NICTA

# Laziness is next to Godliness

Peter J. Stuckey and countless others!

Australian Government
Department of Broadband, Communications and the Digital Economy
Australian Research Council

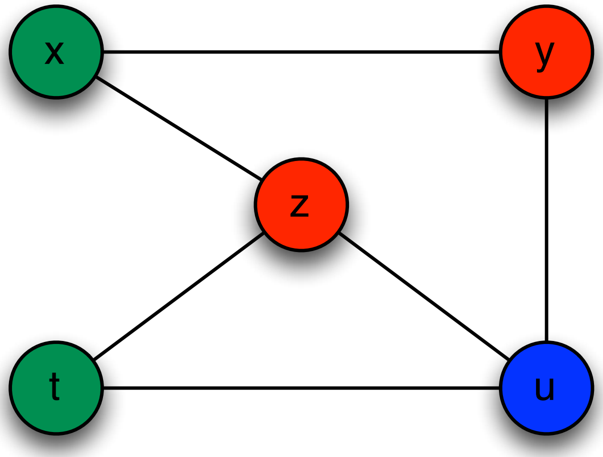
NICTA Funding and Supporting Members and Partners

# Conspirators

- Ignasi Abio, Ralph Becket, Sebastian Brand, Geoffrey Chu, Michael Codish, Broes De Cat, Marc Denecker, 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

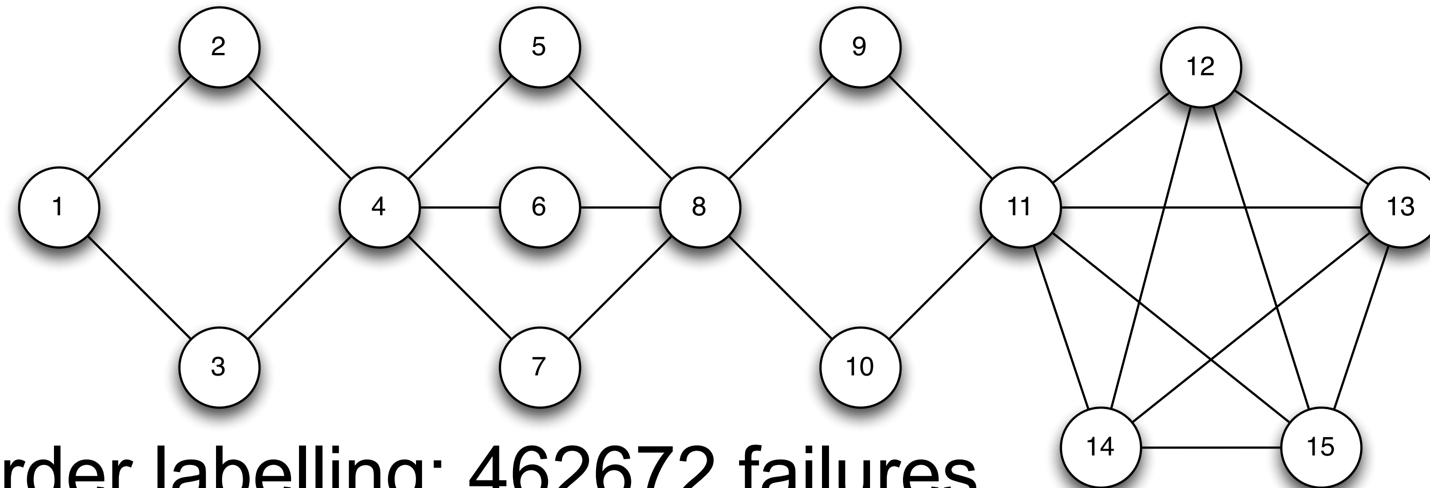
- Finite set of variables $v \in V$
  - Each with finite domain $D(v)$
- Finite set of constraints $C$ over $V$
- Find a value for each variable that satisfies all the constraints
- Example: 3 coloring
  - $V = \{x,y,z,t,u\}$,
  - $D(v) = \{1,2,3\}$, $v \in V$
  - $C = \{x \neq y,\ x \neq z,\ y \neq u,$
    $z \neq t,\ z \neq u,\ t \neq u\}$
  - Solution { x=1, y=2, z=2, t=1, u=3}



# How much of CP search is repeated?

NICTA

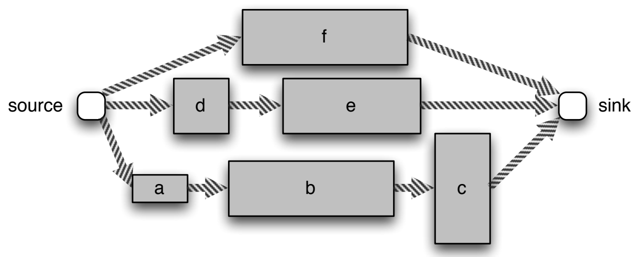
- 4 colour the graph below

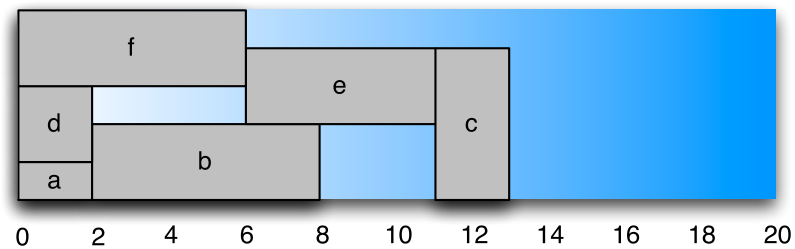



- Inorder labelling: 462672 failures
  - With learning: 18 failures
- Value symmetries removed: 19728 failures
  - With learning: 19 failures
- Reverse labelling: 24 failures
  - With learning: 18 failures

# 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

# Outline

- Propagation based solving
  - Atomic constraints
- Lazy clause generation
  - Explaining propagators
  - Conflict resolution
  - How modern LCG solvers work
- The language of learning: Why search is dead!
  - Lazy encoding
  - Structure based extended resolution
- Lazy grounding and nested constraint programs
- The laziness principle
- Concluding remarks

# Outline

- Propagation based solving
  - Atomic constraints
- Lazy clause generation
  - Explaining propagators
  - Conflict resolution
  - How modern LCG solvers work
- The language of learning: Why search is dead!
  - Lazy encoding
  - Structure based extended resolution
- Lazy grounding and nested constraint programs
- The laziness principle
- Concluding remarks

# Propagation Solving (CP)

- Complete solver for atomic constraints
  - $x = d$, $x \neq d$, $x \geq d$, $x \leq d$
  - Domain $D(x)$ records the result of solving (!)
- Propagators infer new atomic constraints from old ones
  - $x_2 \leq x_5$ infers from $x_2 \geq 2$ that $x_5 \geq 2$
  - $x_1+x_2+x_3+x_4 \leq 9$ infers from $x_1 \geq 1 \wedge x_2 \geq 2 \wedge x_3 \geq 3$ that $x_4 \leq 3$
- Inference is interleaved with search
  - Try adding $c$ if that fails add *not* $c$
- Optimization is repeated solving
  - Find solution $obj = k$ resolve with $obj < k$

# 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];
constraint x[1] + x[2] + x[3] + x[4] <= 9;
```

| | $x_1$=1 | *alldiff* | $x_2 \le x_5$ | $x_5$>2 | $x_2 \le x_5$ | *alldiff* | *sum≤9* | *alldiff* |
|---|---|---|---|---|---|---|---|---|
| $x_1$ | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| $x_2$ | 1..4 | 2..4 | 2..4 | 2..4 | 2 | 2 | 2 | 2 |
| $x_3$ | 1..4 | 2..4 | 2..4 | 2..4 | 2..4 | 3..4 | 3 | ✖ |
| $x_4$ | 1..4 | 2..4 | 2..4 | 2..4 | 2..4 | 3..4 | 3 | ✖ |
| $x_5$ | 1..4 | 1..4 | 2..4 | 3..4 | 2 | 2 | 2 | 2 |

# 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
  - Explaining propagators
  - Conflict resolution
  - How modern LCG solvers work
- The language of learning: Why search is dead!
  - Lazy encoding
  - Structure based extended resolution
- Lazy grounding and nested constraint programs
- The laziness principle
- 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

NICTA

- 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)
  - $x \geq d$ $! [x \leq d\text{-}1]$ 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
  - $[x = d] \Leftrightarrow [x \leq d] \wedge ! [x \leq d\text{-}1]$, $d$ in $l$+1..$u$-1

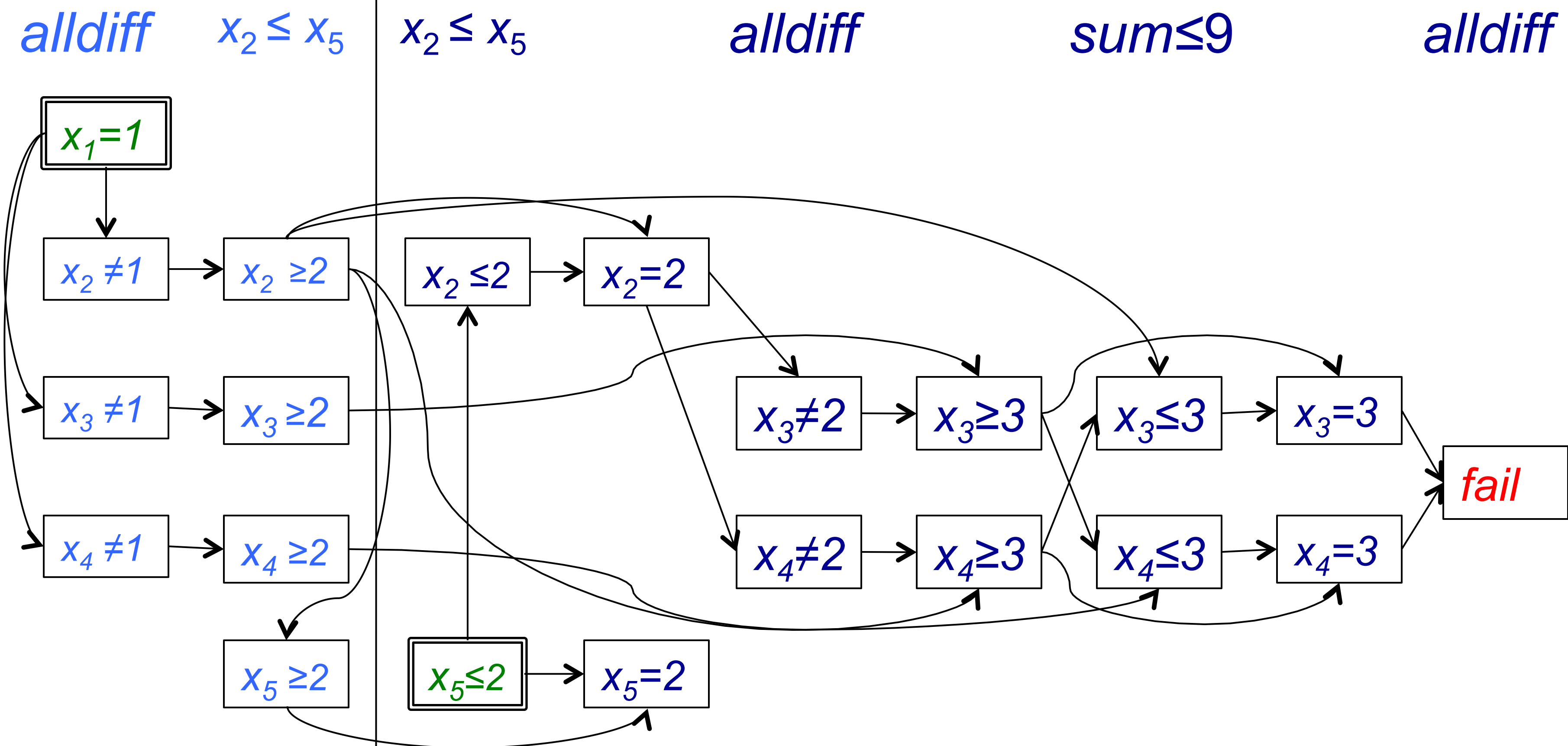
# LCG in a Nutshell

- Propagation is clause generation
  - e.g. $[x \leq 2]$ and $x \geq y$ means that $[y \leq 2]$
  - clause $[x \leq 2] \rightarrow [y \leq 2]$
- Consider

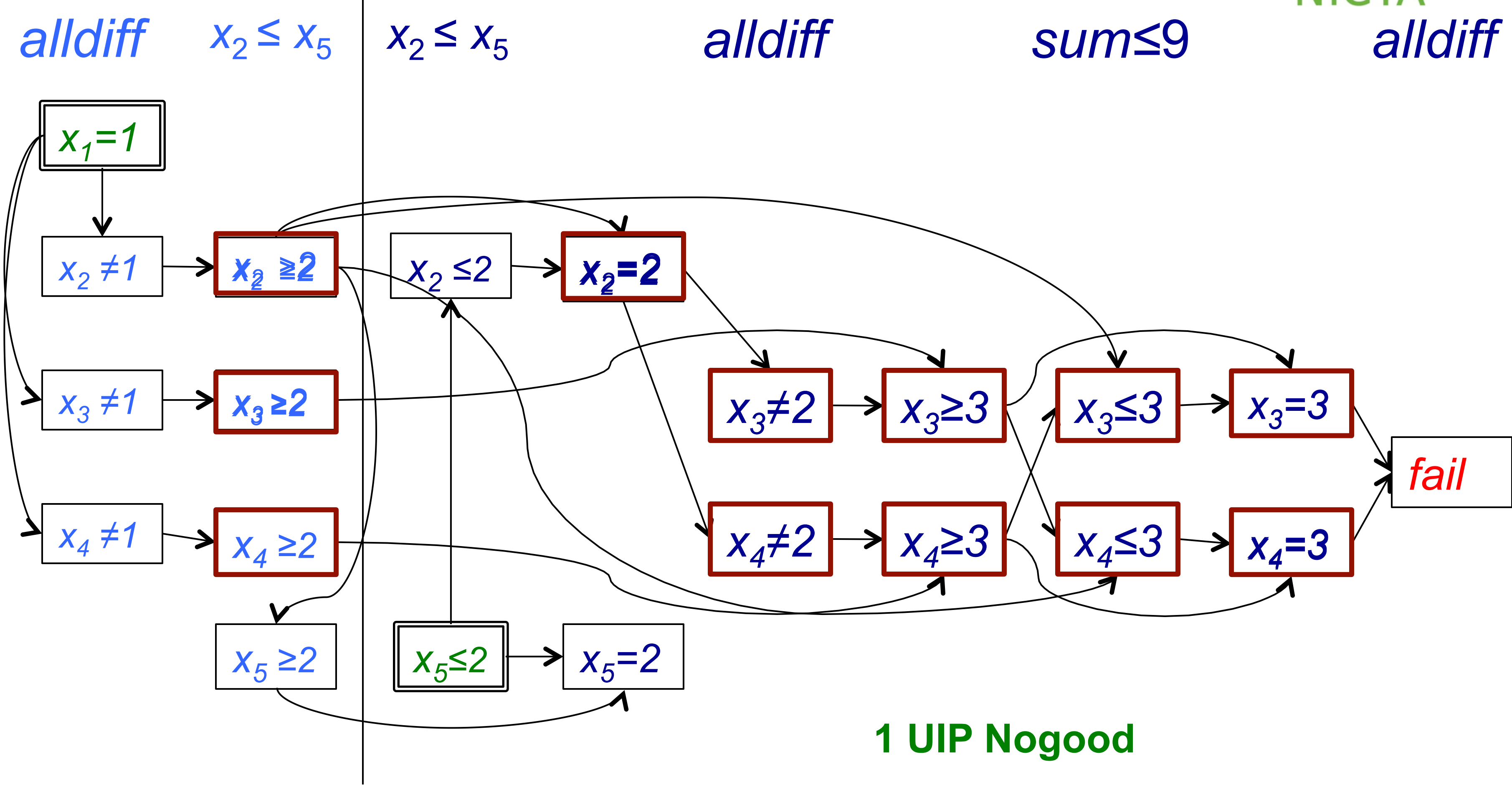
```
- alldifferent([x[1],x[2],x[3],x[4]);
```

- 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. $![x_1 = 1] \vee ![x_2 = 1]$, …
- Propagate these new clauses

# Lazy Clause Generation Ex.



# 1UIP Nogood Creation

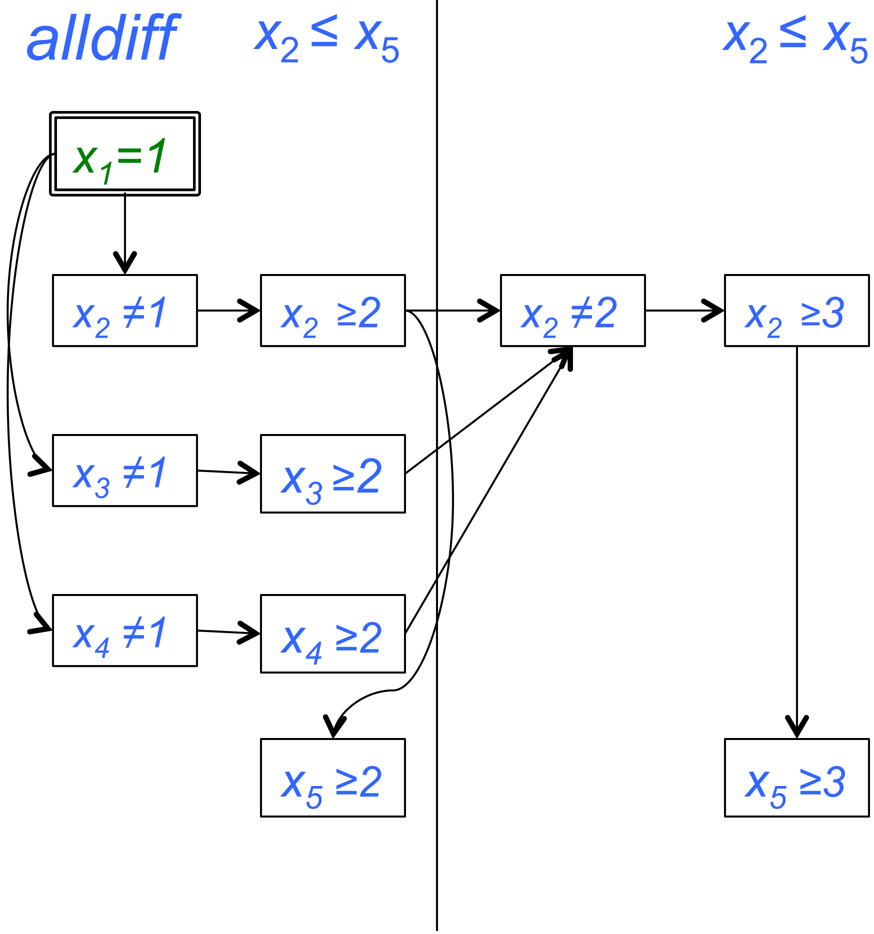



**1 UIP Nogood**

$\{[x_2 \leq 1],[x_3 \leq 1], [x_4 \leq 1], \neg[x_2 = 2]\}$

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

# Backjumping




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

$\{x_2 \geq 2, x_3 \geq 2, x_4 \geq 2, x_2 = 2\} \rightarrow$ *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

| | Time | Solve only | Fails | Max Clauses |
|---|---|---|---|---|
| SAT | 318 | 89 | 3597 | 13.17 |
| LCG | 62 | | 6651 | 1.0 |

# Lazy Clause Generation

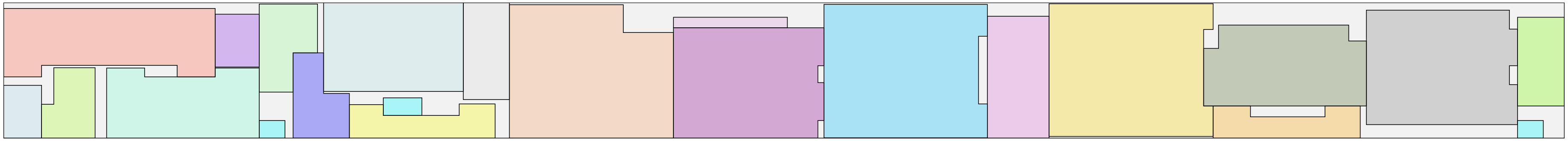
- Strengths
  - High level modelling
  - Learning avoids repeating the same subsearch
  - Strong autonomous search
  - Programmable search
  - Specialized global propagators (but requires work)
- Weaknesses
  - Optimization by repeated satisfaction search
  - Overhead compared to FD when nogoods are useless

# LCG 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

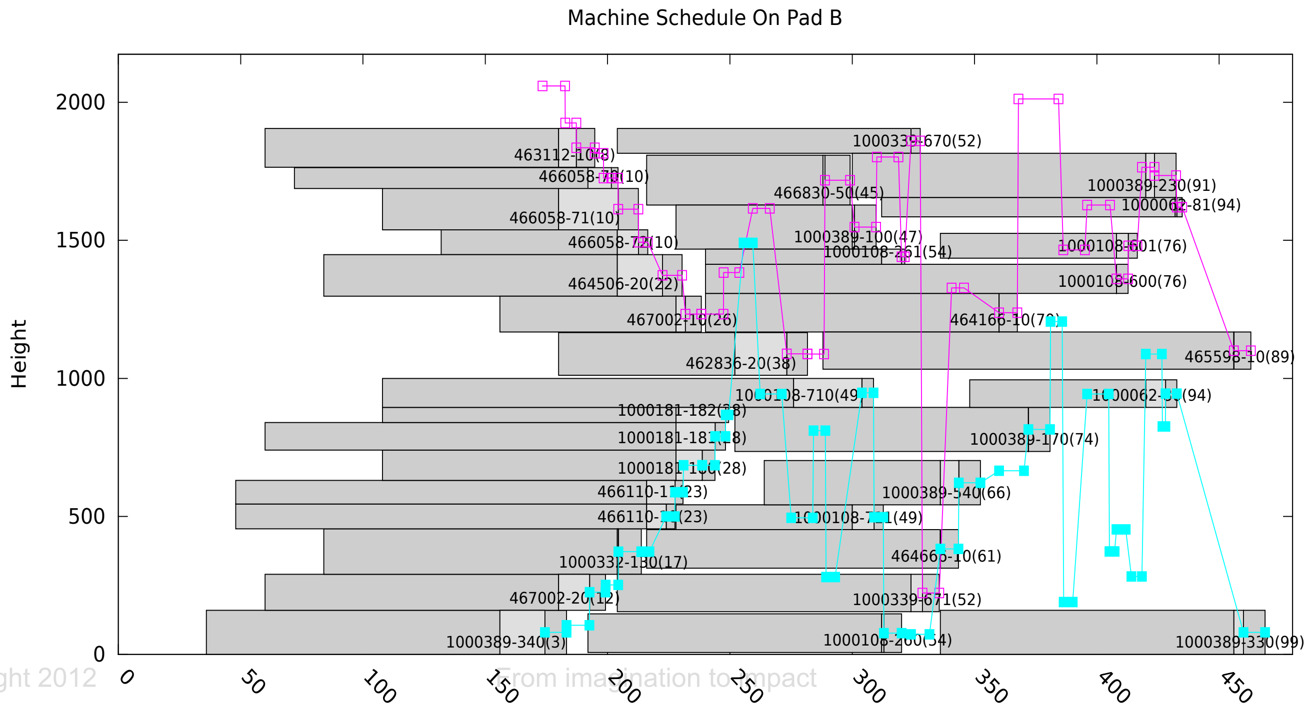
- Real World Application
  - Carpet Cutting
    - Complex packing problem
    - Cut carpet pieces from a roll to minimize length
    - Data from deployed solution



  - Lazy Clause Generation Solution
    - First approach to find and prove optimal solutions
    - Faster than the current deployed solution
    - Reduces waste by 35%

# LCG Successes

- Real World Application
  - – Bulk Mineral Port Scheduling
    - Combined scheduling problem and packing problem
    - Pack placement of cargos on a pad over time (2d)
    - Schedule reclaiming of cargo onto ship
    - LCG solver produces much better solutions



# 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

# Improving Lazy Clause Generation

- Don't Save Explanations
- Lazy Literal Generation
- Lazy (Backwards) Explanation
- The Globality of Explanation
- Weak Propagation, Strong Explanation
- Search for LCG
- Symmetries and LCG

# 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)

| | amaze | fastfood | filters | league | mspsp | nonogram | patt-set |
|---|---|---|---|---|---|---|---|
| Initial | 8690 | 1043k | 8204 | 341k | 13534 | 448k | 19916 |
| Root | 6409 | 729k | 6944 | 211k | 9779 | 364k | 19795 |
| Created | **2214** | **9831** | **1310** | **967** | **6832** | **262k** | **15490** |
| Percent | 34% | 1.3% | 19% | 0.45% | 70% | 72% | 78% |

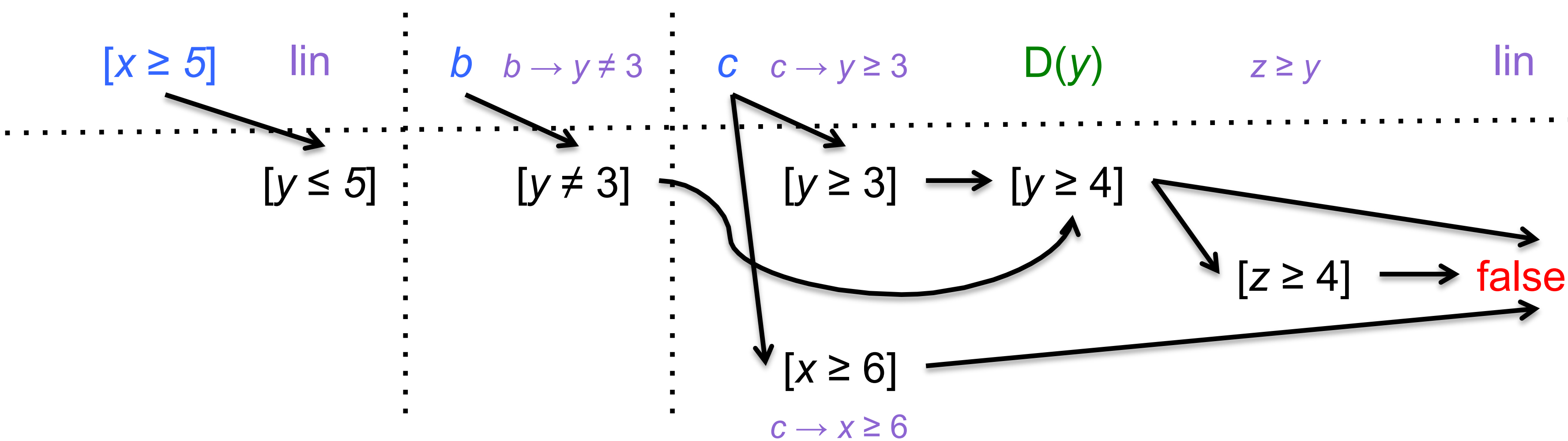
| | proj-plan | radiation | shipshed | solbat | still-life | tpp |
|---|---|---|---|---|---|---|
| Initial | 18720 | 145k | 2071k | 12144 | 18947 | 19335 |
| Root | 18478 | 43144 | 2071k | 9326 | 12737 | 18976 |
| Created | **5489** | **1993** | **12943** | **10398** | **3666** | **9232** |
| Percent | 30% | 4.6% | 0.62% | 111% | 29% | 49% |

# 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: $c \wedge [y \neq 3] \rightarrow$ false or $[y \neq 3] \rightarrow !c$

# LCG propagation example

- Execution

$[x \geq 5]$ lin | $b$ $b \rightarrow y \neq 3$ | $c$ $c \rightarrow y \geq 3$ $z \geq y$ lin

$[y \leq 5]$ $[y \neq 3]$ $[y \geq 3]$ $[z \geq 4]$ false

$[x \geq 6]$

$c \rightarrow x \geq 6$

Absorbtion

**Explanation:** $x \geq 6 \wedge y \geq 4 \wedge z \geq 4 \wedge 4x + 10y + 5z \leq 73$ ➔ false

**Nogood:** $[x \geq 5] \wedge [y \geq 4] \wedge [z \geq 7/3]$ ➔ false

**Lifted Explanation:** $x \geq 5 \wedge y \geq 4 \wedge z \geq 3 \wedge 4x + 10y + 5z \leq 73$ ➔ false

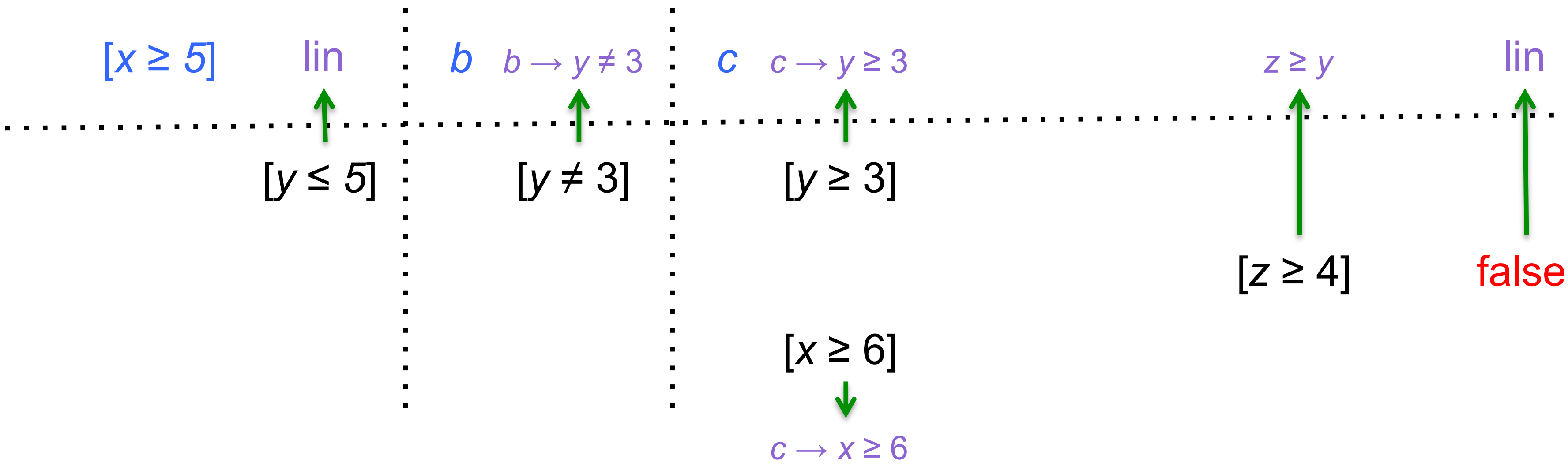
**Lifted Explanation:** $y \geq 4 \wedge z \geq y$ ➔ $z \geq 4$

**Lifted Explanation:** $y \geq 3 \wedge z \geq y$ ➔ $z \geq 3$

**Nogood:** $[x \geq 5] \wedge [y \geq 4] \wedge [z \geq 3]$ ➔ false

# LCG propagation example

- Execution




**Nogood:** $[x \geq 5] \wedge [y \geq 4] \rightarrow$ false

**1UIP Nogood:** $[x \geq 5] \wedge [y \geq 4] \rightarrow$ false

**1UIP Nogood:** $[x \geq 5] \rightarrow [y \leq 3]$

# LCG propagation example

- Backjump

$[x \geq 5]$ lin $x \geq 5 \rightarrow y \leq 3$

$[y \leq 5]$ $[y \leq 3]$

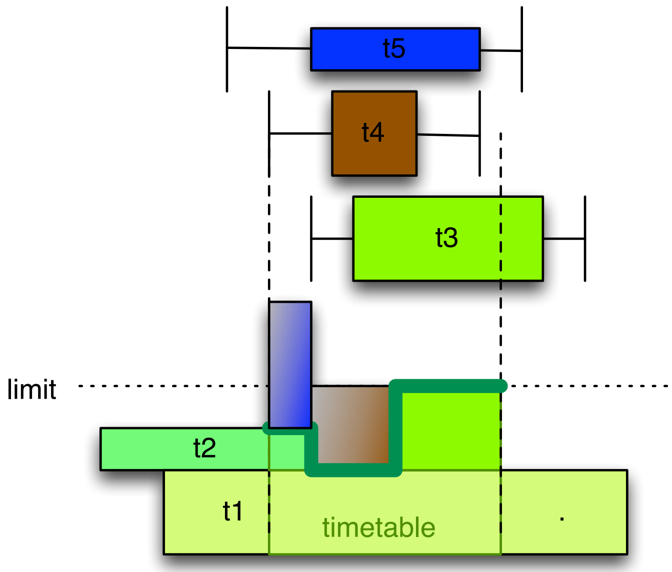
**Nogood:** $[x \geq 5] \wedge [y \geq 4] \rightarrow$ false

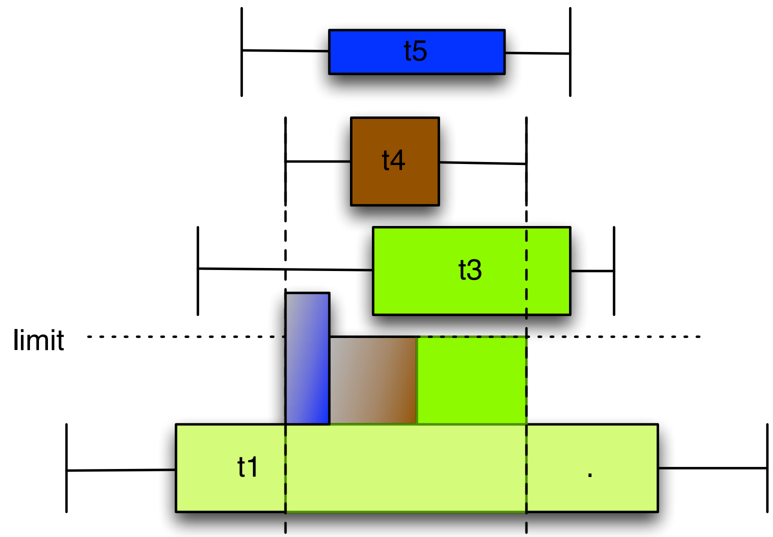
# Backwards versus Forwards

| Class | $n$ | forward | | | backward | | | clausal | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | time | fails | len | time | fails | len | time | fails | len |
| amaze | (5) | 113 | 272546 | **16.8** | **96** | 267012 | 18.2 | 455 | **242110** | 22.2 |
| fast-food | (4) | 345 | 241839 | **44.3** | **264** | **214918** | 45.6 | >2617 | $58027^2$ | 159.8 |
| filters | (4) | **613** | **883948** | **11.2** | 625 | 906724 | 20.7 | >901 | $7331^1$ | 10.0 |
| league | (2) | 11 | 74483 | **28.3** | **10** | **72737** | 31.1 | 14 | 81679 | 34.9 |
| mspsp | (6) | **23** | **55021** | **24.3** | 29 | 62364 | 53.2 | 44 | 70511 | 24.6 |
| nonogram | (4) | **1965** | 96461 | 141.5 | 2124 | 90672 | 168.2 | >3126 | $32805^2$ | 144.8 |
| pattern-set | (2) | 451 | **81397** | **180.4** | **400** | 82410 | 180.8 | >1016 | $3913^1$ | 3505.3 |
| proj-plan | (4) | 83 | **74531** | 42.1 | **78** | 82269 | 63.4 | 150 | 89860 | 46.2 |
| radiation | (2) | 1.5 | **7407** | **17.3** | **1.3** | 7566 | 22.5 | 1.5 | 7382 | 19.9 |
| ship-sched | (5) | 43 | 44897 | **16.0** | **37** | **41353** | 18.2 | 273 | 71120 | 23.2 |
| solbat | (5) | 696 | **337692** | **201.2** | **679** | 357009 | 204.0 | >1528 | $111477^1$ | 239.1 |
| still-life | (5) | 735 | **745949** | **21.9** | **678** | 768155 | 30.2 | >2640 | $269664^2$ | 23.7 |
| tpp | (4) | 613 | 8486 | 27.4 | **126** | 8490 | 30.1 | >902 | $8330^1$ | 12.7 |

# Weak Propagation, Strong Explanation

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

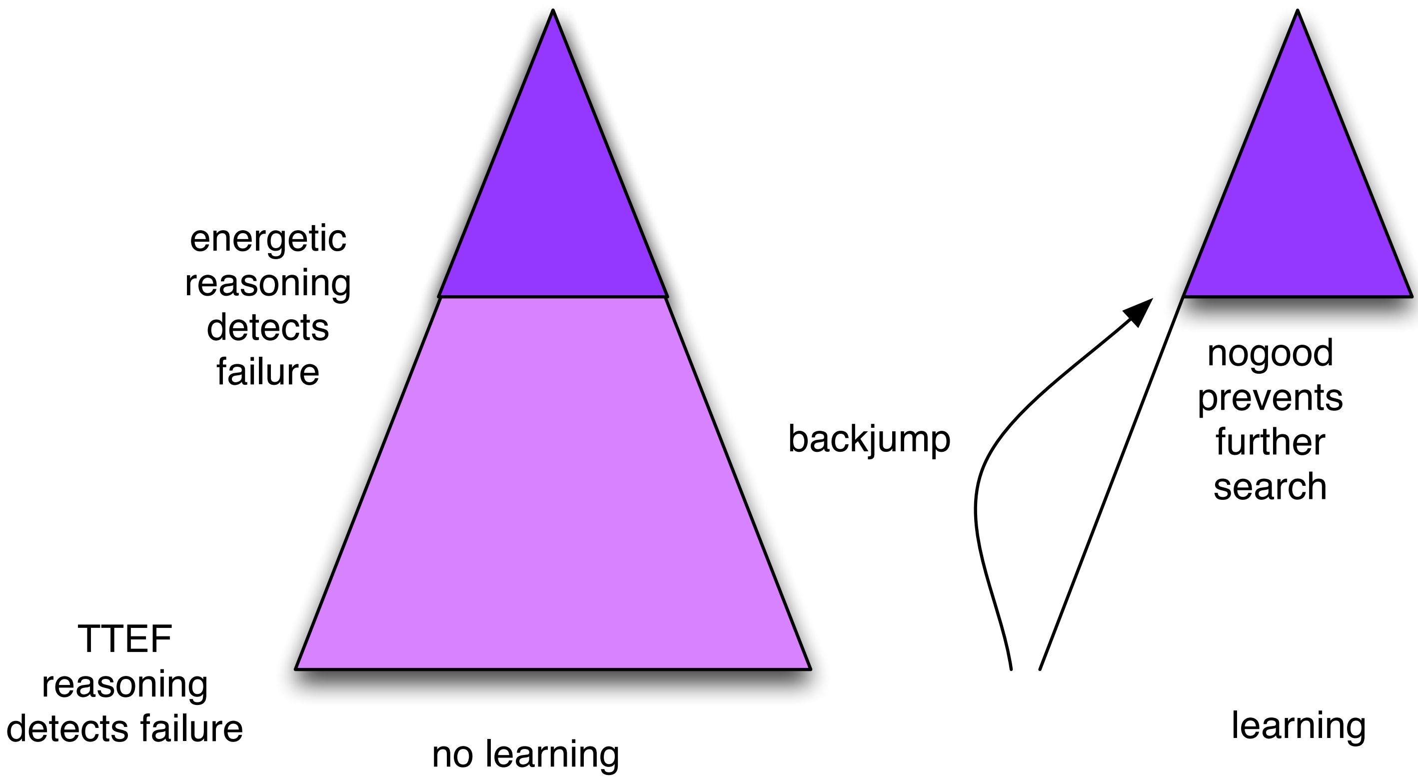



- TTEF propagation    Energetic explanation
- Strong propagation algorithms less important

# Weak Propagation, Strong Explanation

- Late failure discovery doesn't hurt so much




- Strong propagators are not so important!
- Strong explanations are important

# Outline

- Propagation based solving
  - Atomic constraints
- Lazy clause generation
  - Explaining propagators
  - Conflict resolution
  - How modern LCG solvers work
- The language of learning: Why search is dead!
  - Lazy encoding
  - Structure based extended resolution
- Lazy grounding and nested constraint programs
- The laziness principle
- 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 cant 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 | With learning

| n | Failures | Time (s) |
|---|---|---|
| 6 | 240 | 0.00 |
| 7 | 1680 | 0.01 |
| 8 | 13440 | 0.08 |
| 9 | 120960 | 0.42 |
| 10 | 1209600 | 4.47 |

| n | Failures | Time (s) |
|---|---|---|
| 6 | 270 | 0.00 |
| 7 | 1890 | 0.02 |
| 8 | 15120 | 0.20 |
| 9 | 136080 | 2.78 |
| 10 | 1360800 | 31.30 |

# 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

| n | Failures | Time (s) |
|---|---|---|
| 6 | 240 | 0.00 |
| 7 | 1680 | 0.01 |
| 8 | 13440 | 0.08 |
| 9 | 120960 | 0.56 |
| 10 | 1209600 | 5.45 |

With learning

| n | Failures | Time (s) |
|---|---|---|
| 6 | 99 | 0.00 |
| 7 | 264 | 0.01 |
| 8 | 657 | 0.01 |
| 9 | 1567 | 0.04 |
| 10 | 3635 | 0.12 |

# The Language of Lemmas

- Critical to improving proof size
- Choose the right language for expressing lemmas
- Constraint Programming has a massive advantage over other complete methods since we “know” the substructures of the problem
- Methods
  - Lazy Encoding
  - Structure based extended resolution

# Propagation Versus Encoding to SAT

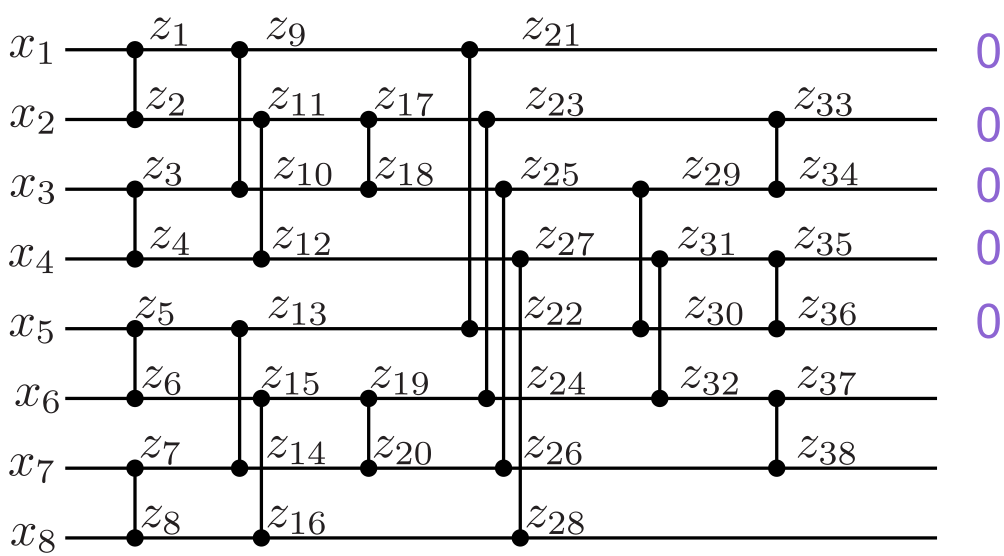
- Experience with cardinality problems
- 501 instances of problems with a single cardinality constraint
  - unsat-based MAXSAT solving

| | | Speed up if encoding | | | | Slow down if encoding | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Suite | TO | 4 | 2 | 1.5 | Win | 1.5 | 2 | 4 | TO | Win |
| Card | 168 | 54 | 14 | 7 | 243 | 7 | 24 | 215 | 12 | 258 |

- 50% of instances encoding is better, 50% worse
- Why can propagation be superior?

# Example: Cardinality constraints

- *x1 + x2 + x3 + x4 + x5 + x6 + x7 + x8* ≤ 3
- Propagator
  – If 3 of {*x1*, …, *x8*} are true, set the rest false.
- Encoding
  – Cardinality or sorting network:
    - z21 = z33 = z34 = z35 = z36 = 0



# Comparison: Encoding vs Propagation

- A propagator
  - Lazily generates an encoding
  - This encoding is partially stored in nogoods
  - The encoding uses no auxiliary Boolean variables
  - $\Sigma_{i=1..n}\, x_i \leq k$ generates $(n\text{-}k)^n C_k = O(n^k)$ explanations
- If the problem is UNSAT (or optimization)
  - CP solver runtime ≥ size of smallest resolution proof
  - Cannot decide on auxiliary variables
    - Exponentially larger proof
  - Compare $\Sigma_{i=1..n}\, x_i \leq k$ encoding is $O(n \log^2 k)$
- But propagation is faster than encoding

# Lazy Encoding

- Choose at runtime between encoding and propagation
- All constraints are initially propagators
- If a constraint generates many explanations
  - Replace the propagator by an encoding
  - At restart (just to make it simple)
- Policy: encode if either
  - The number of different explanations is > 50% of the encoding size
  - More than 70% of explanations are new and > 5000

# Structure Based Extended Resolution

- Internal data structures of global constraints = candidate variables for language of learning
- Examples
  - linear constraints $\Sigma_{i=1..n}\, a_i\, x_i \leq k$ :
    - partial sums $s_j = \Sigma_{i=1..j}\, a_i\, x_i$
  - lexicographic $[x_1, \ldots, x_i, \ldots, x_n] \leq [y_1, \ldots, y_i, \ldots, y_n]$
    - Example propagation $[2, 5, 3, x_4, x_5] \leq [2, 5, y_3, y_4, y_5]$
    - $x_1 = 2 \wedge y_1 = 2 \wedge x_2 = 5 \wedge y_2 = 5 \wedge x_3 \geq 3 \rightarrow y_3 \geq 3$
    - $x_1 < y_1 \vee (x_1 = y_1 \wedge (x_2 < y_2 \vee (x_2 = y_2 \wedge \ldots)))$
    - comparison literals: $x_i < y_i$ $x_i = y_i$
    - $x_1 \geq y_1 \wedge x_2 \geq y_2 \wedge x_3 \geq 3 \rightarrow y_3 \geq 3$
    - A much more reusable explanation!

# Structure Based Extended Resolution

- Examples
  - table constraints

$(x_1,x_2,x_3,x_4) \in \{ (1,2,3,4), (4,3,2,1), (1,2,2,3),$
$(3,1,2,1), (1,1,1,1) \}$

  - Example propagation: $x_1 = 1 \wedge x_2 = 2 \rightarrow x_4 \neq 1$
  - Best explanation: $x_1 \neq 4 \wedge x_2 \neq 1 \rightarrow x_4 \neq 1$
  - OR $x_2 \neq 3 \wedge x_2 \neq 1 \rightarrow x_4 \neq 1$
  - $r_i$ = tuple $i$ is selected
    - $r_2 = (x_1 = 4 \wedge x_2 = 3 \wedge x_3 = 2 \wedge x_4 = 1)$
  - Maximally general explanation
    - $!r_2 \wedge !r_4 \wedge !r_5 \rightarrow x_4 \neq 1$

# Structure Based Extended Resolution

- 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
  - With learning / With extended resolution

With learning:

| n | Failures | Time (s) |
|---|---|---|
| 6 | 270 | 0.00 |
| 7 | 1890 | 0.02 |
| 8 | 15120 | 0.20 |
| 9 | 136080 | 2.78 |
| 10 | 1360800 | 31.30 |

With extended resolution:

| n | Failures | Time (s) |
|---|---|---|
| 6 | 99 | 0.00 |
| 7 | 264 | 0.01 |
| 8 | 657 | 0.01 |
| 9 | 1567 | 0.04 |
| 10 | 3635 | 0.12 |

# Structure Based Extended Resolution

- Extend global propagators to
  - Explain propagation using “internal literals”
  - Maintain truth value of “internal literals”
    - usually already part of the propagation algorithm
- Many benefits of lazy encoding
  - not all, sometimes other literals are very useful
    - e.g. cardinality encodings
  - piggy back “extended resolution” on globals algorithm

# Outline

- Propagation based solving
  - Atomic constraints
- Lazy clause generation
  - Explaining propagators
  - Conflict resolution
  - How modern LCG solvers work
- The language of learning: Why search is dead!
  - Lazy encoding
  - Structure based extended resolution
- Lazy grounding and nested constraint programs
- The LAZINESS principle
- Concluding remarks

# Lazy Grounding

- Before solving we usually have to
  - ground (or flatten) the model
- For example (n = 4)
  - `constraint forall(i in 2..n)(s[i]=x[i]+s[i-1]);`
  - becomes `s[2] = x[2] + s[1] /\ s[3] = x[3] + s[2] /\ s[4] = x[4] + s[3]`
- For some models the grounding is enormous!
- Instead of grounding before solving
  - ground during solve
  - on demand
  - ensure a solution for the non-grounded part
- See the next talk for more details!

# Nested Constraint Programs

- A powerful language for nested optimization problems
- Based on aggregator constraints
  - $y$ = **agg**( [ $f(x_1,\ldots,x_n,z_1,\ldots,z_m)$
    | $z_1,\ldots,z_m$ where $C(x_1,\ldots,x_n,z_1,\ldots,z_m)$ ])

  where **agg** is a function on multisets
  - e.g. sum, min, max, average, median, exists, forall
- Lazy evaluation
  - wait until $x_1,\ldots,x_n$ are fixed
  - evaluate the multiset by search over $z_1,\ldots,z_m$
  - set $y$ to the appropriate value

# Nested Constraint Programs

- Highly expressive:
  - #SAT, QBF, QCSP, Stochastic CP, …
- Find the minimal number of clues $x_{ikjk} = d_k$ required to make a proper sudoku problem (exactly one solution)
- $y$ = min( [ sum([$b_k$ | $k$ in 1..$n$]) | $b_1$, …, $b_n$ where
- 1 = sum([ 1 | $x_{11}$ in 1..9, … $x_{99}$ in 1..9 where
- forall([ $b_k \rightarrow x_{ikjk} = d_k$ | $k$ in 1..$n$]) /\
- **sudoku**($x_{11}$, …, $x_{99}$) ] ) ] )
- where **sudoku** are sudoku constraints

# Nested Constraint Programs

- Naïve approach
  - completely solved by grounding
  - BUT completely impractical
- Actual approach
  - one copy of constraints
  - search on outer aggregator
    - wake a new copy of inner aggregator
- Improvements
  - learning (across invocations of inner aggregators)
  - short circuiting (e.g. when we find two solns we stop)
  - use grounding when known size and small

# Nested Constraint Programs

- Book production (stochastic) planning problem
  - uncertain demand 100..105 in each period
  - plan a production run so that we can cover demand 80% of the time
- Compare with stochastic CP using search and scenario generation (determinization)

| stages | NCP | | search | | scenario | |
|---|---|---|---|---|---|---|
| | fails | time | fails | time | fails | time |
| 1 | 8 | 0.01 | 10 | 0.01 | 4 | 0.00 |
| 2 | 16 | 0.01 | 148 | 0.03 | 8 | 0.02 |
| 3 | 24 | 0.01 | 3604 | 0.76 | 24 | 0.16 |
| 4 | 32 | 0.01 | 95570 | 19.07 | 42 | 1.53 |
| 5 | 40 | 0.01 | 2616858 | 509.95 | 218 | 18.52 |
| 6 | 48 | 0.01 | --- | TO | 1260 | 474.47 |

# Outline

- Propagation based solving
  - Atomic constraints
- Lazy clause generation
  - Explaining propagators
  - Conflict resolution
  - How modern LCG solvers work
- The language of learning: Why search is dead!
  - Lazy encoding
  - Structure based extended resolution
- Lazy grounding and nested constraint programs
- The laziness principle
- Concluding remarks

# The LAZINESS Principle

- "Never perform any work unless there is evidence that it will benefit"
  - LCG = lazy SAT encoding
  - Lazy literal generation = only when needed
  - Lazy explanation = only when needed
  - Lazy encoding = intermediate literals when needed
  - Structure based ER = as for Lazy encoding
  - Lazy grounding = model expansion as needed
  - Nested constraint programs = copy the submodel as required

# The LAZINESS principle

- “Never perform any work unless there is evidence that it will benefit”
- Where does it lead?
- Ideas:
  - only do constraint checking until a constraint causes failure often, then start propagating it
  - don’t learn at all until there are lots of failures
- Obviously other methods are instances of this
  - Benders decomposition
  - Column generation

# Outline

- Propagation based solving
  - Atomic constraints
- Lazy clause generation
  - Explaining propagators
  - Conflict resolution
  - How modern LCG solvers work
- The language of learning: Why search is dead!
  - Lazy encoding
  - Structure based extended resolution
- Lazy grounding and nested constraint programs
- The laziness principle
- Concluding remarks

# Conclusions (Slogans)

- Most of CP search is **repeated**
- Search is Dead, **Long Live Proof**
- Laziness is your friend
  - **Follow the LAZINESS principle!**
- And finally

**Laziness is next to Godliness**

# Whats coming

- ObjectiveCP
  - CP based on a small micro kernel
- ObjectiveCPExplanation
  - An LCG solver in the ObjectiveCP framework
- ObjectiveCPSchedule
  - State of the art scheduling technology
- MiniZinc 2.0

# MiniZinc 2.0 Beta (www.minizinc.org)

- Open LLVM-style architecture
- User-defined functions
  - Functional constraint modelling, functional globals
  - Better CSE
- Option types
  - Concise modelling of decisions that are only relevant dependent on other decisions
- Half reification
  - Better translation of complex logical constraints
  - Substantial efficiency improvements
  - More flexible use of globals
- Globalizer (powerful structural analysis)