

NICTA Funding and Supporting Members and Partners

| Australian National University | $\cup N S M$ | $\begin{aligned} & =\left({ }^{N}\right) \\ & \text { NSW } \end{aligned}$ | Trade \& Investment |  |  |  |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| SYDNEY | Queensland Government |  | い \( |  |  |  |
| ) MJ Grififfith | बण |  | $\begin{aligned} & \text { The University } \\ & \text { Of QueEnslant } \\ & \text { australia } \end{aligned}$ |  |  |  |

## Conspirators

- Ignasi Abio, Ralph Becket, Sebastian Brand, Geoffrey Chu, Michael Codish, Greg Duck, Nick Downing, Thibaut Feydy, Graeme Gange, Vitaly Lagoon, Amit Metodi, Alice Miller, Nick Nethercote, Roberto Nieuwenhuis, Olga Ohrimenko, Albert Oliveras, Patrick Prosser, Enric Rodriguez Carbonell, Andreas Schutt, Guido Tack, Mark Wallace
- All errors and outrageous lies are mine


## Outline

- Modelling and solving
- Propagation based solving
- The advantages of keeping structure
- Better (static) CNF encoding
- Dynamic choice: propagation versus CNF encoding
- Propagation with learning (Lazy Clause Generation)
- MiniZinc
- Conclusion


## A famous problem (in CNF)

c unknown problem
p cnf 69
120
340
560
$\begin{array}{lll}-1 & -3 & 0\end{array}$
$\begin{array}{lll}-1 & -5 & 0\end{array}$
$-3-50$
-2 -4 0
$-2-6 \quad 0$
$-4-60$

## A famous problem (in CNF)

c unknown problem
p conf 1222
$\begin{array}{lllllllllllllll}1 & 2 & 3 & 0 & 4 & 5 & 6 & 0 & 7 & 8 & 9 & 0 & 10 & 11 & 12\end{array} 0$


$\begin{array}{lllllllll}-2 & -5 & 0 & -2 & -8 & 0 & -2 & -11 & 0\end{array}$
$-5-8 \quad 0 \quad-5-1100-8-110$
$\begin{array}{lllllllll}-3 & -6 & 0 & -3 & -9 & 0 & -3 & -12 & 0\end{array}$


## A famous problem (in MiniZinc)

int: n ;
array[1..n] of var 1..n-1: $x$; constraint alldifferent(x); solve satisfy;
$\mathrm{n}=4 ; \quad \%$ data could be \% in different file

## A famous problem (in MiniZinc)

int: n;
set of int: Pigeon $=1 . . n$; set of int: Hole = 1..n-1; array[Pigeon] of var Hole: x; constraint alldifferent(x); solve satisfy;
$\mathrm{n}=4 ; \quad \%$ data could be \% in different file

## A famous problem (in MiniZinc)

int: $n$;
set of int: Pigeon $=1 \ldots n$; set of int: Hole = 1..n-1; array[Pigeon] of var Hole: x; constraint alldifferent(x); solve satisfy;
constraint varsym(x) ;
constraint valsym(x, 1..n-1);
$\mathrm{n}=4 ; \quad \%$ data could be \% in different file

## A famous problem (in SMT-LIB?)

(declare-fun x1 () Int)
(declare-fun x2 () Int)
(declare-fun x3 () Int)
(declare-fun x4 () Int)
(assert (and (<x1 4) (> x1 0)))
(assert (and (<x2 4) (> x2 0)))
(assert (and (<x3 4) (> x3 0)))
(assert (and (<x4 4) (> x4 0)))
(assert (and (distinct x1 x2)
(distinct x1 x3) (distinct x1 x4) (distinct $x 2$ x3) (distinct $x 2$ x4) (distinct x3 x4))

## A famous problem (in SMT-LIB?)

(declare-fun $x 1$ () Int)
(declare-fun x2 () Int)
(declare-fun x3 () Int)
(declare-fun x4 () Int)
(assert (and (< x1 4) (> x1 0)))
(assert (and (< x2 4) (> x2 0)))
(assert (and (< x3 4) (> x3 0)))
(assert (and (<x4 4) (> x4 0)))
(assert (alldifferent x1 x2 x3 x4))

## Modelling and Solving



- The conceptual model
- A formal mathematical statement of the (simplified) problem
- The design model
- In the form that can be handled by a solver


## Modelling and Solving



## Modelling and Solving in SAT



## Modelling and Solving in MiniZinc



Outline

- Modelling and solving
- Propagation based solving
- The advantages of keeping structure
- Better (static) CNF encoding
- Dynamic choice: propagation versus CNF encoding
- Propagation with learning (Lazy Clause Generation)
- MiniZinc
- Conclusion


## 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 $D=\operatorname{solv}(F, D)$
- Repeatedly apply propagators $f \in F$ to $D$ until $f(D)=D$ for all $f \in F$
- finite domain solving
- Add new constraint $c, D^{\prime}=\operatorname{solv}\left(F \cup\left\{f_{c}\right\}, D\right)$
- On failure backtrack and add not $c$
- Repeat until all variables fixed.


## Propagation = Inference

- Example: $z \geq y$ propagator $f$

$$
\begin{aligned}
& -D(y)=\{4,5,6\}, D(z)=\{0,1,2,3,4,5,6\} \\
& -f(D)(y)=\{4,5,6\}, f(D)(z)=\{4,5,6\}
\end{aligned}
$$

- Domain $D$ is a formula: $D=\wedge_{x} x \in D(x)$
- Propagation

$$
-D \wedge c \rightarrow f_{c}(D)
$$

- On example
$-y \in\{4,5,6\} \wedge z \geq y \rightarrow z \in\{4,5,6\}$
- Separation:
- Core constraints (unary) $\wedge_{x} x \in S$ (complete solver)
- Inference of new core constraints from other constraints


## Propagation Strength

- Taking into account multiple constraints at once gives stronger propagation
- Example

$$
\begin{aligned}
& -\{x 1, x 2, x 3\} \mathrm{D}(v)=\{1,2,3,4,5,6,7,8,9\} \\
& -x 1+x 2+x 3=7, \text { alldifferent }([x 1, x 2, x 3])
\end{aligned}
$$

- Individually
$-x 1+x 2+x 3=7 \rightarrow \mathrm{D}(v)=\{1,2,3,4,5\}$
- alldifferent([x1,x2,x3]) nothing new!
- Together
$-\ldots \Rightarrow D(v)=\{1,2,4\}$
- This is how to solve Kakuro puzzles!
- So we should capture complex conjunctions


## Problem substructure

- Assignment substructure:
- alldifferent (x): maps each xto 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 then L resources
- Packing substructure
- diff2 ( $x, y, x d, y d)$ objects at $\left(x_{i}, y_{i}\right)$ with size $\left(\mathrm{xd}_{\mathrm{i}}, \mathrm{yd}_{\mathrm{i}}\right)$ don't overlap


## FD propagation example

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

$$
\begin{aligned}
& -z \geq y, b \rightarrow y \neq 3, c \rightarrow y \geq 3, c \rightarrow x \geq 6 \\
& -4 x+10 y+5 z \leq 71 \text { (lin) }
\end{aligned}
$$

- Example search

|  | $x \geq 5$ | lin | $b$ | $b \rightarrow y \neq 3$ | $c$ | $c \rightarrow y \geq 3$ | $c \rightarrow x \geq 6$ | $z \geq y$ | lin |
| :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- |
| $D(x)$ | $5 . .6$ |  |  |  |  |  | 6 | 0 |  |
| $D(y)$ | $0 . .6$ | $0 . .5$ |  | $0 . .2,4 . .5$ |  | $4 . .5$ |  | 0 |  |
| $D(z)$ | $0 . .6$ |  |  |  |  |  | $4 . .6$ | 0 |  |
| $D(b)$ | $0 . .1$ |  | 1 |  |  |  |  |  |  |
| $D(c)$ | $0 . .1$ |  |  |  | 1 |  |  |  |  |

## FD propagation example

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

$$
\begin{aligned}
& -z \geq y, b \rightarrow y \neq 3, c \rightarrow y \geq 3, c \rightarrow x \geq 6, \\
& -4 x+10 y+5 z \leq 71 \text { (lin) }
\end{aligned}
$$

- Failure detected,
- backtrack and reverse last decision

|  | $x \geq 5$ | lin | $b$ | $b \rightarrow y \neq 3$ | not $c$ |
| :--- | :--- | :--- | :--- | :--- | :--- |
| $D(x)$ | $5 . .6$ |  |  |  |  |
| $D(y)$ | $0 . .6$ | $0 . .5$ |  | $0 . .2,4 . .5$ |  |
| $D(z)$ | $0 . .6$ |  |  |  |  |
| $D(b)$ | $0 . .1$ |  | 1 |  |  |
| $D(c)$ | $0 . .1$ |  |  |  | 0 |

- Special case of FD propagation
- All variables $x$ are binary $D(x)=\{0,1\}$
- All constraints $c$ are clauses
- All propagators $f_{c}$ are handled by unit propagation
- Constraints added in search are $x$ or not $x$


## FD propagation

- Strengths
- High level modelling
- Specialized global propagators capture substructure
- and all work together
- Programmable search
- Weaknesses
- Weak autonomous search
- Optimization by repeated satisfaction
- Small models can be intractable

Outline

- Modelling and solving
- Propagation based solving
- The advantages of keeping structure
- Better (static) CNF encoding
- Dynamic choice: propagation versus CNF encoding
- Propagation with learning (Lazy Clause Generation)
- MiniZinc
- Conclusion


## Outline

- Modelling and solving
- Propagation based solving
- The advantages of keeping structure
- Better (static) CNF encoding
- Dynamic choice: propagation versus CNF encoding
- Propagation with learning (Lazy Clause Generation)
- MiniZinc
- Conclusion


## Better encoding to SAT



## Better CNF encoding

- Not all SAT encodings are equal
- Significant research encoding constraints to SAT
- Atmostone
- Cardinality constraints
- Psuedo-Boolean constraints
- Integer variables
- Significant research on "improving" a CNF model after encoding: preprocessing.


## Example: encoding Sudoku



At least

## $X_{i j k}=\operatorname{cell}(\mathrm{i}, \mathrm{j})$

 contains value $k$$$
\begin{aligned}
\operatorname{one}\left(b_{1}, \ldots, b_{n}\right) & =\left(b_{1} \vee \cdots \vee b_{n}\right) \wedge \\
\text { At most } & \bigwedge_{i<j}\left(\bar{b}_{i} \vee \bar{b}_{j}\right)
\end{aligned}
$$

## So? What's the Problem?



$$
\begin{aligned}
& 1,000,000 \text { 's of clauses: } \\
& \text { 100,000's of variables: } \\
& \text { Bugs are hard to track; } \\
& \text { Optimizations are costly }
\end{aligned}
$$



## Example: encoding Sudoku

| 5 | 3 |  |  | 7 |  |  |  |  |
| :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- |
| 6 |  |  | 1 | 9 | 5 |  |  |  |
|  | 9 | 8 |  |  |  |  | 6 |  |
| 8 |  |  |  | 6 |  |  |  | 3 |
| 4 |  |  | 8 |  | 3 |  |  | 1 |
| 7 |  |  |  | 2 |  |  |  | 6 |
|  | 6 |  |  |  |  | 2 | 8 |  |
|  |  |  | 4 | 1 | 9 |  |  | 5 |
|  |  |  |  | 8 |  |  | 7 | 9 |

```
var 1..9: x11;
var 1..9: x12;
```

alldifferent([x11, ... x19]);
alldifferent([x21, ..., x29]);
x11 = 5;
$x 12=3 ;$


## The Usual Approach



## Our Approach



## Our Approach


more powerful reasoning but on smalel|evfl artanspropagation based solver!

Core constraints: literal equations (complete solver is congruence closure) Other constraints: infer new core constraints.

## Equi-Propagation

- Infer equalities between literals and constants
- Apply substitution to remove equated literals
- E.g. $D(x)=[0 . .4], D(y)=[0 . .4]$
- Order encoding
$-[x 1, x 2, x 3, x 4] \quad[y 1, y 2, y 3, y 4] \quad v i=(v \geq i)$
- Constraint $y \neq 2$
$-\mathrm{y} 2=\mathrm{y} 3$
- Constraint $x+y=3$
$-\mathrm{x} 4=0, \mathrm{y} 4=0, \mathrm{y} 3=!\mathrm{x} 1, \mathrm{y} 2=!\mathrm{x} 2, \mathrm{y} 1=!\mathrm{x} 3$
$-[x 1, x 1, x 3,0] \quad[-x 3,-x 1,-x 1,0]$
- Constraint $3 x+4 z+9 t \geq 3$


## Ben-Gurion Equi-Propagation Encoder

- BEE encoder
- Translates high level instance to CNF
- Integers represented by order/value/binary encoding
- Equi propagation by
- Adhoc rules per constraint type
- fast, precise in practice
- Complete equi-propagation using SAT (?)
- And adhoc partial evaluation rules


## BEE Comparisons

- Balanced Incomplete Block Design
- Compared with
- Sugar (CSP encoder)
- BEE minus equi-propagation + SatELite

| instance | BEE (SymB) |  |  | Sugar (SymB) |  |  | SatELite (SymB) |  |  |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| $[v, b, r, k, \lambda]$ | $\begin{gathered} \text { comp } \\ (\mathrm{sec}) \end{gathered}$ | clauses | $\begin{aligned} & \text { SAT } \\ & (\mathrm{sec}) \end{aligned}$ | $\begin{gathered} \text { comp } \\ (\mathrm{sec}) \end{gathered}$ | clauses | $\begin{aligned} & \text { SAT } \\ & (\mathrm{sec}) \end{aligned}$ | $\begin{gathered} \text { comp } \\ (\mathrm{sec}) \end{gathered}$ | clauses | $\begin{aligned} & \text { SAT } \\ & (\mathrm{sec}) \end{aligned}$ |
| [7, 420, 180, 3, 60] | 1.65 | 698579 | 1.73 | 12.01 | 2488136 | 13.24 | 1.67 | 802576 | 2.18 |
| [7, 560, 240, 3, 80] | 3.73 | 1211941 | 13.60 | 11.74 | 2753113 | 36.43 | 2.73 | 1397188 | 5.18 |
| $[12,132,33,3,6]$ | 0.95 | 180238 | 0.73 | 83.37 | 1332241 | 7.09 | 1.18 | 184764 | 0.57 |
| $[15,45,24,8,12]$ | 0.51 | 116016 | 8.46 | 4.24 | 466086 | $\infty$ | 0.64 | 134146 | $\infty$ |
| $[15,70,14,3,2]$ | 0.56 | 81563 | 0.39 | 23.58 | 540089 | 1.87 | 1.02 | 79542 | 0.20 |
| [16, 80, 15, 3, 2] | 0.81 | 109442 | 0.56 | 64.81 | 623773 | 2.26 | 1.14 | 105242 | 0.35 |
| [19, 19, 9, 9, 4] | 0.23 | 39931 | 0.09 | 2.27 | 125976 | 0.49 | 0.4 | 44714 | 0.09 |
| $[19,57,9,3,1]$ | 0.34 | 113053 | 0.17 | $\infty$ | - | - | 10.45 | 111869 | 0.14 |
| [21, 21, 5, 5, 1] | 0.02 | 0 | 0.00 | 31.91 | 3716 | 0.01 | 0.01 | 0 | 0.00 |
| $[25,25,9,9,3]$ | 0.64 | 92059 | 1.33 | 42.65 | 569007 | 8.52 | 1.01 | 97623 | 8.93 |
| [25,30, 6, 5, 1] | 0.10 | 24594 | 0.06 | 16.02 | 93388 | 0.42 | 1.2 | 23828 | 0.05 |
| Total (sec) |  | 36.66 |  |  | $>722.93$ |  |  | > 219.14 |  |

## BEE Comparison

- Applying SatELite on output of BEE
- YIKES!
- Doesn't shrink much, usually solves slower

| instance |  | BEE |  |  |  | $\Delta$ SatELite |  |  |  |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
|  |  | $\begin{gathered} \text { comp } \\ (\mathrm{sec}) \end{gathered}$ | clauses | vars | $\begin{aligned} & \hline \text { SAT } \\ & (\mathrm{sec}) \end{aligned}$ | $\begin{gathered} \text { comp } \\ (\mathrm{sec}) \end{gathered}$ | clauses | vars | $\begin{aligned} & \text { SAT } \\ & (\mathrm{sec}) \end{aligned}$ |
| $K_{8}$ | 143 | 0.51 | 248558 | 5724 | 1.26 | 2.60 | 248250 | 5452 | 0.98 |
|  | 142 | 0.27 | 248414 | 5716 | 10.14 | 2.59 | 248107 | 5445 | 3.22 |
|  | 141 | 0.20 | 248254 | 5708 | 7.64 | 2.59 | 247947 | 5437 | 32.81 |
|  | 140 | 0.19 | 248078 | 5700 | 14.68 | 2.60 | 247771 | 5429 | 3.50 |
|  | 139 | 0.18 | 247886 | 5692 | 25.6 | 2.59 | 247579 | 5421 | 6.18 |
|  | 138 | 0.18 | 247678 | 5684 | 12.99 | 2.60 | 247371 | 5413 | 12.18 |
|  | 137 | 0.18 | 247454 | 5676 | 22.91 | 2.59 | 247147 | 5405 | 77.16 |
|  | 136 | 0.18 | 247214 | 5668 | 14.46 | 2.59 | 246907 | 5397 | 97.69 |
|  | 135 | 0.18 | 246958 | 5660 | 298.54 | 2.58 | 246651 | 5389 | 705.48 |
|  | 134 | 0.18 | 246686 | 5652 | 331.8 | 2.59 | 246379 | 5381 | $\infty$ |

## BEE Highlights

- Extremal Graph Theory
- Extremely challenging combinatorics problems
- Find the largest number of edges for a simple graph with $n$ nodes and no 3 or 4 cycles: $f_{4}(n)$
- Huge amount of symmetry
- BEE solution
- Encode advanced symmetry breaking constraints
- Discovers two new values
- $f_{4}(31)=80, f_{4}(32)=85$
- BEE is best where the initial problem and constraints fix/identify many variables


## Outline

- Modelling and solving
- Propagation based solving
- The advantages of keeping structure
- Better (static) CNF encoding
- Dynamic choice: propagation versus CNF encoding
- Propagation with learning (Lazy Clause Generation)
- MiniZinc
- Conclusion


## Propagation vs CNF Encoding



## Solution

## Which is better?

- 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 $\leq 3$
- Propagator
- If 3 of $\{x 1, \ldots, x 8\}$ are true, set the rest false.
- Encoding
- Cardinality or sorting network:
- z21 = z33 = z34 = z35 = z36 = 0



## Comparison: Encoding vs Propagation

- A (theory) propagator
- Lazily generates an encoding
- This encoding is partially stored in nogoods
- The encoding uses no auxiliary Boolean variables
$-\Sigma_{\mathrm{i}=1 . . \mathrm{n}} x_{i} \leq k$ generates $(n-k)^{n} \mathrm{C}_{k}=\mathrm{O}\left(n^{k}\right)$ explanations
- If the problem is UNSAT (or optimization)
- CP solver runtime $\geq$ size of smallest resolution proof
- Cannot decide on auxiliary variables
- Exponentially larger proof
- Compare $\Sigma_{\mathrm{i}=1 . . \mathrm{n}} x_{i} \leq k$ encoding is $\mathrm{O}\left(n \log ^{2} k\right)$
- 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


## Lazy Encoding



## Lazy Encoding results

- MSU4 results

|  | $<10 \mathbf{s}$ | $<\mathbf{3 0 s}$ | $<\mathbf{6 0 s}$ | $<120$ s | $<300$ s | $<\mathbf{6 0 0}$ |
| :--- | ---: | ---: | ---: | ---: | ---: | ---: |
| Encoding | 5374 | $\mathbf{5 5 2 5}$ | 5578 | 5621 | 5659 | 5677 |
| Propagation | 4322 | 4530 | 4603 | 4667 | 4737 | 4767 |
| Lazy | $\mathbf{5 2 2 2}$ | 5479 | $\mathbf{5 5 8 5}$ | $\mathbf{5 6 3 6}$ | $\mathbf{5 6 6 6}$ | $\mathbf{5 6 7 9}$ |
| Encoding |  |  |  |  |  |  |

- Tomography

|  | $<\mathbf{1 0 s}$ | $<\mathbf{3 0 s}$ | $\mathbf{< 6 0 s}$ | $<\mathbf{1 2 0 s}$ | $\boldsymbol{<} \mathbf{3 0 0 s}$ | $<\mathbf{6 0 0}$ |
| :--- | ---: | ---: | ---: | ---: | ---: | ---: |
| Encoding | 773 | 1112 | 1314 | 1501 | 1759 | 1932 |
| Propagation | 1457 | 1748 | 1858 | 1962 | $\mathbf{2 0 1 4}$ | $\mathbf{2 0 2 1}$ |
| Lazy | $\mathbf{1 5 5 6}$ | $\mathbf{1 8 1 8}$ | $\mathbf{1 9 3 5}$ | $\mathbf{1 9 7 1}$ | 2012 | $\mathbf{2 0 2 1}$ |
| Encoding |  |  |  |  |  |  |

## Lazy Encoding

- Keep the structure during solving
- Use the structure to decide on solving method
- Almost always equals or exceeds the best of
- Propagation
- Encoding
- Obvious advantages when
- Some constraints are not/rarely involved in failure
- These are never encoded


## Outline

- Modelling and solving
- Propagation based solving
- The advantages of keeping structure
- Better (static) CNF encoding
- Dynamic choice: propagation versus CNF encoding
- Propagation with learning (Lazy Clause Generation)
- MiniZinc
- Conclusion


## 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
- They become "theory propagators"


## LCG in a Nutshell

- Integer variable $x$ in $I . . u$ encoded as Booleans
$-[x \leq d]$, d in I..u-1
$-[x=d]$, d in l..u
- Dual representation of domain $D(x)$
- Restrict to atomic changes in domain (literals)
$-x \leq d \quad$ (itself)
$-x \geq d \quad![x \leq d-1]$ use $[x \geq d]$ as shorthand
$-x=d \quad$ (itself)
$-x \neq d \quad![x=d]$ use $[x \neq d]$ as shorthand
- Propagation is clause generation
- e.g. $[x \leq 2]$ and $x \geq y$ means that $[y \leq 2]$
- clause $[x \leq 2] \rightarrow[y \leq 2]$


## (Original) LCG propagation example

- Variables: $\{x, y, z\} D(v)=[0 . .6]$ Boolean bs
- Constraints:

$$
\begin{aligned}
& -z \geq y, b \rightarrow y \neq 3, c \rightarrow y \geq 3, c \rightarrow x \geq 6 \\
& -4 x+10 y+5 z \leq 71 \text { (lin) }
\end{aligned}
$$

- Execution


1UIP nogood: c $\wedge[y \neq 3] \rightarrow$ false or $[y \neq 3] \rightarrow$ !c

## LCG propagation example

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

$$
\begin{aligned}
& -z \geq y, b \rightarrow y \neq 3, c \rightarrow y \geq 3, c \rightarrow x \geq 6 \\
& -4 x+10 y+5 z \leq 71 \text { (lin) }
\end{aligned}
$$

- Backtrack


1UIP nogood: c $\wedge[y \neq 3] \rightarrow$ false or $[y \neq 3] \rightarrow$ !c

## 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 propagation example

- Execution



## 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

$$
\begin{array}{cc}
{[x \geq 5] \quad \operatorname{lin}} & x \geq 5 \rightarrow y \leq 3 \\
\cdots \cdots \cdots \cdots \cdots \cdots \cdots \cdots \cdots \cdots \\
{[y \leq 5]} & {[y \leq 3]}
\end{array}
$$

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

## 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


## Lessons from LCG

- Lazy literal generation
- Integer variable representation is generated only as needed
- Encoding can be bad
- Even without the size blowup
- Programmed search
- For (many) problems default activity search is bad
- typically where we cannot prove optimality


## Lazy Literal Generation

- For constraint problems over large domains lazy literal generation is crucial

|  | amaze | fastfood | filters | league | mspsp | nonogram | patt-set |
| :--- | ---: | ---: | ---: | ---: | ---: | ---: | ---: |
| Initial | 8690 | 1043 k | 8204 | 341 k | 13534 | 448 k | 19916 |
| Root | 6409 | 729 k | 6944 | 211 k | 9779 | 364 k | 19795 |
| Created | $\mathbf{2 2 1 4}$ | $\mathbf{9 8 3 1}$ | $\mathbf{1 3 1 0}$ | $\mathbf{9 6 7}$ | $\mathbf{6 8 3 2}$ | $\mathbf{2 6 2 k}$ | $\mathbf{1 5 4 9 0}$ |
| Percent | $34 \%$ | $1.3 \%$ | $19 \%$ | $0.45 \%$ | $70 \%$ | $72 \%$ | $78 \%$ |


|  | proj-plan | radiation | shipshed | solbat | still-life | tpp |
| :--- | ---: | ---: | ---: | ---: | ---: | ---: |
| Initial | 18720 | 145 k | 2071 k | 12144 | 18947 | 19335 |
| Root | 18478 | 43144 | 2071 k | 9326 | 12737 | 18976 |
| Created | 5489 | 1993 | $\mathbf{1 2 9 4 3}$ | 10398 | $\mathbf{3 6 6 6}$ | $\mathbf{9 2 3 2}$ |
| Percent | $30 \%$ | $4.6 \%$ | $0.62 \%$ | $111 \%$ | $29 \%$ | $49 \%$ |

## Encoding versus Propagation

- Propagation can be superior
- Even if the encoding propagates as strongly
- And its size complexity is no higher than the propagator
- Example: multi-decision diagrams ( $n$ nodes)
- SAT encoding of MDD propagates equivalent (no bigger O(nd))
- Propagator uses structure of MDD (faster propagation)
- Intermediate variables don't help search (even though its VSIDS)

| $\mathbf{n}$ | Tseitin | fails | Equiv | fails | MDD | fails |
| ---: | ---: | ---: | ---: | ---: | ---: | ---: |
| 14 | 75.24 | 331 k | 24.39 | 63 k | $\mathbf{5 . 5 9}$ | $\mathbf{5 1 k}$ |
| 15 | 366.03 | 1128 k | 67.59 | 148 k | $\mathbf{7 . 8 6}$ | $\mathbf{6 5 k}$ |
| 16 | --- | --- | 82.88 | 148 k | $\mathbf{1 8 . 0 3}$ | $\mathbf{1 2 3 k}$ |
| 17 | --- | --- | 183.28 | 276 k | $\mathbf{6 8 . 3 2}$ | 381 k |
| 18 | --- | --- | 392.91 | 445 k | $\mathbf{1 0 1 . 3 1}$ | 500 k |
| 19 | --- | --- | -- | --- | $\mathbf{1 1 8 . 1 6}$ | $\mathbf{5 3 8 k}$ |
| 20 | --- | --- | -- | --- | $\mathbf{3 8 4 . 9 9}$ | $\mathbf{1 3 4 1 k}$ |

Activity-based search is 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
- VSIDS

|  | Static | DomWDeg | Impact | VSIDS |
| :--- | ---: | ---: | ---: | ---: |
| Time (s) | 206.3 | 0.8 | 951.3 | 1522.2 |
| Solved (70) | 66 | 70 | 55 | 47 |

## 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


## 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 CPLEX, Gurobi, SCIP (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
- SMT based approach (fzn2smt) Fourth behind Gecode
- Illustrates that the approach is strongly beneficial on a wide range of problems

Outline

- Modelling and solving
- Propagation based solving
- The advantages of keeping structure
- Better (static) CNF encoding
- Dynamic choice: propagation versus CNF encoding
- Propagation with learning (Lazy Clause Generation)
- MiniZinc
- Conclusion


## MiniZinc

- A solver independent modelling language for combinatorial optimization problems
- Open source, developed since 2007
- Closest thing to a Constraint Programming standard
- Domains: Booleans, integers, floats, sets of integers
- Globals:
- User defined predicates + functions
- Reflection functions
- Customizable library of global constraint definitions
- Features
- Annotations for adding non-declarative information


## MiniZinc Example: Jobshop Scheduling

int: $n$; set of int: Job=1..n; \% no of jobs int: m; set of int: Task=1..m; \% task per job
int: span;
\% max end time
array[Job,Task] of int: d;
array[Job,Task] of Task: mc;
array[Job,Task] of var 0..span: s;
constraint forall(i in Job, $j$ in 1..m-1)
(s[i,j] $+d[i, j]<=s[i, j+1]) ;$
constraint forall(k in Task)

$$
\begin{aligned}
\text { (unary([s[i,j] } \mid & i \text { in Job, } j \text { in Task } \\
& \text { where mc }[i, j]=k], \\
{[d[i, j] \mid} & i \text { in Job, } j \text { in Task } \\
& \text { where mc[i,j] }=k])) ;
\end{aligned}
$$

var int: obj $=\max ([s[i, m]+d[i, m] \mid i \operatorname{in~Job]);~}$ solve minimize obj;

## MiniZinc Example: Jobshop Scheduling

 int: $n$; set of int: Job=1..n; \% no of jobs int: m; set of int: Task=1..m; \% task int: span;array[Job,Task] of int: d;
array[Job,Task] of Task: mc;
array [Job, Task] of var 0..span: s;
constraint forall(i in Job, $j$ in 1..m-1)
(s[i,j] $+d[i, j]<=s[i, j+1]) ;$
constraint forall(k in Task)

$$
\begin{aligned}
\text { (unary([s[i,j] } \mid & i \text { in Job, } j \text { in Task } \\
& \text { where mc }[i, j]=k], \\
{[d[i, j] \mid} & i \text { in Job, } j \text { in Task } \\
& \text { where mc[i,j] } m b)) ;
\end{aligned}
$$

var int: obj $=\max ([s[i, m]+d[i, m] \mid i \operatorname{in~Job]);~}$ solve minimize obj;

## MiniZinc Example: Jobshop Scheduling

```
int: n; set of int: Job=1..n; % no r
int: m; set of int: Task=1..m; % task
int: span;
Dependent
Parameters
array[Job,Task] of int: d;
array[Job,Task] of Task: mc;
array[Job,Task] of var 0..span: s;
constraint forall(i in Job, j in 1..m-1)
    (s[i,j] + d[i,j] <= s[i,j+1]);
constraint forall(k in Task)
```

```
\[
\begin{aligned}
& \text { (unary([s[i,j] | i in Job, j in Task } \\
& \text { where mc[i,j] = k], } \\
& \text { [d[i,j] | i in Job, j in Task } \\
& \text { where mc[i,j] = k])); }
\end{aligned}
\]
var int: obj \(=\max ([s[i, m]+d[i, m] \quad i \quad i n ~ J o b]) ;\) solve minimize obj;
```


## MiniZinc Example: Jobshop Scheduling

```
int: n; set of int: Job=1..n; % no c
int: m; set of int: Task=1..m; % task
int: span;
Dependent
Parameters
array[Job,Task] of int: d;
array[Job,Task] of Task: mc;
array[Job,Task] of var 0..span: s;
```


## Variables

```
constraint forall(i in Job, j in 1..m-1)
(s[i,j] + d[i,j] <= s[i,j+1]);
constraint forall(k in Task)
```

```
\[
\begin{aligned}
& \text { (unary([s[i,j] | i in Job, j in Task } \\
& \text { where mc[i,j] = k], } \\
& \text { [d[i,j] | i in Job, j in Task } \\
& \text { where mc[i,j] = k])); }
\end{aligned}
\]
var int: obj = max([s[i,m] + d[i,m] | i in Job]); solve minimize obj;
```


## MiniZinc Example: Jobshop Scheduling

```
int: n; set of int: Job=1..n; % noc Dependent
int: m; set of int: Task=1..m; % task
int: span;
Parameters
array[Job,Task] of int: d;
array[Job,Task] of Task: mc;
array[Job,Task] of var 0..span: s;
```


## Variables

```
constraint forall(i in Job, j in 1..m-1)
    (s[i,j] + d[i,j] <= s[i,j+1]);
constraint forall(k in Task)
    (unary([s[i,j] | i in Job, j in Task
                                    where mc[i,j] = k],
    [d[i,j] | i in Job, j in Task
    where mc[i,j] = k]));
var int: obj = max([s[i,m] + d[i,m] | i in Job]);
```

solve minimize obj;

## MiniZinc Example: Jobshop Scheduling

```
int: n; set of int: Job=1..n; % no c Dependent
int: m; set of int: Task=1..m; % task
int: span;
Parameters
array[Job,Task] of int: d;
array[Job,Task] of Task: mc;
array[Job,Task] of var 0..span: s;
```


## Variables

```
constraint forall(i in Job, j in 1..m-1)
(s[i,j] + d[i,j] <= s[i,j+1]);
```

constraint forall(k in Task)
(unary([s[i,j] | i in Job, j in Task
where mc[i,j] = k],
Constraints
[d[i,j] | i in Job, j in Task
where mc[i,j] = k]));
var int: obj = max([s[i,m] + d[i,m] | i in Job]);
solve minimize obj;

## MiniZinc Example: Jobshop Scheduling

int: n; set of int: Job=1..n; \% no c Dependent
int: $m$; set of int: Task=1..m; \% task
int: span;
array[Job,Task] of int: d;
array[Job,Task] of Task: mc;
array[Job,Task] of var 0..span: s;

## Variables

 constraint forall(i in Job, j in 1..m-1)$$
(s[i, j]+d[i, j]<=s[i, j+1]) ;
$$

constraint forall(k in Task)
Global
where mc[i,j] = k],
Constraints
[d[i,j] | i in Job, j in Task where $\operatorname{mc}[i, j]=k]$ ));
var int: 0 bj $=\max ([s[i, m]+d[i, m] \mid i \operatorname{in~Job]);~}$ solve minimize obj;

## MiniZinc Example: Jobshop Scheduling

int: n; set of int: Job=1..n; \% no r Dependent
int: $m$; set of int: Task=1..m; \% task
int: span;
array[Job,Task] of int: d;
array[Job,Task] of Task: mc;
array[Job,Task] of var 0..span: s;

## Variables

 constraint forall(i in Job, j in 1..m-1)$$
(s[i, j]+d[i, j]<=s[i, j+1]) ;
$$

constraint forall(k in Task)
Global

Objective where mc[i,j] = k],

Constraints
j] | i in Job, j in Task
where mc[i,j] = k]));
var int: $\mathrm{b}_{\mathrm{bj}}=\max ([\mathrm{s}[i, \mathrm{~m}]+\mathrm{d}[i, \mathrm{~m}]$ | i in Job]); solve minimize obj;

## MiniZinc Example

- Separate data file

```
n = 2; m = 2; span = 10;
d = [|3,5|6,2|]; mc = [||,2|2,1|];
```

- Flattened to FlatZinc

```
array[1..4] of var 0..10: s
var 5..15: obj;
int_lin_le([1, -1], [s[1], s[2]], -3);
int_lin_le([1, -1], [s[3], s[4]], -6);
unary([s[1],s[4]],[3,2]);
unary([s[2],s[3]],[5,6]);
int_maximum([I1,I2],obj);
var 5..15: I1; var 5..15: I2;
int_lin_eq([-1,1],[I1,s[2]],-5);
int_lin_eq([-1,1],[I2,s[4]],-2);
```


## User-defined constraint treatment

- Solver dependent rewriting
- E.g. replacing unary global by non-overlap disjunction predicate unary(array[int] of var int:s;
array[int] of int:d) =
forall(i,j in index_set(s) where $i<j)$

$$
(s[i]+d[i]<=s[j] \quad \backslash / s[j]+d[j]<=s[i]) ;
$$

- Critical to support by many solvers
- CP solvers: Gecode, Eclipse, SICStus Prolog, Bprolog, Choco, Mistral, Jacop, izplus, Chuffed, CPX, lazyfd, g12-fd
- MIP solvers: SCIP, Cplex, Gurobi, Coin-OR-CBC
- SAT + SMT Solvers: fzntini, bee, minisatID, fzn2smt


## libmzn

- A new open source framework: LLVM like
- Direct interface to solvers and C++ API
- Specialist transformations
- Booleanization
- Linearization
- A good modelling language for
- SAT +
- SMT solvers
- Release
- September 2013



## Conclusions

- Combinatorial problems often include
- Substantial and well understood substructures
- Modelling should
- allow these substructures to be expressed
- Solving should
- allow these substructures to be taken advantage of
- Taking note of substructures can:
- Improve design models (better translation)
- Allow use to choose between encoding and propagation
- Create powerful dynamic encodings


## The Hard Word

- If you want to compete with all optimization technology
- Competition is on a high level model, not CNF
- Then ignoring the structure
- Will not compete!
- So remember

There are no CNF problems

## The future directions

- Details of how modern LCG solvers work - www.cs.mu.oz.au/~pjs/papers/cpx.pdf
- More about MiniZinc
- www.minizinc.org
- More about BEE
- http://amit.metodi.me/research/bee/
- Structure-based extended resolution
- Advantages of encoding + propagation simultaneously
- http://arxiv.org/abs/1306.4418
- Unsatisfiable cores for constraint programming
- Easy to translate UNSAT core methods from SAT
- http://arxiv.org/abs/1305.1690

