \geq d \land x \leq d$ by $x = d$
  - gcc: Based on a simple global cardinality decomposition

Quasi-group completion 25x25 (average of examples solved by all)

<table>
<thead>
<tr>
<th>diseq(13)</th>
<th>bnd(11)</th>
<th>bnd + (13)</th>
<th>gcc(15)</th>
<th>CSPComp2008</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>Fails</td>
<td>Time</td>
<td>Fails</td>
<td>Time</td>
</tr>
<tr>
<td>131</td>
<td>142680</td>
<td>757</td>
<td>9317</td>
<td>129</td>
</tr>
<tr>
<td></td>
<td></td>
<td>129</td>
<td>1144</td>
<td>4.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>&gt; 433</td>
</tr>
</tbody>
</table>
Explanations for Globals

-Globals are better than decomposition
  -More efficient
  -Stronger propagation
- Instrument global constraint to also explain its propagations
  -**mdd**: expensive each explanation as much as propagation
  -**cumulative**: choices in how to explain
- Implementation complexity, Can’t learn partial state
- More efficient + stronger propagation

<table>
<thead>
<tr>
<th></th>
<th>J60 (25% faster)</th>
<th>J90 (25% faster)</th>
<th>J120 (60% faster)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>45s</td>
<td>300s</td>
<td>1800s</td>
</tr>
<tr>
<td>Decompression</td>
<td>84.8</td>
<td><strong>89.2</strong></td>
<td>89.4</td>
</tr>
<tr>
<td>Global</td>
<td><strong>85.8</strong></td>
<td>89.0</td>
<td><strong>89.6</strong></td>
</tr>
</tbody>
</table>
Outline

1. Finite Domain Propagation
   - FD Example

2. SAT Solving
   - SAT Example

3. Lazy Clause Generation
   - Original Lazy Clause Generation
   - Lazier Clause Generation
   - Global Constraints
   - Search

4. Related Work

5. Conclusion
Nogoods and Programmed Search

- **Contrary** to SAT folklore
  - Activity based search can be **terrible**
  - Nogoods work **excellently** with programmed search

Constrained Path Covering Problems

<table>
<thead>
<tr>
<th></th>
<th>Time</th>
<th>Fails</th>
</tr>
</thead>
<tbody>
<tr>
<td>nogoods + VSIDS</td>
<td>&gt; 361.89</td>
<td>&gt; 30,000</td>
</tr>
<tr>
<td>nogoods + programmed</td>
<td>0.71</td>
<td>950</td>
</tr>
<tr>
<td>programmed</td>
<td>&gt; 240.2</td>
<td>&gt; 10,000</td>
</tr>
</tbody>
</table>
Activity-based search

- An excellent default search!
- **Weak** at the beginning (no meaningful activities)
- Need **hybrid approaches**
  - Hot Restart:
    - Start with programmed search to “initialize” meaningful activities.
    - Switch to activity-based after restart
  - Use activity-based as part of a programmed search
- Much more to explore in this direction
Outline

1 Finite Domain Propagation
   - FD Example

2 SAT Solving
   - SAT Example

3 Lazy Clause Generation
   - Original Lazy Clause Generation
   - Lazier Clause Generation
   - Global Constraints
   - Search

4 Related Work

5 Conclusion
SAT modulo theories (SMT)

- Combine a SAT solver with theory solvers to handle non-Boolean constraints.
- (Original) Lazy Clause Generation is a special case
  - Each propagator is its own theory
  - Propagators do “theory propagation”
- Differences
  - LCG transmits “lower level” information
  - LCG learns “finer” nogoods
  - LCG supports programmed search
  - Global Propagators ≈ Theories
- Sometimes the theory view is better:
  - modulo arithmetic + Radio Link Frequency Assignment
- Sometimes finer nogoods are better
  - separation logic + Open Shop Scheduling
- Eventually the approaches will merge!
Generalized Nogoods (g-nogoods)

- Nogood learning has a long history in Constraint Programming
  - longer than in SAT?
- Traditional Nogoods: \( x_1 = d_1 \land \cdots \land x_n = d_n \rightarrow \text{fail} \)
- Generalized Nogoods: \( x_1 \neq d_1 \land \cdots \land x_n \neq d_n \rightarrow \text{fail} \)
  - Introduced by Katsirelos and Bacchus 2003
  - Used SAT technology for propagation (watched literals)
  - Equivalent to lazy clause generation without bounds constraints
  - Interesting 1UIP nogoods not effective?
  - Also defined global explanation approach for alldifferent
  - Didnt consider activity, forgetting and VSIDS search
Mixed Integer Programming

**Strengths**
- Can deal with 100K variables 1M linear constraints
- Strong autonomous search
- “Knows” where the good solutions are

**Weaknesses**
- Have to model using only linear constraints

Can we get add the optimization strength of MIP to lazy clause generation?
Hybrid constraint programming and mixed integer programming (MIP)

- Linear constraints as propagators and part of global MIP
- MIP propagator explains failures (and fathoming) as nogoods

\[ x_1 \leq d_1 \land \cdots \land x_n \leq d_n \rightarrow \text{fail} \]

- Propagates these using SAT technology
- Creates ALLUIP nogoods for MIP failures
- Very good results on some hard MIP problems
Lazy Clause Generation and MIP?

- Mixed integer programming (MIP) solvers know where the good solutions are.
- Lazy clause generation and MIP are compatible:
  - MIP engine explains failure and fathoming (and reduced cost bounds changes).
  - Treated like an other global propagator.
  - SCIP is a lazy clause generation MIP solver!
- In order to use the MIP advantage it probably directs search.
- SCIP default search:
  - Pseudo costs (MIP), then activity (SAT), then impact (CP).
- Plenty more to discover on the best interaction! (see our short paper)
Outline

1 Finite Domain Propagation
   - FD Example

2 SAT Solving
   - SAT Example

3 Lazy Clause Generation
   - Original Lazy Clause Generation
   - Lazier Clause Generation
   - Global Constraints
   - Search

4 Related Work

5 Conclusion
Conclusion

Lazy Clause Generation

- High level modelling
- Strong nogood creation
- Effective autonomous search
- Global Constraints

Defines state-of-the-art for:

- Resource constrained project scheduling (minimize makespan)
- Set constraint problems
- Nonagrams (regular constraints)

Usually 1-2 order of magnitude speedup on FD problem
Future Research

Plenty of better engineering yet to be done

Plenty of open research questions

- Best combination with MIP solving
- Hybrid search: structured + activity based
- Parallelism
- SAT Modulo Theories and Lazy Clause Generation
- Adaptive Behaviour
Questions