

My Solver Produces the Right Answer, And It Can Prove It

Ciaran McCreesh

with Jan Elffers, Stephan Gocht, and Jakob Nordström



**University
ofGlasgow**



What We Have Now
●oooooooooo

Checking Outputs, not Programs
ooooooo

The Long Term Vision
ooooo

A Proof of Concept
ooooo

Ugly Technical Details
oooooooooooooooooooo

Solvers

- Increasingly being used for decision making, not just decision support.

The Problem

- Solvers have bugs.
 - Some models are mishandled.
 - Often reasonably easy to spot, but hard to deal with.
 - Some instances will make the solver give the wrong answer.
 - Often *rare*: between one in a hundred and one in ten thousand, and only on relatively large instances.
 - Even if the answer is right, it might have been reached by unsound reasoning.
 - Much more common, but essentially impossible to detect.

What We Have Now
oo●oooooooo

Checking Outputs, not Programs
oooooo

The Long Term Vision
ooooo

A Proof of Concept
oooo

Ugly Technical Details
oooooooooooooooooooo

For Example...

```
$ ./glasgow_clique_solver 25-727-9826.clq
```

```
size = 10
```

```
vertices = 1 2 4 9 15 20 21 22 24 25
```

```
$ ./MoMC2016 25-727-9826.clq
```

```
M 21 20 11 4 22 19 24 1 13
```

```
s Instance 25-727-9826.clq Max_CLQ 9
```

Unit Testing?

- Tells you that some parts of your solver produce the answer you expect on some inputs.

Whole Program Testing?

- Need many instances that are reasonably easy to solve.
- Optimal solutions must be known.

Proofs of Correctness?

- Tell you nothing about the algorithm implementation.
- Proofs are just as susceptible to missing cases or combinations of effects as programs are, and with fewer opportunities (compiler, testing, real instances) for this to be caught.

Formal Verification?

- A few attempts on relatively simple solvers.
- Nothing on the scary algorithms.
- Particularly hard to do with performance-critical algorithms and solvers.

None of This Works...

- MoMC was widely tested, and its algorithm proven correct.
- For the bug to result in a wrong solution being given:
 - The graph density must be between 0.6 and 0.8.
 - The optimal solution must be unique...
 - ...and it must include the last vertex in the input graph...
 - ...and it must not be detected during presolve...
 - ...and the bound function has to behave in a certain way.
- On the other hand, in any given run, the solver usually throws away large numbers of subproblems without justification.

Two Different Solvers

- Removes the need to know what the optimal solution is, for whole program test instances.
- Given enough instances, eventually they will disagree.
 - But only if you generate instances that could trigger the bug...
- Needs two good solvers that use very different techniques.

Two Independent Implementations

- Run two independent implementations of the same algorithm, compare their step by step function calls (or at least compare number of recursive calls made).
- Requires, e.g. stable sorting, reproducible random number generation.
- A good way of converging on the same bugs...
- Also doesn't help you if the algorithm is incorrect.

Solution Checkers

- For yes-instances of decision problems, it is usually relatively easy to check whether a solution is valid.
- Similarly, for optimisation problems, we can check that a solution is valid.
- Have someone else independently write a solution checker.
 - Should be much less effort than a full solver.

The Idea Behind Proof Logging

- Have the solver output a solution, together with an auditable “proof”.
 - When we’re dealing with NP-hard problems, this proof can be exponentially long, but hopefully only proportional to how long the solver took.
- Someone else writes a “proof checker”, which is much simpler than the solver.
- A bit like a solution checker, but for unsatisfiable instances.

How It Works in SAT

- A common proof format known as DRAT.
- Essentially, a sequence of redundant clauses.
- In the annual SAT competition, solvers are expected to be able to produce proofs.

Are Computer Proofs Socially Acceptable?

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 n th powers are required to sum to an n th power, $n > 2$.

REFERENCE

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

Are Computer Proofs Socially Acceptable?

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 created¹ 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 proof², 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. ▶

Why DRAT Proofs Won't Work Elsewhere

- DRAT proofs are very closely tied to SAT solving.
- SAT solvers can't count:
 - Exponential proofs for simple “pigeonhole” problems.
 - Can't reason about vertex degrees in graphs.
 - Can't do all-different reasoning.
- Solvers that perform stronger reasoning than SAT solvers will need a stronger proof format.
 - But can we trust a proof checker that knows dozens of rich constraints?

However...

- Recent discovery: pseudo-boolean (cutting planes) proofs *can* express everything we currently do in subgraph algorithms, such as...
 - All different and Hall sets,
 - Colour bounds,
 - Neighbourhood degree sequence reasoning,
 - Counting short paths.
- But pseudo-boolean solvers know nothing about graphs, matching algorithms, etc.

Trustworthy Solvers

- High-level solvers should use verifiable compilation techniques to produce low-level models.
- Low-level CP solvers should be “auditable”. For any answer, we should be able to request a proof:
 - In a standard format,
 - Which does not take much longer than the original solution to produce,
 - That is easily verifiable by a much simpler tool,
 - And that does not prevent solvers from supporting new global constraints.
- These proofs should be translatable back into high-level terms.

Safe Model Compilation

- It's possible for compilers, and they have front-end languages and target architectures that are much richer than CP models and solvers.
- We can already translate backwards, for outputting solutions.

A Proof Language for CP

- DRAT won't work.
- Cutting planes might be enough.
- We might need a suite of proof languages that can be combined.

Does Anyone Care?

The screenshot shows the EPSRC website's navigation bar with links for FUNDING, RESEARCH, INNOVATION, SKILLS, NEWS, EVENTS AND PUBLICATIONS, and ABOUT US. Below the navigation, a sidebar shows 'Funding' and 'Calls'. The main content area displays a large heading: 'Security for all in an AI enabled society - call for proposals'. Below the heading is a table of details:

Issue date:	03 July 2019
Opening date:	30 July 2019
Closing date:	10 October 2019 at 16:00
Status:	Open
Tag:	Invitation for proposals

- o Explainable AI: As part of the problem with many approaches currently utilised for AI, black box methods make it difficult to interrogate a system if something goes wrong. Could an AI system be able to explain its security properties and provide assurances about its security as it evolves? Could this be adaptable to different

Does Anyone Care?



- ✓ Ensure that the development, deployment and use of AI systems meets the seven key requirements for Trustworthy AI: (1) human agency and oversight, (2) technical robustness and safety, (3) privacy and data governance, (4) transparency, (5) diversity, non-discrimination and fairness, (6) environmental and societal well-being and (7) accountability.
- ✓ Consider technical and non-technical methods to ensure the implementation of those requirements.
- ✓ Foster research and innovation to help assess AI systems and to further the achievement of the requirements; disseminate results and open questions to the wider public, and systematically train a new generation of experts in AI ethics.

What We Have Now
oooooooooooo

Checking Outputs, not Programs
ooooooo

The Long Term Vision
oooo●○

A Proof of Concept
ooooo

Ugly Technical Details
oooooooooooooooooooo

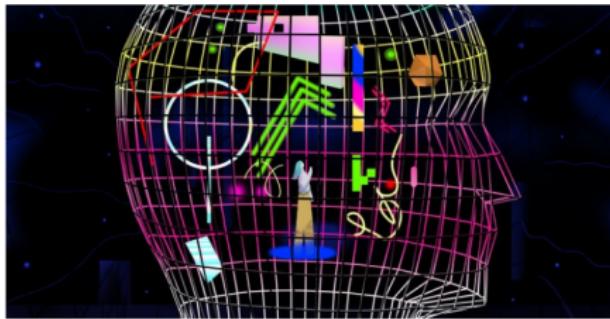
Does Anyone Care?

THE NEW YORKER

ANNALS OF TECHNOLOGY

THE HIDDEN COSTS OF AUTOMATED THINKING

By Jonathan Zittrain July 23, 2019



Overreliance on artificial intelligence may put us in intellectual debt. Illustration by Jon Han

What We Have Now
oooooooooooo

Checking Outputs, not Programs
ooooooo

The Long Term Vision
oooo●

A Proof of Concept
ooooo

Ugly Technical Details
oooooooooooooooooooo

What if it Actually Works?

- Might auditable algorithms and solvers become a necessity, for liability reasons?

Making the Glasgow Subgraph Solver Produce Proofs

- Theoretically possible.
- Practically possible, but quite a bit of engineering work.

A Very Small CP Solver



/ciaranm/certified-constraint-solver

- Written to iron out the engineering details before tackling a more complex solver.
- Supports not equals (AC), all different (GAC), and table constraints (checking).
 - Sufficient for Sudoku, clique, graph colouring, subgraph isomorphism, ...
- The solver is definitely buggy.
- But when it outputs an answer, we can verify it is correct!

For Example...

```
$ cat small.model
intvar A { 1 2 }
intvar B { 1 3 }
intvar C { 1 2 3 }
intvar D { 1 2 }
intvar E { 1 3 }
alldifferent { A B C }
alldifferent { C D E }
notequal B E

$ certified_constraint_solver \
    --prove small.model
status = false
nodes = 5

$ ls small.*
small.log small.model small.opb

$ refpy small.{opb,log}
Verification succeeded.
```

Checking the Proof



/StephanGocht/refpy

- A proof checker for cutting planes proofs.
- Knows nothing about Hall sets, matching algorithms, strongly connected components, ...
- Written entirely independently. No collusion.

End of The Interesting Part



Our Example, Again

```
intvar A { 1 2 }
intvar B { 1 3 }
intvar C { 1 2 3 }
intvar D { 1 2 }
intvar E { 1 3 }
alldifferent { A B C }
alldifferent { C D E }
notequal B E
```

Pseudo-Boolean Solving

- Variables $x_i \in \{0, 1\}$.
- Literals ℓ_i are x_i or $\overline{x_i}$, where $x_i + \overline{x_i} = 1$.
- Constraints $\sum_i a_i \ell_i \geq A$, where $a_i, A \in \mathbb{Z}$.
- Find a satisfying assignment maximising $\sum_i a_i \ell_i$, where $a_i \in \mathbb{Z}$.

Compiling CP Variables to PB

- A CP variable $X \in \{a, b, c\}$ becomes x_a, x_b, x_c .
- Each variable takes exactly one value:

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

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

Compiling CP Variables to PB

CP Model

```
intvar A { 1 2 }  
intvar B { 1 3 }  
intvar C { 1 2 3 }  
intvar D { 1 2 }  
intvar E { 1 3 }
```

Generated OPB Fragment

```
* variable A: (1, x1) (2, x2)  
1 x1 1 x2 >= 1 ;  
-1 x1 -1 x2 >= -1 ;  
* variable B: (1, x3) (3, x4)  
1 x3 1 x4 >= 1 ;  
-1 x3 -1 x4 >= -1 ;  
* variable C: (1, x5) (2, x6) (3, x7)  
1 x5 1 x6 1 x7 >= 1 ;  
-1 x5 -1 x6 -1 x7 >= -1 ;  
* variable D: (1, x8) (2, x9)  
1 x8 1 x9 >= 1 ;  
-1 x8 -1 x9 >= -1 ;  
* variable E: (1, x10) (3, x11)  
1 x10 1 x11 >= 1 ;  
-1 x10 -1 x11 >= -1 ;
```

Compiling Not Equals to PB

- CP variables $X \in \{a, b, c\}$ and $Y \in \{b, c, d\}$, constraint $X \neq Y$.
- For each value they have in common, we can't pick both:

$$x_b + y_b \leq 1 \quad \text{i.e.} \quad -1x_b + -1y_b \geq -1$$

$$x_c + y_c \leq 1 \quad \text{i.e.} \quad -1x_c + -1y_c \geq -1$$

Compiling Not Equals to PB

CP Model

```
intvar B { 1 3 }
intvar E { 1 3 }
notequal B E
```

Generated OPB Fragment

```
* variable B: (1, x3) (3, x4)
* variable E: (1, x10) (3, x11)
* not equals
-1 x3 -1 x10 >= -1 ;
-1 x4 -1 x11 >= -1 ;
```

Compiling All-Different

- CP variables $X \in \{a, b, c\}$, $Y \in \{b, c\}$, $Z \in \{b, c, d\}$, constraint $\text{alldifferent}(\{X, Y, Z\})$.
- We could do pairwise not-equals, as in SAT, or...
- For each value, it can be used at most once:

$$-1x_a \geq -1$$

$$-1x_b + -1y_b + -1z_b \geq -1$$

$$-1y_c + -1z_c \geq -1$$

$$-1z_d \geq -1$$

Compiling All-Different

CP Model

```
intvar A { 1 2 }
intvar B { 1 3 }
intvar C { 1 2 3 }
alldifferent 3 A B C
```

Generated OPB Fragment

```
* variable A: (1, x1) (2, x2)
* variable B: (1, x3) (3, x4)
* variable C: (1, x5) (2, x6) (3, x7)
* all different
-1 x1 -1 x3 -1 x5 >= -1 ;
-1 x2 -1 x6 >= -1 ;
-1 x4 -1 x7 >= -1 ;
```

What We Have Now
oooooooooooo

Checking Outputs, not Programs
ooooooo

The Long Term Vision
oooooo

A Proof of Concept
ooooo

Ugly Technical Details
oooooo●oooooooooooo

Compiling Table

- Involves introducing auxiliary variables in the PB model.

Cutting Planes Proofs

Model axioms

From the input file

Literal axioms

$$\overline{\ell_i \geq 0}$$

Addition

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

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 $c \in \mathbb{N}^+$

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

Machine-Readable Proofs: Getting Started

Proof Log

```
refutation using f l p c 0
```

```
f 18 0
```

```
l 11 0
```

Verifier Output

```
0 (rule 0): >= 0
```

```
1 (rule 1): +1x1 +1x2 >= 1
```

```
2 (rule 1): +1~x1 +1~x2 >= 1
```

```
3 (rule 1): +1x3 +1x4 >= 1
```

```
4 (rule 1): +1~x3 +1~x4 >= 1
```

* and so on

```
18 (rule 1): +1~x4 +1~x7 >= 1
```

```
19 (rule 2): +1x1 >= 0
```

```
20 (rule 2): +1~x1 >= 0
```

```
21 (rule 2): +1x2 >= 0
```

```
22 (rule 2): +1~x2 >= 0
```

* and so on

```
39 (rule 2): +1x11 >= 0
```

```
40 (rule 2): +1~x11 >= 0
```

The CP Search Tree

```
intvar A { 1 2 }
intvar B { 1 3 }
intvar C { 1 2 3 }
intvar D { 1 2 }
intvar E { 1 3 }

notequal B E * C1
alldiff { A B C } * C2
alldiff { C D E } * C3
```

- No propagation initially.
- Guess $A = 1$, so $A \neq 2$:
 - $B \neq 1, C \neq 1, C \neq 3$ (C2)
 - $E \neq 3$ (C1)
 - $C \in \{2\}, D \in \{1, 2\}, E \in \{1\}$ (C3)
- Guess $A = 2$, so $A \neq 1$:
 - $C \neq 2$ (C2)
 - $D \neq 1$ (C3)
 - Guess $B = 1$, so $B \neq 3$:
 - ... (C1, C2, C3) ... ↴
 - Guess $B = 3$, so $B \neq 1$:
 - ... (C1, C2, C3) ... ↴

Overview of our Proof

- 1 Derive $\overline{a_1} \geq 1$.
- 2 Derive $\overline{a_2} + \overline{b_1} \geq 1$.
- 3 Derive $\overline{a_2} + \overline{b_3} \geq 1$.
- 4 Combine 2 and 3 to derive $\overline{a_2} \geq 1$.
- 5 Combine 1 and 4 to derive $0 \geq 1$.

Justifying Deletions

- Whenever a CP propagator performs a deletion $F \neq v$, generate a proof line of the form

$$\underbrace{\overline{a_1} + \overline{b_2} + \overline{c_2} + \overline{e_1}}_{\text{A subsequence of the active guesses}} + \overline{f_v} \geq 1.$$

- Remember the proof line number alongside F 's domain.
- When we get a domain wipeout, combine the reasons with the “takes at least one value” model axiom to get

$$\underbrace{\overline{a_1} + \overline{b_2} + \overline{c_2}}_{\text{A subsequence of earlier guesses}} + \overline{e_2} \geq 1,$$

and remember this as the deletion reason for the last guess.

- If we detect a contradiction involving multiple variables, derive the same thing, using constraint-specific rules.

Proving $A \neq 1$

Proof Log

```
* guess A=1 (x1) so A!=2 (x2)
p 2 0
* all different, B!=1
p 0 1 + 14 + 27 + 41 + 2 d 0
* all different, C!=1
p 0 1 + 14 + 23 + 41 + 2 d 0
* all different, C!=3
p 0 1 + 3 + 14 + 16 + 27 +
    41 + 2 d 0
* not_equals, E!=3
p 3 42 + 18 + 2 d 0
* alldifferent contradiction
p 5 43 + 44 + 4 d 0
p 7 3 d 0
p 9 45 + 3 d 0
p 0 11 + 12 + 27 + 46 +
    47 + 48 + 4 d 0
```

Verifier Output

```
41 (rule 3): +1~x1 +1~x2 >= 1
42 (rule 4): +1~x1 +1~x3 >= 1
43 (rule 5): +1~x1 +1~x5 >= 1
44 (rule 6): +1~x1 +1~x7 >= 1
45 (rule 7): +1~x1 +1~x11 >= 1
46 (rule 8): +1x6 +1~x1 >= 1
47 (rule 9): +1x8 +1x9 >= 1
48 (rule 10): +1x10 +1~x1 >= 1
49 (rule 11): +1~x1 >= 1
```

The Remainder of the Proof

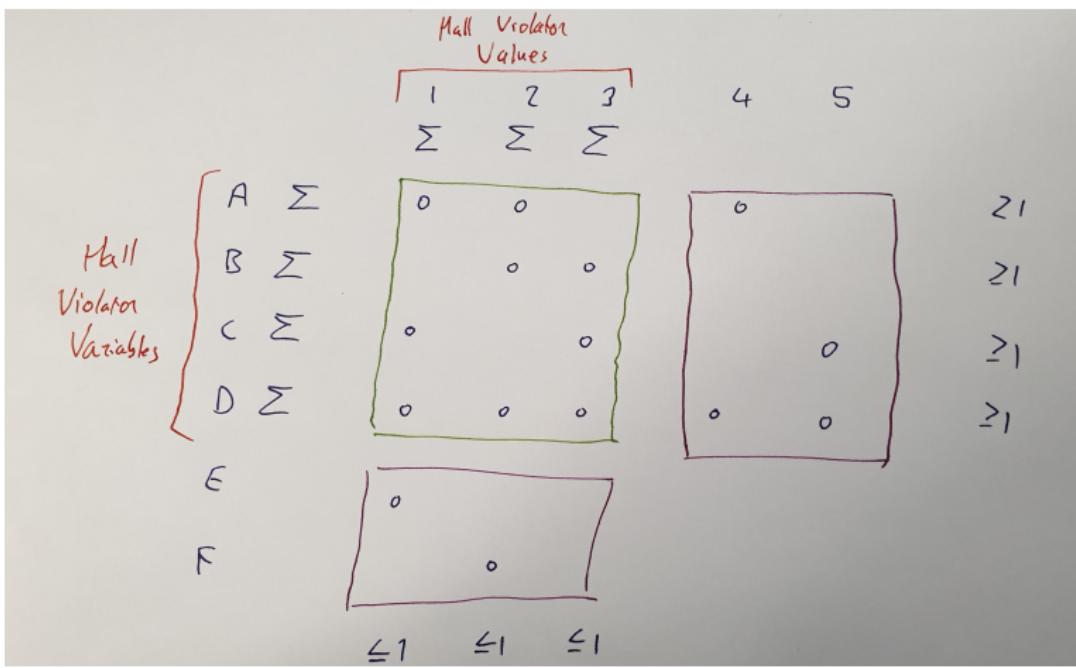
Proof Log

```
* guessing A=1 (x1)
*
* ...
p 49 20 + 2 d 0
* guessing A=2 (x2)
*
* ...
* guessing A=2 (x2) B=1 (x3)
*
* ...
p 59 24 + 2 d 0
* guessing A=2 (x2) B=3 (x4)
*
* ...
p 66 26 + 2 d 0
p 3 60 + 67 + 3 d 0
p 68 22 + 2 d 0
p 1 50 + 69 + 3 d 0
c 70 0
```

Verifier Output

```
50 (rule 12): +1~x1 >= 1
60 (rule 22): +1~x3 +1~x2 >= 1
67 (rule 29): +1~x4 +1~x2 >= 1
68 (rule 30): +1~x2 >= 1
69 (rule 31): +1~x2 >= 1
70 (rule 32): >= 1
Verification succeeded.
```

The Hall Violator Step



The Hall Violator Step

- 1 For each variable in the Hall violator, take the “at least one value from its domain” model axiom, and for each value already eliminated from this domain, add the reason.
- 2 Divide 1 by a large number to remove duplicate guesses.
- 3 Add together each of the “this value can be used at most once” model axioms for values in the Hall violator.
- 4 From 3, for each variable in the constraint, for each value in its initial domain, if either the value is not in the Hall violator, or the value is not in its current domain, cancel it using a literal axiom.
- 5 Add together 2 and 4. Divide by a large number.
- 6 Everything (except guesses) cancels out, but only if you correctly found a Hall violator.

Are Cutting Planes Proofs Enough for CP?

- Can do:
 - Anything SAT can do.
 - All the clever subgraph isomorphism things we do currently.
 - GAC on all(?) of the polynomial-time flow-based constraints.
 - BC for linear inequalities over boolean variables (duh).
 - Probably some other constraints that SAT can't do.
- Can't do:
 - Sparse domains.
 - Probably some other constraints.
- Interesting project: categorise the global constraints catalogue.

<http://www.dcs.gla.ac.uk/~ciaran>
ciaran.mccreesh@glasgow.ac.uk



University
of Glasgow

