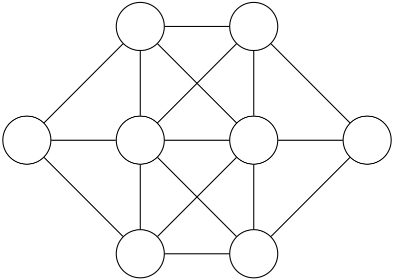
Your First Constraint Programming Puzzle



- Place each of the numbers 1 to 8 in circles.
- Adjacent circles can't have consecutive numbers.

A Constraint Programming Solver You Can Trust (But Don't Have To)

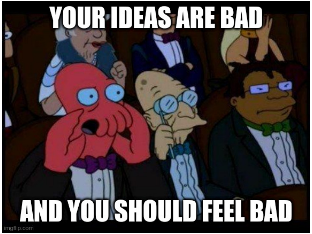
Ciaran McCreesh

University of Glasgow

Royal Academy of Engineering

Heckling Encouraged

- This talk is an overview of what I plan to do for the next five years.
- I'm not heavily committed to many of the details.



The Constraint Programming Process

Real world problem

$\downarrow$ Modelling

High level model (Essence, MiniZinc)

$\downarrow$ Compilation

Low level model (CP, MIP, SAT, ...)

$\downarrow$ Preprocessing

Solver's internal representation

$\downarrow$ Solving

Solution

?

A Very Realistic Real World Problem

- You have eight exams to schedule over eight days, one exam per day.
- Students can’t have an exam two days in a row.
- Some students take both of subjects 1 and 2, 1 and 3, 1 and 4, 1 and 5, 2 and 4, …

A Very Realistic Real World Problem

THE CONVERSATION

Academic rigour, journalistic flair

Search analysis, research, academics...

COVID-19 Arts + Culture Business + Economy Education Environment + Energy Health + Medicine Politics + Society Science + Technology COP26

What problems will AI solve in future? An old British gameshow can help explain

November 3, 2015 1.17pm GMT

Original Crystal Maze presenter Richard O'Brien. Adam Butler/PA

The Crystal Maze, the popular UK television show from the early 1990s, included a puzzle that is very useful for explaining one of the main conundrums in artificial intelligence. The puzzle appeared a few times in the show's Futuristic

Authors

Ian Miguel
Professor of Computer Science, University of St Andrews

Patrick Prosser
Senior Lecturer in Computer Science, University of Glasgow

Disclosure statement

Ian receives research funding from the EPSRC and the Royal Academy of Engineering. He is Director of the Graduate Academy of the Scottish Informatics and Computer Science Alliance and on the board of the Data Lab innovation centre.

Patrick Prosser does not work for, consult, own shares in or receive funding from any company or organisation that would benefit from this article, and has disclosed no relevant affiliations beyond their academic appointment.

Partners

University of Glasgow

University of St Andrews

University of Glasgow and University of St Andrews provide funding as members of The Conversation UK.

The Conversation UK receives funding from these organisations

A High Level Model

```
include "globals.mzn";

int: n = 8;
array[1..n] of var 1..n: xs;

int: m = 17;
array[1..m, 1..2] of 1..n: edges =
[| 1, 2 | 1, 3 | 1, 4 | 1, 5
 | 2, 4 | 2, 5 | 2, 6 | 3, 4
 | 3, 7 | 4, 5 | 4, 7 | 4, 8
 | 5, 6 | 5, 7 | 5, 8 | 6, 8 | 7, 8 |];

constraint (alldifferent(xs));

constraint forall (e in 1..m) (
    abs(xs[edges[e, 1]] - xs[edges[e, 2]]) != 1);
```

A High Level Model

```
$ minizinc --solver org.gecode.gecode -a crystalmaze.mzn
xs = array1d(1..8, [5, 3, 2, 8, 1, 7, 6, 4]);
----------
xs = array1d(1..8, [6, 4, 2, 8, 1, 7, 5, 3]);
----------
xs = array1d(1..8, [3, 5, 7, 1, 8, 2, 4, 6]);
----------
xs = array1d(1..8, [4, 6, 7, 1, 8, 2, 3, 5]);
----------
==========
```

Using a Solver Directly

```
Problem p;

vector<IntegerVariableID> xs;
for (int i = 0 ; i < 8 ; ++i)
    xs.push_back(p.create_integer_variable(1_i, 8_i));

vector<pair<int, int> > edges{ { 0, 1 }, { 0, 2 }, { 0, 3 }, { 0, 4 },
    { 1, 3 }, { 1, 4 }, { 1, 5 }, { 2, 3 }, { 2, 6 }, { 3, 4 }, { 3, 6 },
    { 3, 7 }, { 4, 5 }, { 4, 6 }, { 4, 7 }, { 5, 7 }, { 6, 7 } };

for (auto & [ x1, x2 ] : edges) {
    auto diff = p.create_integer_variable(-7_i, 7_i);
    p.post(Minus{ xs[x1], xs[x2], diff });
    p.post(NotEquals{ diff, 0_c });
    p.post(NotEquals{ diff, 1_c });
    p.post(NotEquals{ diff, -1_c });
}

p.post(AllDifferent{ xs });
p.branch_on(xs);
solve(p, [&] (const State & s) -> bool {
    cout << "␣␣" << s(xs[0]) << "␣" << s(xs[1]) << endl;
    cout << s(xs[2]) << "␣" << s(xs[3]) << "␣" << s(xs[4]) << "␣" << s(xs[5]) << endl;
    cout << "␣␣" << s(xs[6]) << "␣" << s(xs[7]) << endl << endl;
    return true;
});
```

Using a Solver Directly

```
$ ./crystal_maze
  3 5
7 1 8 2
  4 6

  4 6
7 1 8 2
  3 5

  5 3
2 8 1 7
  6 4

  6 4
2 8 1 7
  5 3
```

How Solvers Work

- Variables are a set of non-deleted values.
- Inference from each constraint.
- Propagation until we can't do inference.
- Backtracking search.

The Inconvenient Secret

- For somewhere between 0.1% (my clique experiments) and 1.28% (MiniZinc challenge 2021) of instances, we get the wrong solution.
  - False claims of unsatisfiability.
  - False claims of optimality.
  - Infeasible solutions produced.
  - The same solver run on the same instance on the same hardware twice in a row can claim both unsatisfiability and satisfiability.
- This includes academic and commercial CP and MIP solvers.
- Extensive testing hasn't fixed this.
- Formal methods are far from being able to handle solvers.
- The situation for SAT solvers is somewhat better.

Proof Logging in SAT

- Solvers must produce independently-verifiable proofs.
- Seems to reduce bugs, rather than just catching them.
- Vital for social acceptability of computer-generated maths.
- Most of the focus is on unsatisfiability.

Proof Logging in SAT

COUNTEREXAMPLE TO EULER'S CONJECTURE ON SUMS OF LIKE POWERS

BY L. J. LANDER AND T. R. PARKIN

Communicated by J. D. Swift, June 27, 1966

A direct search on the CDC 6600 yielded

$$27^5 + 84^5 + 110^5 + 133^5 = 144^5$$

as the smallest instance in which four fifth powers sum to a fifth power. This is a counterexample to a conjecture by Euler [1] that at least n nth powers are required to sum to an nth power, $n > 2$.

REFERENCE

1. L. E. Dickson, *History of the theory of numbers*, Vol. 2, Chelsea, New York, 1952, p. 648.

Proof Logging in SAT

2 JUNE 2016 | VOL 534 | NATURE | 17

MATHEMATICS

Maths proof smashes size record

Supercomputer produces a 200-terabyte proof — but is it really mathematics?

BY EVELYN LAMB

Three computer scientists have announced the largest-ever mathematical proof: a file that comes in at a whopping 200 terabytes, equivalent to all the digitized text held by the US Library of Congress. The researchers have created[1] a 68-gigabyte compressed version of their solution — which would allow anyone with about 30,000 hours of spare processor time to download, reconstruct and verify it — but a human could never hope to read through it.

Computer-assisted proofs too large to be directly verifiable by humans have become common, as have computers that solve problems in combinatorics — the study of finite discrete structures — by checking through umpteen individual cases. Still, "200 terabytes is unbelievable", says Ronald Graham, a mathematician at the University of California, San Diego. The previous record-holder is thought to be a 13-gigabyte proof[2], published in 2014.

The puzzle that required the 200-terabyte proof, called the Boolean Pythagorean triples problem, has troubled mathematicians for decades. In the 1980s, Graham offered a prize of US$100 for anyone who could solve it. (He presented the cheque to one of the three computer scientists, Marijn Heule of the University of Texas at Austin, last month.) The problem asks whether it is possible to colour each positive integer either red or blue, so that no trio of integers a, b and c that satisfy Pythagoras' famous equation $a^2 + b^2 = c^2$ are all the same colour. For example, for the Pythagorean triple 3, 4 and 5, if 3 and 5 were blue, 4 would have to be red.

Resolution Proofs

Model axioms From the input

Resolution

$$\frac{x_1 \lor x_2 \lor \ldots \lor x_i \lor c \qquad \overline{c} \lor y_1 \lor y_2 \lor \ldots y_j}{x_1 \lor x_2 \lor \ldots \lor x_i \lor y_1 \lor y_2 \lor \ldots \lor y_j}$$

- To prove unsatisfiability: resolve until you reach the empty clause.

Resolution Proofs

$x \vee y \vee z$	(1)
$\overline{x} \vee \overline{y} \vee \overline{z}$	(2)
$\overline{x} \vee y$	(3)
$\overline{x} \vee z$	(4)
$x \vee \overline{y}$	(5)
$x \vee \overline{z}$	(6)

1, 5 on y	$x \vee z$	(7)
6, 7 on z	x	(8)
3, 8 on x	y	(9)
4, 8 on x	z	(10)
2, 8 on x	$\overline{y} \vee \overline{z}$	(11)
9, 11 on y	$\overline{z}$	(12)
10, 12 on z	$\emptyset$	(13)

Equisatisfiability and Completeness

- Start with the constraints we're given.
- At each step in a proof, add a new constraint which obviously doesn't affect satisfiability.
- If we can derive contradiction, there were no solutions to the original problem.
- Using resolution, we can always do this for any unsatisfiable SAT problem.

Reverse Unit Propagation Proofs

- Unit propagation:
  - Look for a clause containing just one literal ℓ.
  - Delete $\overline{\ell}$ from every other clause.
  - Repeat until you can't do anything.
- Reverse unit propagation:
  - Add the negation of a constraint C, and unit propagate.
  - If contradiction is reached, derive C.
- Can rewrite to resolution in polynomial time.

Backtracking Search as RUP

- Every time you backtrack, output a RUP step for the sequence of guesses you just made.

Backtracking Search as RUP

$$
\begin{aligned}
&x \lor y \lor z && (1)\\
&\overline{x} \lor \overline{y} \lor \overline{z} && (2)\\
&\overline{x} \lor y && (3)\\
&\overline{x} \lor z && (4)\\
&x \lor \overline{y} && (5)\\
&x \lor \overline{z} && (6)
\end{aligned}
$$

Backtracking Search as RUP

$$x \lor y \lor z \quad (1)$$
$$\overline{x} \lor \overline{y} \lor \overline{z} \quad (2)$$
$$\overline{x} \lor y \quad (3)$$
$$\overline{x} \lor z \quad (4)$$
$$x \lor \overline{y} \quad (5)$$
$$x \lor \overline{z} \quad (6)$$

$$\text{RUP } x \quad (7)$$

Backtracking Search as RUP

$$x \lor y \lor z \quad (1)$$
$$\overline{x} \lor \overline{y} \lor \overline{z} \quad (2)$$
$$\overline{x} \lor y \quad (3)$$
$$\overline{x} \lor z \quad (4)$$
$$x \lor \overline{y} \quad (5)$$
$$x \lor \overline{z} \quad (6)$$

RUP x (7)

$\overline{x}$ assumed

$\overline{y}$ from 5

$\overline{z}$ from 6

x from 1

Backtracking Search as RUP

$$x \vee y \vee z \quad (1)$$
$$\overline{x} \vee \overline{y} \vee \overline{z} \quad (2)$$
$$\overline{x} \vee y \quad (3)$$
$$\overline{x} \vee z \quad (4)$$
$$x \vee \overline{y} \quad (5)$$
$$x \vee \overline{z} \quad (6)$$

$$\text{RUP } x \quad (7)$$
$$\text{RUP } \overline{x} \quad (8)$$

Backtracking Search as RUP

$$x \vee y \vee z \qquad (1)$$

$$\overline{x} \vee \overline{y} \vee \overline{z} \qquad (2)$$

$$\overline{x} \vee y \qquad (3)$$

$$\overline{x} \vee z \qquad (4)$$

$$x \vee \overline{y} \qquad (5)$$

$$x \vee \overline{z} \qquad (6)$$

RUP x (7)

RUP $\overline{x}$ (8)

x assumed

y from 3

z from 4

$\overline{x}$ from 2

Backtracking Search as RUP

$$x \lor y \lor z \tag{1}$$
$$\overline{x} \lor \overline{y} \lor \overline{z} \tag{2}$$
$$\overline{x} \lor y \tag{3}$$
$$\overline{x} \lor z \tag{4}$$
$$x \lor \overline{y} \tag{5}$$
$$x \lor \overline{z} \tag{6}$$

$$\text{RUP } x \tag{7}$$
$$\text{RUP } \overline{x} \tag{8}$$
$$\text{RUP } \emptyset \tag{9}$$

Backtracking Search as RUP

$$x \lor y \lor z \quad (1)$$
$$\overline{x} \lor \overline{y} \lor \overline{z} \quad (2)$$
$$\overline{x} \lor y \quad (3)$$
$$\overline{x} \lor z \quad (4)$$
$$x \lor \overline{y} \quad (5)$$
$$x \lor \overline{z} \quad (6)$$

$$\text{RUP } x \quad (7)$$
$$\text{RUP } \overline{x} \quad (8)$$
$$\text{RUP } \emptyset \quad (9)$$

x from 7

$\overline{x}$ from 8

This Won't Work for Constraint Programming

- "All different" requires exponential length proofs in resolution.
- Internal representation has to closely match the input.
- Also need to consider optimisation, enumeration, satisfiable instances, …

Richer Proof Logs

- Logs could contain every kind of propagation done by every implementation of every constraint?
  - Hard to trust the proof logs.
  - Can't entirely trust certain "proofs" of valid kinds of inference from the literature either…
- Can we make a proof system which is "powerful enough", but also simple?

Extension Variables

- Given a constraint (not necessarily CNF) C and a fresh variable y, introduce

$$y \leftrightarrow C$$

- Now we have polynomial length proofs for "all different".
  - But not necessarily a useful polynomial...

Pseudo-Boolean Models

- A set of $\{0, 1\}$-valued variables x_i, 1 means true.
- Constraints are linear inequalities

$$\sum_i c_i x_i \geq C$$

- Write $\overline{x}_i$ to mean $1 - x_i$.
- Can rewrite CNF to pseudo-Boolean directly,

$$x_1 \lor \overline{x}_2 \lor x_3 \qquad \leftrightarrow \qquad x_1 + \overline{x}_2 + x_3 \geq 1$$

Cutting Planes Proofs

Model axioms From the input

Literal axioms

$$\frac{}{\ell_i \geq 0}$$

Addition

$$\frac{\sum_i a_i \ell_i \geq A \qquad \sum_i b_i \ell_i \geq B}{\sum_i (a_i + b_i)\ell_i \geq A + B}$$

Multiplication
for any $c \in \mathbb{Z}$

$$\frac{\sum_i a_i \ell_i \geq A}{\sum_i c a_i \ell_i \geq cA}$$

Division
for any $c \in \mathbb{N}^+$

$$\frac{\sum_i a_i \ell_i \geq A}{\sum_i \left\lceil \frac{a_i}{c} \right\rceil \ell_i \geq \left\lceil \frac{A}{c} \right\rceil}$$

- Extremely easy and compact proofs for all-different.

Reverse Unit Propagation, Revisited

- Can define RUP similarly for pseudo-Boolean constraints.
- "Unit propagation" is integer bounds consistency.
- It does the same thing on clauses.
- RUP can be rewritten to cutting planes in polynomial time.

VeriPB

/StephanGocht/VeriPB

- MIT licence, written in Python with parsing in C.
- Useful features like tracing and proof debugging.

- Jan Elffers, Stephan Gocht, Ciaran McCreesh, Jakob Nordström: Justifying All Differences Using Pseudo-Boolean Reasoning. AAAI 2020.
- Stephan Gocht, Ciaran McCreesh, Jakob Nordström: Subgraph Isomorphism Meets Cutting Planes: Solving With Certified Solutions. IJCAI 2020.
- Stephan Gocht, Ross McBride, Ciaran McCreesh, Jakob Nordström, Patrick Prosser, James Trimble: Certifying Solvers for Clique and Maximum Common (Connected) Subgraph Problems. CP 2020.
- Stephan Gocht, Jakob Nordström: Certifying Parity Reasoning Efficiently Using Pseudo-Boolean Proofs. AAAI 2021.
- Bart Bogaerts, Stephan Gocht, Ciaran McCreesh, Jakob Nordström: Certified Dominance and Symmetry Breaking for Combinatorial Optimisation. AAAI 2022.

Compiling CP Variables to PB

- A CP variable $X \in \{1, 2, 3\}$ becomes x_1, x_2, x_3.
- Each variable takes exactly one value:

$$\sum_{v \in \mathrm{D}(X)} x_v \geq 1 \qquad \sum_{v \in \mathrm{D}(X)} -1 x_v \geq -1$$

- Questionable design choice: also create a $\geq$ encoding,

$$\begin{aligned} x_{\geq v} &\rightarrow x_{\geq v-1} \\ \overline{x}_{\geq v} &\rightarrow \overline{x}_{\geq v+1} \\ x_v &\rightarrow x_{\geq v} \\ x_v &\rightarrow \overline{x}_{\geq v+1} \\ \overline{x}_v \wedge \overline{x}_{v+1} \wedge \ldots &\rightarrow \overline{x}_{\geq v} \\ x_{\geq v} \wedge \overline{x}_{\geq v+1} &\rightarrow x_v \end{aligned}$$

Compiling CP Variables to PB

CP Model

$$x \in \{1, 2, 3\}$$

Generated OPB Fragment

```
* variable x domain
1 x_1 1 x_2 1 x_3 >= 1 ;
-1 x_1 -1 x_2 -1 x_3 >= -1 ;
* variable x greater or equal encoding
1 x_ge_1 >= 1 ;
1 x_ge_1 1 ~x_ge_2 >= 1 ;
1 ~x_1 1 x_ge_1 >= 1 ;
1 ~x_1 1 ~x_ge_2 >= 1 ;
1 x_1 1 x_2 1 x_3 1 ~x_ge_1 >= 1 ;
1 ~x_ge_1 1 x_ge_2 1 x_1 >= 1 ;
1 ~x_ge_2 1 x_ge_1 >= 1 ;
1 x_ge_2 1 ~x_ge_3 >= 1 ;
1 ~x_2 1 x_ge_2 >= 1 ;
1 ~x_2 1 ~x_ge_3 >= 1 ;
1 x_2 1 x_3 1 ~x_ge_2 >= 1 ;
1 ~x_ge_2 1 x_ge_3 1 x_2 >= 1 ;
1 ~x_ge_3 1 x_ge_2 >= 1 ;
1 ~x_3 1 x_ge_3 >= 1 ;
1 x_3 1 ~x_ge_3 >= 1 ;
```

Compiling Not Equals to PB

- CP variables $X \in \{1, 2, 3\}$ and $Y \in \{2, 3, 4\}$, constraint $X \neq Y$.
- For each value they have in common, we can't pick both:

$$x_2 + y_2 \leq 1 \qquad \text{i.e.} \qquad -1x_2 + -1y_2 \geq -1$$
$$x_3 + y_3 \leq 1 \qquad \text{i.e.} \qquad -1x_3 + -1y_3 \geq -1$$

- In OPB:

```
* not equals x y
-1 x_2 -1 y_2 >= -1 ;
-1 x_3 -1 y_3 >= -1 ;
```

Compiling All-Different

- CP variables $X \in \{1, 2, 3\}$, $Y \in \{2, 3\}$, $Z \in \{2, 3, 4\}$, constraint *alldifferent*$(\{X, Y, Z\})$.
- We could do pairwise not-equals, as in SAT, or…
- For each value, it can be used at most once:

$$\begin{aligned} -1x_1 &\geq -1 \\ -1x_2 + -1y_2 + -1z_2 &\geq -1 \\ -1y_3 + -1z_3 &\geq -1 \\ -1z_4 &\geq -1 \end{aligned}$$

- In OPB:

```
* all different X Y Z
-1 x_2 -1 y_2 -1 z_2 >= -1 ;
-1 y_3 -1 z_3 >= -1 ;
```

Compiling Table

- CP variables take one of a list of feasible tuples:

$$(X, Y, Z) \in \{(1, 2, 3), (1, 3, 4), (2, 1, 1)\}$$

- Encode using a selector variable S:

$$s_1 + s_2 + s_3 = 1$$
$$s_1 \rightarrow x_1 \wedge y_2 \wedge z_3$$
$$s_2 \rightarrow x_1 \wedge y_3 \wedge z_4$$
$$s_3 \rightarrow x_2 \wedge y_1 \wedge z_1$$

The Glasgow Constraint Solver

/ciaranm/glasgow-constraint-solver

- MIT licence, written in fancy modern C++.
- Currently implements the bare minimum needed to give this talk.
- I couldn't think of a name.

Proof = Search + Justified Deletions

- Whenever a variable loses a value, this must be visible to the proof verifier.
- Any constraint where unit propagation gives the same consistency as CP requires no work.
- Can use RUP statements or explicit cutting planes proofs for the rest.

Not Equals

```
auto value1 = state.optional_single_value(v1);
auto value2 = state.optional_single_value(v2);
if (value1 && value2)
    return pair{
        (*value1 != *value2) ? Inf::NoChange : Inf::Contradiction,
        Prop::DisableUntilBacktrack
    };
else if (value1)
    return pair{
        state.infer(v2 != *value1, NoJustification{ }),
        Prop::DisableUntilBacktrack
    };
else if (value2)
    return pair{
        state.infer(v1 != *value2, NoJustification{ }),
        Prop::DisableUntilBacktrack
    };
else
    return pair{ Inf::NoChange, Prop::Enable };
```

Table Constraints

```
// check whether selectable tuples are still feasible
state.for_each_value(table.selector, [&] (Integer tuple_idx) {
    bool is_feasible = /* ... */;

    if (! is_feasible) {
        switch (state.infer(
                    table.selector != tuple_idx,
                    NoJustification{ })) {
            case Inf::NoChange:      break;
            case Inf::Change:        changed = true; break;
            case Inf::Contradiction: contradiction = true; break;
        }
    }
});
```

Table Constraints

```
// check for supports in selectable tuples
for (auto & var : table.vars) {
    state.for_each_value(var, [&] (Integer val) {
        bool supported = /* ... */;

        if (! supported) {
            switch (state.infer(var != val, JustifyUsingRUP{ })) {
                case Inf::NoChange:      break;
                case Inf::Change:        changed = true; break;
                case Inf::Contradiction: contradiction = true; break;
            }
        }
    });

    if (contradiction)
        return pair{ Inf::Contradiction, Prop::Enable };
}
```

Linear Inequalities

- If specified using the $\geq$ encoding, follows using RUP.
- Probably possible to introduce the $\geq$ encoding using extension variables?

All-Different

$$
\begin{array}{rcl}
V & \in & \{\,1 \quad \quad \quad 4\,\} \\
W & \in & \{\,1 \quad 2 \quad 3 \quad \,\} \\
X & \in & \{\, \quad 2 \quad 3 \quad \,\} \\
Y & \in & \{\,1 \quad \quad 3 \quad \,\} \\
Z & \in & \{\,1 \quad \quad 3 \quad \,\}
\end{array}
$$

All-Different

$$
\begin{array}{rl}
V \in \{ & 1 \quad \quad \quad 4 \} \\
W \in \{ & 1 \quad 2 \quad 3 \quad \} \\
X \in \{ & \quad 2 \quad 3 \quad \} \\
Y \in \{ & 1 \quad \quad 3 \quad \} \\
Z \in \{ & 1 \quad \quad 3 \quad \}
\end{array}
$$

All-Different

$$
\begin{array}{rl}
V \in \{\, 1 & \quad\quad\quad 4 \,\} \\
W \in \{\, 1 & 2 \quad 3 \quad\;\; \} \\
X \in \{\, & 2 \quad 3 \quad\;\; \} \\
Y \in \{\, 1 & \quad\; 3 \quad\;\; \} \\
Z \in \{\, 1 & \quad\; 3 \quad\;\; \}
\end{array}
$$

`w_1` + `w_2` + `w_3` ≥ 1

All-Different

$V \in \{1 \quad\quad\quad 4\}$					
$W \in \{1 \quad 2 \quad 3 \quad\ \}$	w_1 +	w_2 +	w_3	$\geq$ 1
$X \in \{\quad\ 2 \quad 3 \quad\ \}$		x_2 +	x_3	$\geq$ 1
$Y \in \{1 \quad\quad\ 3 \quad\ \}$	y_1	+	y_3	$\geq$ 1
$Z \in \{1 \quad\quad\ 3 \quad\ \}$	z_1	+	z_3	$\geq$ 1

All-Different

$$
\begin{array}{rl}
V \in \{\ 1 & \quad\quad\ 4\ \} \\
W \in \{\ 1 & 2\ \ 3\quad \} \\
X \in \{\ \ & 2\ \ 3\quad \} \\
Y \in \{\ 1 & \quad 3\quad \} \\
Z \in \{\ 1 & \quad 3\quad \}
\end{array}
$$

$$
\begin{aligned}
\text{w\_1} + \text{w\_2} + \text{w\_3} &\geq 1 \\
\text{x\_2} + \text{x\_3} &\geq 1 \\
\text{y\_1} + \text{y\_3} &\geq 1 \\
\text{z\_1} + \text{z\_3} &\geq 1
\end{aligned}
$$

$$
\begin{aligned}
\rightarrow \quad -\text{v\_1} + -\text{w\_1} + -\text{y\_1} + -\text{z\_1} &\geq -1 \\
\rightarrow \quad -\text{w\_2} + -\text{x\_2} &\geq -1 \\
\rightarrow \quad -\text{w\_3} + -\text{x\_3} + -\text{y\_3} + -\text{z\_3} &\geq -1
\end{aligned}
$$

All-Different

$$
\begin{array}{rlllll}
V & \in \{\ 1 & & & 4\ \} \\
{\color{red}W} & \in \{\ 1 & 2 & 3 & \ \} \\
{\color{red}X} & \in \{ & 2 & 3 & \ \} \\
{\color{red}Y} & \in \{\ 1 & & 3 & \ \} \\
{\color{red}Z} & \in \{\ 1 & & 3 & \ \}
\end{array}
\qquad
\begin{array}{rl}
\texttt{w\_1} + \texttt{w\_2} + \texttt{w\_3} & \geq 1 \\
\texttt{x\_2} + \texttt{x\_3} & \geq 1 \\
\texttt{y\_1} + \texttt{y\_3} & \geq 1 \\
\texttt{z\_1} + \texttt{z\_3} & \geq 1
\end{array}
$$

$$
\begin{array}{ll}
\rightarrow & -\texttt{v\_1} + -\texttt{w\_1} + -\texttt{y\_1} + -\texttt{z\_1} \geq -1 \\
\rightarrow & -\texttt{w\_2} + -\texttt{x\_2} \geq -1 \\
\rightarrow & -\texttt{w\_3} + -\texttt{x\_3} + -\texttt{y\_3} + -\texttt{z\_3} \geq -1
\end{array}
$$

$$-\texttt{v\_1} \geq 1$$

All-Different

$$
\begin{array}{rl}
V \in & \{\, 1 \quad\quad\quad\quad 4 \,\} \\
W \in & \{\, 1 \quad 2 \quad 3 \quad\quad \} \\
X \in & \{\, \quad\;\; 2 \quad 3 \quad\quad \} \\
Y \in & \{\, 1 \quad\quad\;\; 3 \quad\quad \} \\
Z \in & \{\, 1 \quad\quad\;\; 3 \quad\quad \}
\end{array}
\qquad
\begin{array}{l}
\\
\mathtt{w\_1} + \mathtt{w\_2} + \mathtt{w\_3} \geq 1 \\
\mathtt{x\_2} + \mathtt{x\_3} \geq 1 \\
\mathtt{y\_1} + \mathtt{y\_3} \geq 1 \\
\mathtt{z\_1} + \mathtt{z\_3} \geq 1
\end{array}
$$

$$
\begin{array}{ll}
\rightarrow & -\mathtt{v\_1} + -\mathtt{w\_1} + -\mathtt{y\_1} + -\mathtt{z\_1} \geq -1 \\
\rightarrow & -\mathtt{w\_2} + -\mathtt{x\_2} \geq -1 \\
\rightarrow & -\mathtt{w\_3} + -\mathtt{x\_3} + -\mathtt{y\_3} + -\mathtt{z\_3} \geq -1
\end{array}
$$

$$
\begin{array}{l}
-\mathtt{v\_1} \geq 1 \\
\mathtt{v\_1} \geq 0
\end{array}
$$

All-Different

$$
\begin{array}{rl}
V \in & \{ 1 \quad\quad\quad 4 \} \\
W \in & \{ 1 \quad 2 \quad 3 \quad \} \\
X \in & \{ \quad 2 \quad 3 \quad \} \\
Y \in & \{ 1 \quad\quad 3 \quad \} \\
Z \in & \{ 1 \quad\quad 3 \quad \}
\end{array}
\qquad
\begin{array}{l}
w_1 + w_2 + w_3 \geq 1 \\
x_2 + x_3 \geq 1 \\
y_1 + y_3 \geq 1 \\
z_1 + z_3 \geq 1
\end{array}
$$

$$
\begin{array}{ll}
\rightarrow & -v_1 + -w_1 + -y_1 + -z_1 \geq -1 \\
\quad \rightarrow & -w_2 + -x_2 \geq -1 \\
\quad\quad \rightarrow & -w_3 + -x_3 + -y_3 + -z_3 \geq -1
\end{array}
$$

$$
\begin{array}{l}
-v_1 \geq 1 \\
v_1 \geq 0
\end{array}
$$

$$
0 \geq 1
$$

All-Different

```
// is our matching big enough?
if (left_covered.size() != vars.size()) {
    // nope. we've got a maximum cardinality matching that leaves
    // at least one thing on the left uncovered.
    return state.infer(FalseLiteral{ }, JustifyExplicitly{
        [&] (Proof & proof) {
            prove_matching_is_too_small(vars, vals,
                    constraint_numbers, proof, edges,
                    left_covered, matching);
        }
    });
}
```

Putting This Together

```
Problem p{ Proof{ "three_all_differents.opb",
    "three_all_differents.veripb" } };

auto w = p.create_integer_variable(0_i, 1_i, "w");
auto x = p.create_integer_variable(1_i, 2_i, "x");
auto y = p.create_integer_variable(0_i, 2_i, "y");
auto z = p.create_integer_variable(0_i, 1_i, "z");
p.post(AllDifferent{ { w, x, y } });
p.post(AllDifferent{ { x, y, z } });
p.post(AllDifferent{ { w, z } });

solve(p, [&] (const State & s) -> bool {
    cout << s(w) << "␣" << s(x) << "␣"
        << s(y) << "␣" << s(z) << endl;
    return true;
});
```

Putting This Together

```
$ ./three_all_differents
propagators: 4
recursions: 3
failures: 2
propagations: 12
max depth:  1
solutions: 0
solve time: 0.000819s
$ veripb --stats three_all_differents.{opb,veripb}
c statistic: time total: 0.00s
Verification succeeded.
```

Putting This Together

```
* #variable= 9 #constraint= 16
* convenience true and false variables
* variable w domain
1 w_0 1 w_1 >= 1 ;
-1 w_0 -1 w_1 >= -1 ;
* variable x domain
1 x_1 1 x_2 >= 1 ;
-1 x_1 -1 x_2 >= -1 ;
* variable y domain
1 y_0 1 y_1 1 y_2 >= 1 ;
-1 y_0 -1 y_1 -1 y_2 >= -1 ;
* variable z domain
1 z_0 1 z_1 >= 1 ;
-1 z_0 -1 z_1 >= -1 ;
* constraint all different
-1 w_0 -1 y_0 >= -1 ;
-1 w_1 -1 x_1 -1 y_1 >= -1 ;
-1 x_2 -1 y_2 >= -1 ;
* constraint all different
-1 y_0 -1 z_0 >= -1 ;
-1 x_1 -1 y_1 -1 z_1 >= -1 ;
-1 x_2 -1 y_2 >= -1 ;
* constraint all different
-1 w_0 -1 z_0 >= -1 ;
-1 w_1 -1 z_1 >= -1 ;
```

Putting This Together

```
pseudo-Boolean proof version 1.0
f
* guessing w_0, decision stack is [ ]
u 1 ~w_0 1 ~y_0 >= 1 ;
u 1 ~w_0 1 ~z_0 >= 1 ;
* all different, found hall set { x y } { 1 2 }
p 3 5 + 13 + 14 + 0
* backtracking
u 1 ~w_0 >= 1 ;
* guessing w_1, decision stack is [ ]
u 1 ~w_1 1 ~x_1 >= 1 ;
u 1 ~w_1 1 ~y_1 >= 1 ;
u 1 ~w_1 1 ~y_2 >= 1 ;
u 1 ~w_1 1 ~z_1 >= 1 ;
* backtracking
u 1 ~w_1 >= 1 ;
* backtracking
u >= 1 ;
c -1
```

How Expensive is Proof Logging?

- Laurent D. Michel, Pierre Schaus, Pascal Van Hentenryck: MiniCP: a lightweight solver for constraint programming. Math. Program. Comput. 13(1) (2021).
- Five benchmark problems allowing comparison of solvers "doing the same thing":
  - Simple models.
  - Fixed search order and well-defined propagation consistency levels.
  - Few global constraints.
- Compiled Java code for MiniCP, and source for benchmarks.
- Sadly, there are some discrepancies…

How Expensive is Proof Logging?

		Runtime (s)				
Instance	Nodes	Glasgow	Glasgow+Proof	VeriPB	MiniCP	Choco
Magic Series	1,192	17.3	35.4	331.0×	24.0	25.4
46M propagations, 415MByte OPB file, 1.2GByte VeriPB file						
(other model)	596	1.6	3.9	29.8×		
690K propagations, 419MByte OPB file, 141MByte VeriPB file						
Magic Square	6,024,078	54.1	604.6	30.2×	34.7	18.9
181M propagations, 527KByte OPB file, 50GByte VeriPB file						
n Queens	49,339,389	395.0	2454.8	19.2×	491.7	278.5
2.2B propagations, 48MByte OPB file, 169GByte VeriPB file						
QAP	123,333	35.9	125.4	10 days?	8.6	6.3
13M propagations, 14GByte OPB file (with cheating), 6.2GByte VeriPB file Need to deal with very large integer domains Element2D with constant arrays using GAC table is slow						
TSP	Haven't implemented Circuit constraint yet					

Should We Trust This?

Real world problem

↓ Modelling

High level model (Essence, MiniZinc)

↓ Compilation

Low level model (CP, MIP, SAT, . . .)

↓ Preprocessing

Solver's internal representation

↓ Solving

Solution

?

PB Encoding

VeriPB Proof +

Verifier

Trusting the Modelling and Compilation

- Outwith the scope of this project.

Trusting the Pseudo-Boolean Encoding

- Use simple encodings, not good encodings.
- Still plenty of room for errors, e.g. the Element constraint is fiddly.
- Test that single-constraint models find exactly the right set of solutions, compared to generate-and-test?

Trusting the Pseudo-Boolean Encoding

```
auto data = vector{
    tuple{ { 2, 5 }, { 1, 6 }, { 1, 12 } },
    tuple{ { 1, 6 }, { 2, 5 }, { 5, 8 } },
    /* ... */
    tuple{ { 1, 1 }, { 2, 4 }, { -5, 5 } }
};

for (auto & [ r1, r2, r3 ] : data) {
    if (! run_arithmetic_test<Plus>(r1, r2, r3,
            [] (int a, int b, int c) {
                return a + b == c;
            }))
        return EXIT_FAILURE;
    if (! run_arithmetic_test<Div>(r1, r2, r3,
            [] (int a, int b, int c) {
                return 0 != b && a / b == c;
            }))
        return EXIT_FAILURE;
}
```

Trusting the Pseudo-Boolean Encoding

```
set<tuple<int, int, int> > expected, actual;
for (int v1 = v1_range.first ; v1 <= v1_range.second ; ++v1)
    for (int v2 = v2_range.first ; v2 <= v2_range.second ; ++v2)
        for (int v3 = v3_range.first ; v3 <= v3_range.second ; ++v3)
            if (is_satisfing(v1, v2, v3))
                expected.emplace(v1, v2, v3);

Problem p{ Proof{ "test.opb", "test.veripb" } };
auto v1 = p.create_integer_variable(v1_range.first, v1_range.second);
auto v2 = p.create_integer_variable(v2_range.first, v2_range.second);
auto v3 = p.create_integer_variable(v3_range.first, v3_range.second);
p.post(Arithmetic_{ v1, v2, v3 });
solve(p, [&] (const State & s) -> bool {
    actual.emplace(s(v1), s(v2), s(v3));
    return true;
});

if ((expected != actual) ||
        (0 != system("veripb␣test.opb␣test.veripb")))
    return false;
```

Trusting the Preprocessing and Solving

- Fully verifiable (except possibly for enumeration. . .).

Trusting the Verification

- Verifier is very simple, and knows nothing about constraint programming.
- Probably possible to produce a formally verified verifier.
  - Still potentially buggy, but formally verified code can only contain "a better class" of bug.
- Will require a "core" proof format.
  - Most syntactic sugar removed.
  - Rewriting or annotating RUP constraints?
- A buggy proof simplification tool might break valid proofs, but is unlikely to make invalid proofs valid.

Other Things We Can Verify

- Automatic tabulation.
- Symmetries.
- Restarts.
- Clause learning.
- Discrepancy search?

What Will We Have?

- Don't know that the solvers are right.
- Do know that if a solver ever produces a wrong answer, it can be detected.
  - Even if due to a hardware or compiler error, or faulty maths.
  - We will need to get used to verification being (a constant factor) slower than solving.
- Also helps with testing and solver development: bugs are caught if incorrect reasoning is performed, rather than if a wrong answer is produced.
- We also have a record of exactly what was actually solved.
- Potentially possible to re-use proof logs for empirical algorithmics?

https://ciaranm.github.io/

ciaran.mccreesh@glasgow.ac.uk

University of Glasgow

Royal Academy of Engineering