Search is Dead
Long Live Proof

Peter J. Stuckey and countless others!
What is this talk about

• Solving combinatorial optimization problems
• There are many techniques
  – Mixed integer programming (MIP)
  – Boolean Satisfiability (SAT) and SAT modulo theories
  – Evolutionary Algorithms
  – Local search/Meta heuristics
  – Constraint Programming
• All involve search
  – But complete methods should think about PROOF not SEARCH
• Note that for SAT this notion is well explored
Conspirators

• Ignasi Abio, Ralph Becket, Sebastian Brand, Geoffrey Chu, Michael Codish, Greg Duck, Nick Downing, Thibaut Feydy, Kathryn Francis, Graeme Gange, Vitaly Lagoon, Amit Metodi, Nick Nethercote, Roberto Nieuwenhuis, Olga Ohrimenko, Albert Oliveras, Enric Rodriguez Carbonell, Andreas Schutt, Guido Tack, Pascal Van Hentenryck, Mark Wallace

• All errors and outrageous lies are mine
Constraint Satisfaction Problems

- Finite set of variables \( v \in V \)
  - Each with finite domain \( D(v) \)
- Finite set of constraints \( C \) over \( V \)
- Find a value for each variable that satisfies all the constraints

Example: 3 coloring
- \( V = \{x,y,z,t,u\} \),
- \( D(v) = \{1,2,3\}, \ v \in V \)
- \( C = \{x \neq y, x \neq z, y \neq u, z \neq t, z \neq u, t \neq u\} \)
- Solution \( \{ x=1, y=2, z=2, t=1, u=3\} \)
How much of CP search is repeated?

- 4 colour the graph below

- Inorder labelling: 462672 failures
  - With learning: 18 failures

- Value symmetries removed: 19728 failures
  - With learning: 19 failures

- Reverse labelling: 24 failures
  - With learning: 18 failures
How much of CP search is repeated?

- Resource Constrained Project Scheduling
  - BL instance (20 tasks)

- Input order: 934,535 failures
  - With learning: 931 failures

- Smallest start time order: 296,567 failures
  - With learning: 551 failures

- Activity-based search: > 2,000,000 failures
  - With learning: 1144 failures
How much of CP search is repeated?

- Short answer: a lot

- Methods to alleviate the problem
  - Symmetry/dominance handling
  - Restarts + dynamic search strategies
  - Learning/Caching

- This talk is about how to use learning to avoid this repeated search!
A Brief History of Learning for CP

- Intelligent Backtracking
  - Bruynooghe 1981 (LP)
- Conflict Directed Backjumping
  - Prosser 1993 (CSP)
- Nogood Learning
  - GRASP Marques-Silva and Sakallah 1996 (EDA/SAT)
  - RELSAT Bayardo and Schrag 1996 (CSP)
- G-nogoods
  - Katsirelos and Bacchus 2005 (CP)
- Nogoods from Restarts
  - Lecoutre, Sais, Tabary, + Vidal 2007 (CP)
- Lazy Clause Generation
  - Codish, Ohrimenko + Stuckey 2007 (CP)
Outline

• Propagation based solving
  – Atomic constraints

• Lazy clause generation basics
  – Explaining propagators
  – Conflict resolution

• LCG successes
  – Scheduling, Packing

• Improving LCG
  – How modern LCG solvers work

• Search is Dead

• Concluding remarks
Outline

• Propagation based solving
  – Atomic constraints
• Lazy clause generation basics
  – Explaining propagators
  – Conflict resolution
• LCG successes
  – Scheduling, Packing
• Improving LCG
  – How modern LCG solvers work
• Search is Dead
• Concluding remarks
Propagation based solving

- **domain** $D$ maps `var x` to possible values $D(x)$
- **propagator** $f_c: D \rightarrow D$ for constraint $c$
  - monotonic decreasing function
  - removes value which cannot be part of solution
- **propagation solver** $D = \text{solv}(F, D)$
  - Repeatedly apply propagators $f \in F$ to $D$ until $f(D) = D$ for all $f \in F$
- **finite domain solving**
  - Add new constraint $c$, $D' = \text{solv}(F \cup \{f_c\}, D)$
  - On failure backtrack and add `not c`
  - Repeat until all variables fixed.
Propagation = Inference

• Example: $z \geq y$ propagator $f$
  
  - $D(y) = \{4,5,6\}$, $D(z) = \{0,1,2,3,4,5,6\}$
  
  - $f(D)(y) = \{4,5,6\}$, $f(D)(z) = \{4,5,6\}$

• Domain $D$ is a formula: $D = \land_x x \in D(x)$

• Propagation
  
  - $D \land c \Rightarrow f_c(D)$

• On example
  
  - $y \in \{4,5,6\} \land z \geq y \Rightarrow z \in \{4,5,6\}$

• Separation:
  
  - Core constraints (unary) $\land_x x \in S$ (complete solver)
  
  - Inference of new core constraints from other constraints
Propagation Strength

- Taking into account multiple constraints at once gives stronger propagation

- Example
  - \( \{x_1, x_2, x_3\} \) \( D(\nu) = \{1,2,3,4,5,6,7,8,9\} \)
  - \( x_1 + x_2 + x_3 = 7 \), \text{alldifferent}([x_1, x_2, x_3])

- Individually
  - \( x_1 + x_2 + x_3 = 7 \) \( \Rightarrow \) \( x_1 \in \{1,2,3,4,5\} \) (and \( x_2, x_3 \))
  - \text{alldifferent}([x_1, x_2, x_3]) \text{ nothing new!}

- Together
  - \( \ldots \) \( \Rightarrow \) \( D(\nu) = \{1,2,4\} \)
  - This is how to solve Kakuro puzzles!

- So we should capture complex conjunctions
Problem substructure

- **Assignment substructure:**
  - `alldifferent(x)`: maps each `x` to a different value

- **Hamiltonian circuit substructure:**
  - `circuit(next)`: `next` defines a Hamiltonian tour

- **Resource utilization substructure**
  - `cumulative(s, d, r, L)`: tasks with `starttime` `s`, `duration` `d`, and resource usage `r`, never use more than `L` resources

- **Packing substructure**
  - `diff2(x, y, xd, yd)`: objects at `(xi, yi)` with size `(xd_i, yd_i)` don’t overlap
Finite Domain Propagation Ex.

```
array[1..5] of var 1..4: x;
constraint alldifferent([x[1], x[2], x[3], x[4]]);
constraint x[2] <= x[5];
```

<table>
<thead>
<tr>
<th>x_1</th>
<th>x_2</th>
<th>x_3</th>
<th>x_4</th>
<th>x_5</th>
<th>x_1=1</th>
<th>alldiff</th>
<th>x_2 &lt;= x_5</th>
<th>x_5&gt;2</th>
<th>x_2 &lt;= x_5</th>
<th>alldiff</th>
<th>sum &lt;= 9</th>
<th>alldiff</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1..4</td>
<td>1..4</td>
<td>1..4</td>
<td>1..4</td>
<td>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>1..4</td>
<td>2..4</td>
<td>2..4</td>
<td>2..4</td>
<td>2..4</td>
<td>2..4</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</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>2..4</td>
<td>2..4</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>1..4</td>
<td>1..4</td>
<td>2..4</td>
<td>2..4</td>
<td>2..4</td>
<td>2..4</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
FD propagation

• **Strengths**
  – High level modelling
  – Specialized global propagators capture substructure
    • and all work together
  – Programmable search

• **Weaknesses**
  – Weak autonomous search *(improved recently)*
  – Optimization by repeated satisfaction
  – Small models can be intractable
Outline

• Propagation based solving
  – Atomic constraints
• Lazy clause generation basics
  – Explaining propagators
  – Conflict resolution
• LCG successes
  – Scheduling, Packing
• Improving LCG
  – How modern LCG solvers work
• Search is Dead
• Concluding remarks
Lazy Clause Generation (LCG)

- A hybrid SAT and CP solving approach
- Add *explanation* and *nogood learning* to a propagation based solver
- Key change
  - Modify propagators to explain their inferences as clauses
  - Propagate these clauses to build up an implication graph
  - Use SAT conflict resolution on the implication graph
LCG in a Nutshell

- Integer variable $x$ in $l..u$ encoded as Booleans
  - $[x \leq d]$, $d$ in $l..u-1$
  - $[x = d]$, $d$ in $l..u$
- Dual representation of domain $D(x)$
- Restrict to atomic changes in domain (literals)
  - $x \leq d$ (itself)
  - $x \geq d$, $[x \leq d-1]$ use $[x \geq d]$ as shorthand
  - $x = d$ (itself)
  - $x \neq d$, $[x = d]$ use $[x \neq d]$ as shorthand
- Clauses DOM to model relationship of Booleans
  - $[x \leq d] \Rightarrow [x \leq d+1]$, $d$ in $l..u-2$
  - $[x = d] \Leftrightarrow [x \leq d] \land ! [x \leq d-1]$, $d$ in $l+1..u-1$
LCG in a Nutshell

• Propagation is clause generation
  – e.g. \([x \leq 2] \text{ and } x \geq y \Rightarrow [y \leq 2] \]
  – clause \([x \leq 2] \Rightarrow [y \leq 2] \]

• Consider
  – alldifferent([x[1], x[2], x[3], x[4]]);

• Setting \(x_1 = 1\) we generate new inferences
  – \(x_2 \neq 1, x_3 \neq 1, x_4 \neq 1\)

• Add clauses
  – \([x_1 = 1] \Rightarrow [x_2 \neq 1], [x_1 = 1] \Rightarrow [x_3 \neq 1], [x_1 = 1] \Rightarrow [x_4 \neq 1] \]
  – i.e. ![x_1 = 1] \lor ![x_2 = 1], \ldots\

• Propagate these new clauses
Lazy Clause Generation Ex.

\[
\begin{align*}
\text{alldiff} & \quad x_2 \leq x_5 \\
\text{alldiff} & \quad x_2 \leq x_5 \\
\text{sum} \leq 9 & \quad \text{alldiff} \\
\text{alldiff} &
\end{align*}
\]

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

\[
\begin{align*}
x_2 \leq x_5 & \quad \text{fail}
\end{align*}
\]
1UIP Nogood Creation

1 UIP Nogood

\{x_2 \geq 2, x_3 \geq 2, x_4 \geq 2, x_2 = 2\} \rightarrow false
Backjumping

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

- Backtrack to second last level in nogood
- Nogood will propagate
- Note stronger domain than usual backtracking
  - \( D(x_2) = \{3..4\} \)

\( \{x_2 \geq 2, x_3 \geq 2, x_4 \geq 2, x_2 = 2\} \Rightarrow \text{false} \)
What’s Really Happening

• CP model = **high level** “Boolean” model
• 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 to SAT

• For some models we can generate all possible explanation clauses before commencement
  – usually this is too big
• Open Shop Scheduling (tai benchmark suite)
  – averages

<table>
<thead>
<tr>
<th></th>
<th>Time</th>
<th>Solve only</th>
<th>Fails</th>
<th>Max Clauses</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>6651</td>
<td></td>
<td>1.0</td>
</tr>
</tbody>
</table>
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
LCG for CSPs

• If you are solving extensional CSPs
  – LCG $\cong$ SAT

• Hard to beat SAT on non-numeric CSPs

• Positive table of $n$ tuples of length $k$
  – $k \times n$ binary clauses
  – $1$ $n$-ary clause
  – (for domain propagation) $k \times n$ literals in reverse clauses
  – Actually we can do better with MDDs

• Negative table of $n$ tuples of length $k$
  – $n$ $k$-ary clauses
LCG is SMT

• Each CP propagator is a theory propagator
• They operate on the shared Boolean representation of integer (and other) variables
• But (at least for original LCG) each explanation clause is also recorded
  – Still useful for complex propagators where explanation is expensive, also causes reprioritization
  – Used for state-of-the-art scheduling results.
LCG is not SMT

• Essential differences
  – LCG:
    • focus on optimization
    • communication by literals on domains
    • global constraint propagators with explanation
      – Capturing substructure
  – SMT:
    • focus on theorem proving + verification
    • communication by theory constraints
    • theory "propagators" that treat all similar constraints simultaneously (e.g. difference logic, linear arithmetic)
      – Capturing sub-theories
Outline

• Propagation based solving
  – Atomic constraints

• Lazy clause generation basics
  – Explaining propagators
  – Conflict resolution

• LCG successes
  – Scheduling, Packing

• Improving LCG
  – How modern LCG solvers work

• Search is Dead

• Concluding remarks
LCG Successes

• Scheduling
  – Resource Constrained Project Scheduling Problems (RCPSP)
    • (probably) the most studied scheduling problems
    • LCG closed 71 open problems
    • Solves more problems in 18s then previous SOTA in 1800s
  – RCPSP/Max (more complex precedence constraints)
    • LCG closed 578 open instances of 631
    • LCG recreates or betters all best known solutions by any method on 2340 instances except 3
  – RCPSP/DC (discounted cashflow)
    • Always finds solution on 19440 instances, optimal in all but 152 (versus 832 in previous SOTA)
    • LCG is the SOTA complete method for this problem
LCG Successes

• Real World Application
  – Carpet Cutting
    • Complex packing problem
    • Cut carpet pieces from a roll to minimize length
    • Data from deployed solution
  
  – Lazy Clause Generation Solution
    • First approach to find and prove optimal solutions
    • Faster than the current deployed solution
    • Reduces waste by 35%
LCG Successes

• MiniZinc Challenge
  – comparing CP solvers on a series of challenging problems
  – Competitors
    • CP solvers such as Gecode, Eclipse, SICstus Prolog
    • MIP solvers SCIP, CPLEX, Gurobi (encoding by us)
    • Decompositions to SMT and SAT solvers
  – LCG solvers (from our group) were
    • First (Chuffed) and Second (CPX) in all categories in 2011 and 2012
    • First (Chuffed) in all categories in 2010
  – Illustrates that the approach is strongly beneficial on a wide range of problems
Outline

• Propagation based solving
  – Atomic constraints

• Lazy clause generation basics
  – Explaining propagators
  – Conflict resolution

• LCG successes
  – Scheduling, Packing

• Improving LCG
  – How modern LCG solvers work

• Search is Dead

• Concluding remarks
Improving Lazy Clause Generation

- Don’t Save Explanations
- Lazy Literal Generation
- Lazy (Backwards) Explanation
- The Globality of Explanation
- Explaining Global Constraints
- Search for LCG
- Symmetries and LCG
Don’t Save Explanations

• Explanation clauses are only needed for conflict resolution
  – Don’t record them in the SAT solver
  – Just record them in the implication graph
  – Throw them away on backjumping

• Advantages
  – Less memory required
  – Faster

• Disadvantages
  – Memoizing complex explanations
  – Reprioritizing propagation to follow earlier paths
  – All our scheduling results save explanations
Lazy Literal Generation

• Generate Boolean literals representing integer variables on demand

• E.g.
  – decision \( x_1 = 1 \) generates literal \([x_1 = 1]\)
  – alldiff generates \([x_2 \geq 2]\) (equivalently \(![x_2 \neq 1]\) )

• Integer domain maintains relationship of literals
  – DOM clauses disappear

• A bit tricky to implement efficiently
Lazy Literal Generation

- For constraint problems over large domains, lazy literal generation is crucial (MiniZinc Chall. 2012)

<table>
<thead>
<tr>
<th></th>
<th>amaze</th>
<th>fastfood</th>
<th>filters</th>
<th>league</th>
<th>mspsp</th>
<th>nonogram</th>
<th>patt-set</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial</td>
<td>8690</td>
<td>1043k</td>
<td>8204</td>
<td>341k</td>
<td>13534</td>
<td>448k</td>
<td>19916</td>
</tr>
<tr>
<td>Root</td>
<td>6409</td>
<td>729k</td>
<td>6944</td>
<td>211k</td>
<td>9779</td>
<td>364k</td>
<td>19795</td>
</tr>
<tr>
<td>Created</td>
<td>2214</td>
<td>9831</td>
<td>1310</td>
<td>967</td>
<td>6832</td>
<td>262k</td>
<td>15490</td>
</tr>
<tr>
<td>Percent</td>
<td>34%</td>
<td>1.3%</td>
<td>19%</td>
<td>0.45%</td>
<td>70%</td>
<td>72%</td>
<td>78%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>proj-plan</th>
<th>radiation</th>
<th>shipshed</th>
<th>solbat</th>
<th>still-life</th>
<th>tpp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial</td>
<td>18720</td>
<td>145k</td>
<td>2071k</td>
<td>12144</td>
<td>18947</td>
<td>19335</td>
</tr>
<tr>
<td>Root</td>
<td>18478</td>
<td>43144</td>
<td>2071k</td>
<td>9326</td>
<td>12737</td>
<td>18976</td>
</tr>
<tr>
<td>Created</td>
<td>5489</td>
<td>1993</td>
<td>12943</td>
<td>10398</td>
<td>3666</td>
<td>9232</td>
</tr>
<tr>
<td>Percent</td>
<td>30%</td>
<td>4.6%</td>
<td>0.62%</td>
<td>111%</td>
<td>29%</td>
<td>49%</td>
</tr>
</tbody>
</table>
Lazy Explanation

- Explanations only needed for nogood learning
  - Forward: record propagator causing atomic constraint
  - Backward: ask propagator to explain the constraint
- Standard for SMT and SAT extensions
- Only create needed explanations
- Scope for:
  - Explaining a more general failure than occurred
  - Making use of the current nogood in choosing an explanation
- Interacts well with lazy literal generation
(Original) LCG propagation example

• Variables: \( \{x, y, z\} \) \( D(v) = [0..6] \) Booleans \( b, c \)

• Constraints:
  - \( z \geq y, b \rightarrow y \neq 3, c \rightarrow y \geq 3, c \rightarrow x \geq 6, \)
  - \( 4x + 10y + 5z \leq 71 \) (lin)

• Execution

1UIP nogood: \( c \land [y \neq 3] \rightarrow \) false or \( [y \neq 3] \rightarrow \neg c \)
LCG propagation example

• Execution

\[
\begin{align*}
[x \geq 5] & \quad \text{lin} & b & \quad b \to y \neq 3 & c & \quad c \to y \geq 3 & z \geq y & \quad \text{lin} & \text{false} \\
[y \leq 5] & \quad & [y \neq 3] & \quad & [y \geq 3] & \quad & & \quad & \\
[x \geq 6] & \quad & & & & & & & \\
& & & & c \to x \geq 6 & & & & \\
\end{align*}
\]

Explanation: \( x \geq 6 \) \land \lnot \text{good} \to [x \geq 5] \land [y \geq 4] \land [z \geq 3] \rightarrow \text{false}

Lifted Explanation: \( y \geq 4 \) \lor [y \geq 4] \rightarrow [z \geq 4] \land 4x + 10y + z \leq 7 \rightarrow \text{false}

Lifted Explanation: \( y \geq 3 \land \lnot \text{good} \to [x \geq 5] \land [y \geq 4] \land [z \geq 3] \rightarrow \text{false} \)
LCG propagation example

• Execution

\[ x \geq 5 \]
\[ y \leq 5 \]
\[ y \neq 3 \]
\[ y \geq 3 \]
\[ z \geq 4 \]
\[ x \geq 6 \]

**Nogood:** \( [x \geq 5] \land [y \geq 4] \Rightarrow \text{false} \)

**1UIP Nogood:** \( [x \geq 5] \land [y \geq 4] \Rightarrow \text{false} \)

**1UIP Nogood:** \( [x \geq 5] \Rightarrow [y \leq 3] \)
LCG propagation example

- Backjump

\[
\begin{align*}
[x \geq 5] & \quad \text{lin} & x \geq 5 \implies y \leq 3 \\
[y \leq 5] & & [y \leq 3]
\end{align*}
\]

\textbf{Nogood:}\ [x \geq 5] \land [y \geq 4] \implies \text{false}
### Backwards versus Forwards

<table>
<thead>
<tr>
<th>Class</th>
<th>$n$</th>
<th>forward</th>
<th></th>
<th>backward</th>
<th></th>
<th>clausal</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>time</td>
<td>fails</td>
<td>len</td>
<td>time</td>
<td>fails</td>
<td>len</td>
</tr>
<tr>
<td>amaze</td>
<td>(5)</td>
<td>113</td>
<td>272546</td>
<td>16.8</td>
<td>96</td>
<td>267012</td>
<td>18.2</td>
</tr>
<tr>
<td>fast-food</td>
<td>(4)</td>
<td>345</td>
<td>241839</td>
<td>44.3</td>
<td>264</td>
<td>214918</td>
<td>45.6</td>
</tr>
<tr>
<td>filters</td>
<td>(4)</td>
<td>613</td>
<td>883948</td>
<td>11.2</td>
<td>625</td>
<td>906724</td>
<td>20.7</td>
</tr>
<tr>
<td>league</td>
<td>(2)</td>
<td>11</td>
<td>74483</td>
<td>28.3</td>
<td>10</td>
<td>72737</td>
<td>31.1</td>
</tr>
<tr>
<td>mspsp</td>
<td>(6)</td>
<td>23</td>
<td>55021</td>
<td>24.3</td>
<td>29</td>
<td>62364</td>
<td>53.2</td>
</tr>
<tr>
<td>nonogram</td>
<td>(4)</td>
<td>1965</td>
<td>96461</td>
<td>141.5</td>
<td>2124</td>
<td>90672</td>
<td>168.2</td>
</tr>
<tr>
<td>pattern-set</td>
<td>(2)</td>
<td>451</td>
<td>81397</td>
<td>180.4</td>
<td>400</td>
<td>82410</td>
<td>180.8</td>
</tr>
<tr>
<td>proj-plan</td>
<td>(4)</td>
<td>83</td>
<td>74531</td>
<td>42.1</td>
<td>78</td>
<td>82269</td>
<td>63.4</td>
</tr>
<tr>
<td>radiation</td>
<td>(2)</td>
<td>1.5</td>
<td>7407</td>
<td>17.3</td>
<td>1.3</td>
<td>7566</td>
<td>22.5</td>
</tr>
<tr>
<td>ship-sched</td>
<td>(5)</td>
<td>43</td>
<td>44897</td>
<td>16.0</td>
<td>37</td>
<td>41353</td>
<td>18.2</td>
</tr>
<tr>
<td>solbat</td>
<td>(5)</td>
<td>696</td>
<td>337692</td>
<td>201.2</td>
<td>679</td>
<td>357009</td>
<td>204.0</td>
</tr>
<tr>
<td>still-life</td>
<td>(5)</td>
<td>735</td>
<td>745949</td>
<td>21.9</td>
<td>678</td>
<td>768155</td>
<td>30.2</td>
</tr>
<tr>
<td>tpp</td>
<td>(4)</td>
<td>613</td>
<td>8486</td>
<td>27.4</td>
<td>126</td>
<td>8490</td>
<td>30.1</td>
</tr>
</tbody>
</table>
The Globality of Explanation

• Nogoods extract global information from the problem
• Can overcome weaknesses of local propagators
• Example:
  – \( D(x_1) = D(x_2) = \{0..100000\}, \ x_2 \geq x_1 \land (b \iff x_1 > x_2) \)
  – Set \( b = true \) and 200000 propagations later failure.
• 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!
  – e.g. Resource constrained project scheduling!
Explaining Globals

• Globals are better than decompositions
  – More efficient
  – Stronger propagation

• Instrument global constraint to also explain its propagations
  – regular: each explanation as expensive as propagation
  – cumulative: choices in how to explain

• Implementation complexity
• Can’t learn partial state
• More efficient + stronger propagation + control of explanation
Weak Propagation, Strong Explanation

• Explain a weak propagator strongly
• We get strong explanations, but later!

TTEF propagation
• Strong propagation algorithms less important

Energetic explanation
Weak Propagation, Strong Explanation

- **Late failure** discovery **doesn’t hurt** so much

- **Strong propagators** are not so important!
- **Strong explanations** are important
Search for LCG

- Strong Autonomous Search
- Activity based search
  - Michel and Van Hentenryck CPAIOR 2012
  - Chaff Moskewitz et al DAC 2001
    - Bump activity of all literals seen in conflict resolution
    - Decay activity of all literals periodically
- Concentrates search on literals causing local failure
- Highly local (1000 fails ago is irrelevant)
- The ONLY SEARCH used in SAT and SMT
Search for LCG

• Restarts are (almost) **FREE**
  – All failure detected in previous searches is recorded
  – Restarting never repeats work
    • Whether a fixed search
    • Or a dynamic search

• Aggressive Restarting

• Works well with activity based search
  – Concentrate on failure
Activity-based search can be **BAD**

- **Car sequencing problem**
  - production line scheduling
- **Comparing different search strategies**
  - Static: selecting in order
  - DomWDeg: weight variables appearing in constraints that fail
  - Impact: prioritising decisions that reduce domains
  - Activity based

<table>
<thead>
<tr>
<th></th>
<th>Static</th>
<th>DomWDeg</th>
<th>Impact</th>
<th>Activity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (s)</td>
<td>206.3</td>
<td>0.8</td>
<td>951.3</td>
<td>1522.2</td>
</tr>
<tr>
<td>Solved (70)</td>
<td>66</td>
<td>70</td>
<td>55</td>
<td>47</td>
</tr>
</tbody>
</table>
Hybrid Searches

- Most of our state-of-the-art results use
- Hybrid searches
  - Problem specific objective based search
    - To find good solutions early
  - Switching to activity based search
    - To prove optimality
- Sometimes alternating the two!
- Or throwing a weighted coin to decide which
- More on why this works later
Symmetries and LCG

• LCG interacts well with symmetries
• Symmetry breaking constraints
  – Problem: search strategy disagrees with constraints
  – Solution: activity based search
    • Either the search agrees and constraints get no activity
    • Or the search disagrees and sym constraints get activity
• Dynamic symmetry breaking
  – SBDS is a nogood method
  – Adds symmetric versions of the decision nogood
  – LCG adds symmetric versions of the 1UIP nogood
    • Much stronger
  – No other symmetry breaking method can find these!
Symmetries and LCG

- 5-colour this graph (value symmetry)
- Already coloured $x_1, x_2, x_3, x_4, x_5$
- Setting $x_6 = 1, x_7 = 2$, causes failure
- Dec. Nogood: $x_1 = 1, x_2 = 2, x_3 = 3, x_4 = 4, x_5 = 5, x_6 = 1 \Rightarrow x_7 \neq 2$
- No value symmetric versions are applicable
Symmetries and LCG

- 5-colour this graph (value symmetry)
- Already coloured $x_1, x_2, x_3, x_4, x_5$
- Setting $x_6 = 1, x_7 = 2$, causes failure
- 1UIP Nogood: $x_4 = 4, x_5 = 5, x_6 = 1 \Rightarrow x_7 \neq 2$
- Value Symmetric version is relevant
  - $x_4 = 4, x_5 = 5, x_6 = 1 \Rightarrow x_7 \neq 3$
Symmetries and LCG

- 5-colour this graph (value symmetry)
- Already coloured $x_1, x_2, x_3, x_4, x_5$
- Setting $x_6 = 1, x_7 = 2$, causes failure
- Adding the two nogoods immediately fails with nogood
  - $x_4 = 4, x_5 = 5 \implies x_6 \neq 1$
  - Symmetry gives: $x_4 = 4, x_5 = 5 \implies x_6 \neq 2$ and $x_4 = 4, x_5 = 5 \implies x_6 \neq 3$
Outline

• Propagation based solving
  – Atomic constraints
• Lazy clause generation basics
  – Explaining propagators
  – Conflict resolution
• LCG successes
  – Scheduling, Packing
• Improving LCG
  – How modern LCG solvers work
• Search is Dead
• Concluding remarks
Search is Dead, Long Live Proof

• Search is simply a proof method
  – With learning its lemma generation

• Optimization problems
  – Require us to prove there is no better solution
  – As a side effect we find good solutions
  – Even if we can’t prove optimality,
    • we should still aim to prove optimality

• Primal heuristics (good solutions fast)
  – Reduce the size of optimality proof

• Dual heuristics (good lower bounds fast)
  – Reduce the size of the optimality proof
Search is Dead, Long Live Proof

• The role of Search
  – Find good solutions
    • Only if this helps the proof size to be reduced
  – Find powerful nogoods (lemmas)
    • That are reusable and hence reduce proof size

• Other inferences can reduce proof size
  – Symmetries
  – Dominance
  – Stronger propagators (stronger base inference)

• And a critical factor for reducing proof size
  – Stronger languages of learning
The Language of Learning

• Is **critical**

• Consider the following MiniZinc model
  
  - `array[1..n] of var 1..n: x;`
  
  - `constraint alldifferent(x);`
  
  - `constraint sum(x) < n*(n+1) div 2;`

• Unsatisfiable

  - **No learning**
  
<table>
<thead>
<tr>
<th>n</th>
<th>Failures</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>240</td>
<td>0.00</td>
</tr>
<tr>
<td>7</td>
<td>1680</td>
<td>0.01</td>
</tr>
<tr>
<td>8</td>
<td>13440</td>
<td>0.08</td>
</tr>
<tr>
<td>9</td>
<td>120960</td>
<td>0.42</td>
</tr>
<tr>
<td>10</td>
<td>1209600</td>
<td>4.47</td>
</tr>
</tbody>
</table>

  - **With learning**
  
<table>
<thead>
<tr>
<th>n</th>
<th>Failures</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>270</td>
<td>0.00</td>
</tr>
<tr>
<td>7</td>
<td>1890</td>
<td>0.02</td>
</tr>
<tr>
<td>8</td>
<td>15120</td>
<td>0.20</td>
</tr>
<tr>
<td>9</td>
<td>136080</td>
<td>2.78</td>
</tr>
<tr>
<td>10</td>
<td>1360800</td>
<td>31.30</td>
</tr>
</tbody>
</table>
The Language of Learning

• Is critical

• Consider the following MiniZinc model

  - `array[1..n] of var 1..n: x;`
  - `array[1..n] of var 0..n*(n+1) div 2: s;`
  - `constraint alldifferent(x);`
  - `constraint s[1] = x[1] \ s[n] < n*(n+1) div 2;`
  - `constraint forall(i in 2..n) (s[i]=x[i]+s[i-1]);`

• Unsatisfiable
  - No learning

<table>
<thead>
<tr>
<th>n</th>
<th>Failures</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>240</td>
<td>0.00</td>
</tr>
<tr>
<td>7</td>
<td>1680</td>
<td>0.01</td>
</tr>
<tr>
<td>8</td>
<td>13440</td>
<td>0.08</td>
</tr>
<tr>
<td>9</td>
<td>120960</td>
<td>0.56</td>
</tr>
<tr>
<td>10</td>
<td>1209600</td>
<td>5.45</td>
</tr>
</tbody>
</table>

  - With learning

<table>
<thead>
<tr>
<th>n</th>
<th>Failures</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>99</td>
<td>0.00</td>
</tr>
<tr>
<td>7</td>
<td>264</td>
<td>0.01</td>
</tr>
<tr>
<td>8</td>
<td>657</td>
<td>0.01</td>
</tr>
<tr>
<td>9</td>
<td>1567</td>
<td>0.04</td>
</tr>
<tr>
<td>10</td>
<td>3635</td>
<td>0.12</td>
</tr>
</tbody>
</table>
The Language of Lemmas

• **Critical** to improving proof size
• Choose the **right language** for expressing lemmas
• See
  – Lazy encoding. CP2013
  – Structure based extended resolution
• Constraint Programming has a **massive advantage** over other complete methods since we “know” the substructures of the problem
Outline

• Propagation based solving
  – Atomic constraints

• Lazy clause generation basics
  – Explaining propagators
  – Conflict resolution

• LCG successes
  – Scheduling, Packing

• Improving LCG
  – How modern LCG solvers work

• Search is Dead

• Concluding remarks
Conclusions

• Most of CP search is repeated
• Remember the past to avoid repeating it
• Search is only a mechanism for generating good lemmas
• Consider other mechanisms for proof size reduction
  – inference, language, dominance, relaxation, decomposition, primal heuristics, CEGAR
What's left to be done?

- Language of Learning
- Explaining propagators
  - Sometime building strong explanation is hard
- Conflict directed explanation
  - We can take into account the current conflict while explaining
- Dominances and LCG
  - Dynamic dominance breaking search with learning
- Parallelizing LCG
  - Good luck! It seems proof is essentially sequential
Whats coming

• **ObjectiveCP**
  – CP based on a small micro kernel
  – See Pascals talk

• **ObjectiveCPExplanation**
  – An LCG solver in the ObjectiveCP framework

• **ObjectiveCPSchedule**
  – State of the art scheduling technology

• **MiniZinc 2.0**
Final Word

• NICTA optimization group is looking for a constraint programmer
  – Supply chains and logistics

• University of Melbourne should be advertising for a lecturer position soon in Optimization

• We are always keen to host interns in the “worlds most livable city”

• So come and join us!