Justifying All Differences Using Pseudo-Boolean Reasoning

Jan Elffers,[1,2] **Stephan Gocht,**[1,2] **Ciaran McCreesh,**[3] **Jakob Nordström**[2,4]
[1]Lund University, Lund, Sweden
[2]University of Copenhagen, Copenhagen, Denmark
[3]University of Glasgow, Glasgow, Scotland
[4]KTH Royal Institute of Technology, Stockholm, Sweden
{jan.elffers, stephan.gocht}@cs.lth.se, ciaran.mccreesh@glasgow.ac.uk, jn@di.ku.dk

Abstract

Constraint programming solvers support rich global constraints and propagators, which make them both powerful and hard to debug. In the Boolean satisfiability community, proof-logging is the standard solution for generating trustworthy outputs, and this has become key to the social acceptability of computer-generated proofs. However, reusing this technology for constraint programming requires either much weaker propagation, or an impractical blowup in proof length. This paper demonstrates that simple, clean, and efficient proof logging is still possible for the all-different constraint, through pseudo-Boolean reasoning. We explain how such proofs can be expressed and verified mechanistically, describe an implementation, and discuss the broader implications for proof logging in constraint programming.

Introduction

Constraint programming solvers are increasingly being used for fully automated decision making without a human in the loop, even in safety-critical applications. Unfortunately, these solvers will sometimes have bugs, and these bugs are hard to detect using conventional testing methods (Akgün et al. 2018; Gillard, Schaus, and Deville 2019). Meanwhile, formal proofs of correctness can be useful in verifying the mathematical description of some of the algorithms underlying these solvers, but are not yet suitable for verifying a full implementation of a high-performance modern solver. It would therefore be reassuring to have a different way to be confident that a solver has produced a correct answer.

When a constraint programming solver outputs "yes" for a decision instance, it is usually relatively easy to verify that the answer it provides is valid—for example, by having a different person implement a solution checker, which is typically much simpler than writing a program which finds a solution. Similarly, for optimisation problems, verifying the *feasibility* of a solution is simple. However, for "no" decision instances, and for verifying optimality, a solver likely took a large number of complicated steps to reach that conclusion, and there is no simple way of demonstrating that those steps were valid. In the Boolean satisfiability community, proof logging is the standard approach to this problem: in order to take part in the SAT competitions (Heule, Järvisalo, and Suda 2019), a solver must be able to output a "certificate" or "proof log" alongside a claimed unsatisfiable result. This log is a (potentially very large) file in a standard format known as DRAT (Heule, Hunt Jr., and Wetzler 2013b; 2013a; Wetzler, Heule, and Hunt Jr. 2014), which may be verified by an external tool. Importantly, the proof verifying tools used are much simpler than the solvers, giving us confidence in their correctness. Proof logging for Boolean satisfiability has been key to the social acceptance of computer-produced proofs of mathematical conjectures (e.g. Heule, Kullmann, and Marek 2016, Lamb 2016).

Due to their use of rich global constraints like "all different", constraint programming solvers cannot simply reuse this approach: compiling constraint programming to Boolean satisfiability results in weaker reasoning for many constraints (Bessiere et al. 2009b), and although it is *theoretically* possible to use the DRAT format to justify rich propagation, developing any approach that is feasible *in practice* has remained stubbornly out of reach.

In this work, we instead propose the use of a new proof format based upon pseudo-Boolean reasoning and cutting planes proofs (Cook, Coullard, and Turán 1987). We show that this format can easily and efficiently capture all of the reasoning carried out by the all different propagator. This allowing us to develop, for the first time, an efficient verification system for non-trivial constraint programming inference techniques: we describe a tool which can verify these proofs, as well as the implementation of a small constraint solver which produces them. We conclude with a discussion of the broader implications for constraint programming.

Pseudo-Boolean Models

An instance of the pseudo-Boolean (PB) decision problem, or a PB formula, is defined by a set of $\{0,1\}$-valued variables $\{x_1,\ldots,x_n\}$ and a set of linear constraints over these variables, each of which is of the form $\sum_{i=1}^{n} a_i \ell_i \geq A$, where the a_is and A are all integers, and each ℓ_i is either an unnegated literal x_i or a negated literal $\overline{x_i}$. Using the equality $x_i + \overline{x_i} = 1$, which encodes the semantics of negation, we can always rewrite a PB constraint so that all a_i are non-negative and A is strictly positive, and so when describing reasoning rules we will assume that all constraints are written in this so-called *normalized form* (this is purely for notational convenience, and does not affect expressive power). For a constraint in normalized form, A is often referred to as the *degree of falsity*, or just *degree*. The objective is to assign values to the variables so that all constraints are respected. If this is possible, we say that the PB instance is *satisfiable*; otherwise it is *unsatisfiable*.

By treating 0 as "false" and 1 as "true", any instance of the Boolean satisfiability (SAT) problem in conjunctive normal form (CNF) can be viewed as a PB formula by observing that, e.g. $x \vee \overline{y} \vee z$ is satisfied if and only if $x + \overline{y} + z \geq 1$. On the other hand, not all pseudo-Boolean constraints can be translated into a single SAT clause. For example, "cardinality" constraints such as $x_1 + x_2 + x_3 + x_4 + x_5 \geq 3$ must be encoded before they can be handled by a SAT solver, and more general constraints such as $x_1 + 2\overline{x_2} + 3x_3 + 4\overline{x_4} + 5\overline{x_5} \geq 7$ require even more complicated handling.

Cutting Planes Proofs

To reason about satisfiability or unsatisfiability of pseudo-Boolean formulae we use the *cutting planes* proof system (Cook, Coullard, and Turán 1987), which can be described as follows. We have four sets of derivation rules, which we describe using the standard notation with a list of preconditions above a horizontal line that allow us to infer the constraint below the line. Initially, these preconditions can be the constraints in the PB formula, which we refer to as *(input) axioms*, but later they can be any constraint derived by a previous rule application. Firstly, we may assert unconditionally the *literal axiom* that any x_i or $\overline{x_i}$ is non-negative:

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

Secondly, we may create a new constraint by *addition* of any two constraints:

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

(Here we implicitly assume that the equality $x_i + \overline{x_i} = 1$ is applied to cancel any literals of opposing signs, and shift any constant terms to the right-hand side, so that the resulting constraint is in normalized form). Thirdly, we can apply *multiplication* by a positive integer c to any constraint:

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

Fourthly, we may apply *division* by any positive integer c, where all fractional values in the divided constraint are rounded upwards:

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

(Note that the soundness of this rule requires that the constraint is in normalized form.)

Cutting planes is a complete proof system for pseudo-Boolean formulas in the same way that the resolution proof system (Blake 1937; Davis and Putnam 1960) is complete for CNF formulas—it is always possible to derive $0 \geq 1$ from a PB formula using a cutting planes proof if and only if this formula is unsatisfiable. We refer the interested reader to Buss and Nordström (2019) for more details.

Unit Propagation

Two key notions in the context of SAT solving and pseudo-Boolean solving, which will be important for us also, are those of *propagating* and *conflicting* constraints. Let $C = \sum_i a_i \ell_i \geq A$ be a PB constraint and let ρ be a partial truth value assignment. Then the *slack* of C under ρ measures how much room there is left for error if we want to satisfy C given ρ. Formally, $\mathrm{slack}(C; \rho) = \sum_{i\,:\,\rho(\ell_i) \neq 0} a_i - A$ is the sum of the coefficients of all non-falsified literals minus the degree. If $\mathrm{slack}(C; \rho) < 0$, then there is no way C can be satisifed, and we say that the constraint is *conflicting* under ρ. If for some coefficient a_i we have $\mathrm{slack}(C; \rho) < a_i$, then ℓ_i must be set to true to avoid conflict, and we say that C *unit propagates* ℓ_i under ρ. By way of example, for the empty assignment the constraint $C = x_1 + 2\overline{x_2} + 3x_3 + 4\overline{x_4} + 5\overline{x_5} \geq 7$ has slack 8 and does not propagate anything, but if we set $x_5 = 1$ then the slack drops to 3 and C propagates $x_4 = 0$. If we instead set $x_4 = 1$, then the slack decreases to -1 and we have a conflict. We note that the pseudo-Boolean notation of unit propagation is just a generalization of that used in conflict-driven clause learning (CDCL, Marques Silva and Sakallah 1999), since a disjunctive clause unit propagates only when the slack is 0 (since all coefficients are 1), which happens precisely when all literals in the clause except one is falsified.

Reverse Unit Propagation

The concept of unit propagation has turned out to be very useful for proof logging as explained next. A constraint C can be derived from a PB formula F if and only if F together with the negation of C is unsatisfiable. In general, deciding whether this is so is an NP-complete problem, but in the context of CDCL solving it is much easier. Namely, if F is the set of clauses derived so far and C is the new clause learned from the most recent conflict, then it holds that F plus $\neg C$ (i.e., the conjunction of the negations of all literals in C) unit propagates to conflict. When this is the case, we say that C follows from F by *reverse unit propagation (RUP)* or is a *RUP clause*. The correctness of a basic CDCL proof search loop can be verified efficiently by just emitting the learned clauses one by one and checking that they are RUP clauses (Goldberg and Novikov 2003; Van Gelder 2008). The more expressive DRAT proof logging format (Heule, Hunt Jr., and Wetzler 2013b; 2013a; Wetzler, Heule, and Hunt Jr. 2014) used in current state-of-the-art SAT solvers is based on an extension of this simple but powerful idea.

The RUP concept readily transfers to a pseudo-Boolean setting. We say that the constraint $\sum_i a_i \ell_i \geq A$ is RUP for a PB formula F if the negation of this constraint (i.e. $\sum_i -a_i \ell_i \geq 1 - A$) together with F unit propagates to conflict, and if this is the case then it is clearly sound to derive

$\sum_i a_i \ell_i \geq A$ from F. This is useful in that, as we will show, it allows for very efficient proof logging for some constraint programming propagation algorithms.

Machine-Verifiable Proofs

In order to produce a machine-verifiable proof of unsatisfiability for a PB formula, we need a file that expresses the problem, and a second file that provides the proof. There is a standard format[1] for expressing pseudo-Boolean problems, which we use as a starting point. Briefly, each line in the file is either a comment (starting with an asterisk), or specifies a constraint. For example, the line

```
3 x1 + 2 ~x3 -3 x6 >= 2 ;
```

specifies that $3x_1 + 2\overline{x_3} - 3x_6 \geq 2$. For solver competitions, a series of additional guarantees are provided, such that the file will start with a special header comment, and that the variables will be named "x1" through "xN". Many pseudo-Boolean solvers treat these guarantees as requirements on the input, and will reject or misbehave if they are not followed, so these guarantees are *de facto* rules. Our tools *can* generate files following these restrictions, but do not have to do so: we find it more readable to be able to generate PB variable names like "xFoo_3" to correspond to a constraint programming variable "Foo" taking the value 3.

For logging proofs, we have created a new format. The format is a simple text file, which is at least somewhat human-readable, and which has been designed to reduce the amount of work required from solver implementers to a minimum. In particular, a key design choice is that solver writers will not need to maintain an entire pseudo-Boolean solver alongside their existing constraint programming solver, and can instead output proofs using a simpler template-based approach—we discuss this further in the following section.

Proof headers. The proof file must begin with a header line. Typically, this will immediately be followed by an "f" rule, as follows (the asterisk line is a comment and is ignored):

```
pseudo-Boolean proof version 1.0
* read in the 18 model constraints
f 18 0
```

This "f" rule instructs the proof verifier to read in the pseudo-Boolean model file. The "18" must correspond to the number of constraints in the problem, except that any "equals" constraint in the model is considered to be two inequalities instead. Each constraint read in is numbered, starting from 1. The zero is a line terminator.

Deriving constraints. Subsequent lines in a proof will use these numbered constraints, ultimately deriving a contradiction. The first way to do so is using a "p" rule, which takes an expression in reverse Polish notation, and creates a new numbered constraint with its result. For example, the line

```
p 42 3 * 43 + 2 d 0
```

[1]http://www.cril.univ-artois.fr/PB12/format.pdf

means "create a new constraint by multiplying the constraint numbered 42 by 3, then adding constraint 43, then dividing by 2"; again, the zero is a line terminator. The "p" rule can thus express any number of applications of the addition, multiplication, and division axioms as a single step—during development we found this to be much more convenient and compact than requiring a step per axiom application.

Literal axioms. The "p" rule may also be used to introduce literal axioms. For example, the line

```
p x1 ~x2 + 5 + 0
```

will create a new constraint by adding the literal axioms $x_1 \geq 0$ and $\overline{x_2} \geq 0$ to constraint number 5.

Reverse unit propagation. The "u" rule gives another way of introduction of a new constraint, which this time is given explicitly in OPB format. For example,

```
u -1 x8 -1 x25 -1 x26 -1 x5 >= -3 ;
```

would create a new numbered constraint saying

$$-x_8 + -x_{25} + -x_{26} + -x_5 \geq -3.$$

In order for such a constraint to be introduced, it must be an "obvious" consequence of the constraints known so far. Here "obvious" is defined to mean "follows by reverse unit propagation", as described in the previous section.

Although the "u" rule is theoretically no more powerful than the "p" rule, using this rule substantially reduces the implementation effort for solver authors. It avoids the need for solvers to understand pseudo-Boolean constraints to (e.g.) perform cancellations correctly, and instead offloads that work onto the proof verifier. It also avoids the need to explicitly log any steps for propagation of constraints which can be encoded into pseudo-Boolean form in a way where unit propagation gives the same propagation strength as the constraint.

Asserting contradiction. Once a contradiction has been derived, the "c" rule is used to verify that assertion and terminate the proof. So, a typical proof may end as follows:

```
u >= 1 ;
c 146750 0
```

Here the penultimate line asserts that contradiction ($0 \geq 1$) follows by unit propagation from the constraints learned so far, and the final line asserts that the previous constraint (which has number 146750) is in fact a contradiction.

Other rules. The proof format also supports other rules, including ways of deleting constraints (for reduced memory usage) and verifying solutions. These are explained in the documentation for our proof-checking tool, which we will now describe.

A Proof Checking Tool

We have implemented a proof checking tool for this proof format.[2] It is written in Python, with critical parts in C++ for performance reasons. The tool can also output a log of exactly what it is deriving at every stage of the verification process, which we have found to be tremendously helpful when debugging solvers.

Constraint Programming

In constraint programming, we have a more general problem to solve than in the pseudo-Boolean setting. We still have a set of variables, but now variables may take their values from a finite set, rather than being Boolean; we will use capital letters for constraint programming variables, to distinguish them from PB variables. We also have a set of constraints, but these may be in a variety of forms. This generality is a particular strength of constraint programming: in a single model, we may mix Boolean constraints, arithmetic constraints, and other "global" constraints such as "all different". The all different constraint operates on a set of variables of any size, and states that each variable in this set must be given a different value. A given problem could have a single all different constraint, which could operate over some or all of its variables, or it could have many all different constraints, each operating over a different (and potentially overlapping) subset of values. The all different constraint was one of the first global constraints to have a dedicated propagation algorithm (Régin 1994), and remains one of the core constraints present in any constraint programming toolkit—it therefore presents a good minimum standard that any proof logging system must be able to meet.

Compiling Constraint Programming

One approach to solving a constraint programming problem is to compile it to another format, such as Boolean or pseudo-Boolean satisfiability. A simple way of doing so is as follows: for each constraint programming variable X with domain $D(X)$, we create $|D(X)|$ Boolean variables. We then need constraints saying that at least one of these Boolean variables is set to true—this is a disjunction, which may be expressed directly in CNF, or as a single sum inequality in pseudo-Boolean notation. Then we need to mandate that at most one of these Boolean variables is set to true—this is also a sum inequality in pseudo-Boolean notation, but requires all-pairs binary constraints in CNF.

For example, given a constraint satisfaction problem with variables $W \in \{1,2,3\}$, $X \in \{2,3\}$, $Y \in \{1,3\}$, and $Z \in \{2,4\}$, we might compile this into OPB format as follows (we omit the header line):

```
* variable W in { 1 2 3 }
1 xW_1 1 xW_2 1 xW_3 >= 1 ;
-1 xW_1 -1 xW_2 -1 xW_3 >= -1 ;
* variable X in { 2 3 }
1 xX_2 1 xX_3 >= 1 ;
-1 xX_2 -1 xX_3 >= -1 ;
* variable Y in { 1 3 }
1 xY_1 1 xY_3 >= 1 ;
-1 xY_1 -1 xY_3 >= -1 ;
* variable Z in { 2 4 }
1 xZ_2 1 xZ_4 >= 1 ;
-1 xZ_2 -1 xZ_4 >= -1 ;
```

[2]https://github.com/StephanGocht/VeriPB/, https://doi.org/10.5281/zenodo.3548582

Note that for compatibility with pseudo-Boolean solvers, it would be better to use variable names "x1" through "x9"; our tools can also generate numbered variable names, but here will will use more descriptive variable names.

To compile a constraint programming not-equals constraint $X \neq Y$ into either CNF or pseudo-Boolean form, we post a "not both true" constraint for each value that appears in the intersection of the two domains. For example, we could encode $W \neq X$ in the above model using two constraints:

```
* W not equals X, value 2
-1 xW_2 -1 xX_2 >= -1 ;
* W not equals X, value 3
-1 xW_3 -1 xX_3 >= -1 ;
```

Note that no constraint appears for the value 1, which is only present in W's domain.

This suggests a very simple way of compiling an all different constraint: for each distinct pair of variables X and Y in the constraint's scope, we follow the steps to compile a $X \neq Y$ constraint. However, a much more compact encoding is possible in pseudo-Boolean form. For each value that appears in at least one domain, we post a constraint summing over every Boolean variable that corresponds to a CP variable in that constraint taking that value, saying that this sum is at most one. In other words, we are saying that each value can be used at most one time. For example, we could compile an all-different constraint over all four variables as:

```
-1 xW_1 -1 xY_1 >= -1 ;
-1 xW_2 -1 xX_2 -1 xZ_2 >= -1 ;
-1 xW_3 -1 xX_3 -1 xY_3 >= -1 ;
-1 xZ_4 >= -1 ;
```

(The final line could be deleted, because only Z can take the value 4, but leaving it in place reduces the number of special cases needed when implementing a solver.)

Other more sophisticated compilation methods exist, such as those described by Ohrimenko and Stuckey (2008) and Bessiere et al. (2009a). However, these methods are aimed at getting better performance out of solvers, whilst we need only a correct encoding for proof-logging purposes.

Propagators

Constraint programming solvers rarely use these decomposition methods. Instead, solvers have special algorithms called *propagators* associated with each constraint. A propagator can do two things (Schulte and Tack 2009):

1. It can signal that no solution is possible for its associated constraint, based upon the values remaining in the domains of the associated variables.
2. It can remove values from the domains of its associated variables.

A propagator may only remove a value from a domain if that value cannot occur in any solution to that constraint. A propagator which will always remove all such values is known as "achieving generalised arc consistency (GAC)" (or sometimes "domain consistency"). For some constraints, achieving GAC is either intractable or impractical, but for the all different constraint GAC may be achieved efficiently and practically (Régin 1994; Gent, Miguel, and Nightingale 2008). Furthermore, GAC for the all different constraint *cannot* be achieved by any polynomial-sized decomposition into Boolean satisfiability (Bessiere et al. 2009b). This is important in practice: there are many examples where strong propagation of constraints is the key to solving hard problems (e.g. Stergiou and Walsh 1999).

From Propagating to Justifying All-Different

The canonical GAC propagation algorithm was introduced by Régin (1994), and has seen considerable subsequent work on how to implement it as efficiently as possible (Gent, Miguel, and Nightingale 2008). We will briefly describe, without proofs, the basic (non-incremental) form of the algorithm, although everything we describe can also be applied to more modern highly tuned implementations. The algorithm works in two stages: firstly, it determines whether it is possible to satisfy the constraint at all, and then if it is, it finds the complete set of values which may safely be deleted from its variables.

Matchings and Hall violators. Let $\{X_1, \ldots, X_N\}$ be the set of variables in an all-different constraint. The *value graph* for this constraint is a bipartite graph, with a vertex in its left set for each variable X_n, and a vertex in its right set for each value that is present in at least one X_n's domain; there is an edge between a variable's vertex and a value's vertex if and only if that variable's domain contains that value. A *matching* is a set of edges in a bipartite graph such that no vertex appears as an endpoint of more than one edge; a matching is *left-saturating* if it covers every vertex on the left, and is of *maximum cardinality* if it contains as many edges as possible.

It is easy to see that left-saturating matchings in a value graph are in one-to-one correspondence with solutions to the all-different constraint. In particular, the constraint can be satisfied if and only if a maximum cardinality matching is left-saturating. Since finding a maximum cardinality matching may be done in polynomial time (Hopcroft and Karp 1973), it is easy to implement a propagator which checks whether or not the constraint is satisfiable.

We are now left with the problem of justifying a backtrack if we find that a maximum cardinality matching is not left-saturating. Using only resolution, this would require exponentially many steps (Haken 1985), but with pseudo-Boolean proofs we are in a better situation. We use Hall's (1935) marriage theorem, which states that a left-saturating matching exists in a bipartite graph if and only if for every subset $W \subseteq \{X_1, \ldots, X_N\}$ we have that $|W| \leq |N(W)|$, where $N(W)$ denotes the neighbourhood of W. In particular, if a left-saturating matching does not exist, then there exists a *Hall violator* W where $|N(W)| < |W|$; in our terms, this is a set of n variables whose domains contain strictly fewer than n values between them.

A conventional propagator does not care about the existence of Hall violators, and only looks at the size of a maximum cardinality matching. However, the usual augmenting paths algorithm for finding a maximum cardinality matching can easily be extended to output a Hall violator by following an alternating path backwards from an unmatched left-vertex.

Given such a set of variables H, a justifying propagator must be able to express that "either one of the variables in H must be given a value that is currently not present in its domain, or there is a contradiction". To do this, we count sets of variable-value pairs in two different ways. Firstly, we have (from the model) that each variable in H must be given at least one value—call these constraints $\mathit{AL1}(h)$. We sum together these constraints, to achieve an expression of the form $\sum_{h \in H} \mathit{AL1}(h) \geq |H|$. Now, letting $D(H)$ mean the values in the union of the domains of the variables in H, and denoting the "value can be used at most once" constraints from the model as $\mathit{AM1}(v)$, we sum these to get $\sum_{v \in D(H)} \mathit{AM1}(v) \leq |D(H)|$. Since H is a Hall violator, $|H| > |D(H)|$, so the sum of these two sums gives a suitable justification.

Continuing our running example, suppose that the Z variable could not take the value 4, due to it being eliminated by another constraint or by a guessed assignment during search. In this case, a maximum cardinality matching in the value graph would leave a single variable uncovered. Suppose the matching found is $\{W = 1, X = 2, Y = 3\}$ leaving Z uncovered. In this case, the Hall violator has the four variables $\{W, X, Y, Z\}$, and the three associated values are $\{1, 2, 3\}$. By summing up the lines saying

```
1 xW_1 1 xW_2 1 xW_3 >= 1 ;
1 xX_2 1 xX_3 >= 1 ;
1 xY_1 1 xY_3 >= 1 ;
1 xZ_2 1 xZ_4 >= 1 ;
-1 xW_1 -1 xY_1 >= -1 ;
-1 xW_2 -1 xX_2 -1 xZ_2 >= -1 ;
-1 xW_3 -1 xX_3 -1 xY_3 >= -1 ;
```

using a proof logging command which could look like (if the lines for the variable axioms for W, X, Y and Z are 1, 3, 5 and 7, and the all-different constraint starts on line 9):

```
p 1 3 + 5 + 7 + 9 + 10 + 11 + 0
```

we derive the constraint

```
1 xZ_4 >= 1 ;
```

which means that Z must take the value 4 after all—and if it cannot then we have proved unsatisfiability.

Strongly connected components and Hall sets. The second stage of the propagation process takes place only if a left-saturating matching has been found. If such a matching M exists, a new *directed* bipartite graph known as the *residual graph* is created by taking the value graph, and directing edges as right-to-left if they are present in M and left-to-right otherwise. This graph has the property that certain edges that start in one strongly connected component and end in another correspond to variable-value assignments that will never appear in any maximum cardinality matching—we refer to Gent, Miguel, and Nightingale (2008) for full details.

Again, we cannot directly express graph-theoretic properties in a proof log, but a connection between combinatorics and graph theory saves us. Every edge which describes a deletion is due to the existence of a Hall set—that is, a set of n variables whose union contains exactly n values (Quimper and Walsh 2005). More specifically, there is no solution to an all-different constraint where variable X_i gets value v_j if and only if there exists a set H of variables not including X_i whose domains contain exactly $|H|$ values between them, one of which is v_j.

Given a Hall set H, we may output a pseudo-Boolean constraint justifying the deletions it triggers by following the same process as for a Hall violator: we sum up the "variable must be given at least one value" constraints and the "value must be used at most once" constraints, and this time arrive at an equality which shows that no variable outside of the Hall set may be given any value in the Hall set.

It remains only to identify the relevant Hall sets, which is also straightforward: they correspond precisely to the strongly connected components in the residual graph which have a deletion edge entering them (Dulmage and Mendelsohn 1958). Note that a single Hall set can justify multiple deletions (and for space reasons it is advantageous to detect this and avoid emitting duplicate constraints).

Returning to our running example, the variables $\{W, X, Y\}$ form a Hall set with three values $\{1, 2, 3\}$. The Z variable also includes the value 2, which may be deleted. We may justify this by summing the lines

```
1 xW_1 1 xW_2 1 xW_3 >= 1 ;
1 xX_2 1 xX_3 >= 1 ;
1 xY_1 1 xY_3 >= 1 ;
-1 xW_1 -1 xY_1 >= -1 ;
-1 xW_2 -1 xX_2 -1 xZ_2 >= -1 ;
-1 xW_3 -1 xX_3 -1 xY_3 >= -1 ;
```

using a command like

```
p 1 3 + 5 + 9 + 10 + 11 + 0
```

to derive a new constraint

```
1 ~xZ_2 >= 1 ;
```

which corresponds to saying that Z may not take the value 2. If we had another variable Q with domain $\{2, 5, 6\}$, the constraint generated using the same sum would instead be

```
1 ~xZ_2 1 ~xQ_2 >= 2
```

showing that neither Z nor Q could take the value 2.

Finally, we note that Hall sets may nest. For example, given $A, B \in \{1, 2\}$, $C, D \in \{1, 2, 3, 4\}$, and $E \in \{1, 2, 3, 4, 5\}$, the process we describe would output Hall sets $\{A, B\}$ and $\{C, D\}$, *not necessarily in that order*. This does not matter for our purposes (so long as we are using the "u" proof rule rather than a series of "p" steps to describe the search tree, as discussed in the following section). However, if it would for some reason be preferable to output Hall sets $\{A, B\}$ and $\{A, B, C, D\}$ (which justify the same deletions from E, independently of the order in which they are carried out), this may be done by outputting every vertex in the residual graph which is reachable from the end of the deletion edge, rather than looking at strongly connected components.

Justifications Versus Explanations

Much of what we have discussed resembles the *explanations* produced by lazy clause generation constraint programming solvers (Ohrimenko, Stuckey, and Codish 2009; Downing, Feydy, and Stuckey 2012). Lazy clause generation solvers will create new clauses on the fly as the result of propagations, allowing for SAT-style conflict analysis to be mixed with constraint programming propagation. However, explanations can be created "out of nowhere" without justification: an explaining propagator merely asserts that the clause it produces is valid, and does not have to demonstrate its derivation. Nonetheless, there is potential for crossover between these two areas going forward: in one direction, perhaps generating more expressive PB constraints and using PB conflict analysis will lead to better lazy clause generation solvers, and in the other direction, it may be possible to reduce the amount of work needed to produce justifying propagators by building upon what is known about explanations.

Another related piece of work is the constraint programming solver described by Veksler and Strichman (2010), which fits somewhere in between justifications and explanations. This solver produces proof logs, but in a format which requires the proof verifier to support specialised inference rules for every new global constraint. In contrast, our approach shows that practical proof logging is still possible even without requiring the proof verifier to know about the propagation behaviour of any global constraint.

A Justifying Constraint Programming Solver

Finally, we briefly describe the implementation of a small constraint programming solver which can output a justification of all of the choices it makes.[3] This solver is implemented in C++, and supports the all different constraint with full GAC propagation, as well as equals, not-equals, and (forward checking) table constraints. The solver has not been designed for performance, but rather to identify the best engineering decisions for implementing a proof logging solver.

The solver differs from a conventional constraint programming solver in three areas: being able to compile models to the pseudo-Boolean format, being able to log search operations, and being able to log propagation.

Compilation. As well as solving a constraint programming model, a proof-logging solver must be able to trans-

[3]https://github.com/ciaranm/certified-constraint-solver, https://doi.org/10.5281/zenodo.3549712

late the model into an equivalent pseudo-Boolean model. We described the theory behind this in the previous section. From an implementation perspective, this was reasonably straightforward: the solver must track the PB variable naming used for each variable-value which it encodes, and some constraints must remember the line number used when outputting some of their rules (for example, the all different constraint must be able to recall the appropriate "at most one" line for each of the values in its scope).

Search. Proof logging during search is remarkably simple when using reverse unit propagation. Whenever the solver backtracks (either due to propagation failure, or a domain wipeout), it suffices to output a proof line of the form

```
u -1 xA_3 -1 xB_4 -1 xC_1 >= -2 ;
```

where the variables are the decision variables on the solver's trail—that is, the solver is asserting that whatever it just guessed "obviously" leads to a contradiction, and so at least one of the guessed assignments must be incorrect. Ultimately, this leads to the solver outputting

```
u >= 1 ;
```

after backtracking on the first decision variable, which can then be followed by the assertion of contradiction.

Propagation. Due to reverse unit propagation, it is not necessary to make any changes at all for the equals and not-equals constraints—these propagations need not be logged. For the all different constraint, it suffices to output "p" rules for every Hall violator and Hall set which leads to a contradiction or propagation, as described in the previous section. (Again due to the use of reverse unit propagation, there is no need to adapt these constraints to mention the trail.)

We believe this demonstrates the simplicity of proof logging in this format. In earlier prototypes not making use of reverse unit propagation, the burden upon the solver writer was *vastly* greater, with a trail-aware "p" rule being required for every single propagating step.

Experiments

To test our solver and proof verifier, we generated a number of 25×25 unsatisfiable Sudoku instances. Solving such instances in a reasonable amount of time requires the full capabilities of all-different propagation, and tests all of the functionality of our tools. For a representative instance which required 1,691 guessed decisions to solve, our solver took 41 seconds to prove unsatisfiability, which increased to 42 seconds when logging a proof (we stress that this implementation has not been designed for performance). The proof log contained 109,519 Hall set propagations, and 846 Hall violators, and could be verified in 6 seconds.

We also tried deliberately introducing bugs into our solver—for example, by failing to find maximum cardinality matchings that required two augmenting steps, and by randomly omitting logging for a small number of Hall sets. In each case the proof verifier caught the mistakes, although only if the instances selected actually triggered the faulty behaviour. (For example, it is surprisingly rare for multiple augmenting path steps to be required to find a maximum cardinality matching, when starting from a greedy matching.) Because our solver was designed from the ground up with proof logging, we were also able to use proof logs to catch bugs early on in the development process that had not been detected by conventional testing techniques.

Conclusion

We have shown that it is both possible and practical for a constraint programming solver to produce a pseudo-Boolean proof log for unsatisfiability, even when all different constraints are in use. This is unexpected: pseudo-Boolean reasoning knows nothing about graphs, matchings, augmenting paths, or strongly connected components, all of which are required for all different propagation. This suggests that we should be more broadly interested not just in algorithms for propagation, but in languages for justifying propagation—unlike in the Boolean satisfiability community, these concepts are not equivalent. We therefore intend to investigate which other families of global constraint can be justified easily using pseudo-Boolean reasoning. Obviously, any constraint for which we already know a strongly-propagating SAT or pseudo-Boolean encoding requires no further work, but we believe that several other common constraints are also justifiable.

One might ask whether a new proof format is really necessary. The main difference to the existing DRAT proof format used by SAT solvers is that we are using cutting planes proofs instead of resolution proofs. This makes it very simple to express the counting arguments we are using to justify propagations and conflict of the all-different constraint. This kind of reasoning cannot be done efficiently with resolution. However, DRAT allows the introduction of new variables, which is known to be very powerful. (Our proof format currently does not have this capability but an extension is in progress.) In theory, using additional variables allows DRAT to verify cutting planes reasoning and hence to justify the all-different constraint.

A natural and interesting question, therefore, is how DRAT proof logging would compare to our approach. As we already mentioned previously, though, the problem is that DRAT proof logging for cardinality reasoning is a *theoretical* result. The fact that DRAT logging can be done in principle, with at most a polynomial blow-up, does not mean that it is possible to do in practice, and to the best of our knowledge no-one has been able to produce any implementation that can be used to run practical experiments. This means that we cannot compare the performance of our pseudo-Boolean proof logging with DRAT proof logging, not because DRAT would run so much more slowly, but because it is so much more complicated that no-one has even implemented it. In contrast, our pseudo-Boolean proof logging is both fast and simple.

The approach we describe does still require the user to trust that the pseudo-Boolean model file produced corresponds exactly to the high level constraint programming model given as input. This should not necessarily be taken as given—not all global constraints can be encoded in as straightforward a manner as all different, and additionally the compilers for higher-level constraint modelling languages such as Essence and MiniZinc could introduce further bugs. It may therefore be worthwhile to investigate techniques from conventional compilers to verify this part of the process.

We stress that our approach does not prove that a solver is correct—it simply ensures that if a solver ever produces an incorrect answer, then this can be detected and a human brought in to fix the problem. On the other hand, when a justifying solver does produce a correct answer by legitimate means, the proof can be archived for posterity. We can thus always be confident that the answer is indeed correct, even if we do not trust the solver that produces the proof or the person who is claiming that the proof was produced by a trustworthy solver. And finally, we note that proof logging will catch more esoteric problems such as compiler bugs, hardware errors, and cosmic rays that could make a correct solver output an incorrect answer.

Acknowledgments

The first and second authors were funded by the Swedish Research Council (VR) grant 2016-00782. The fourth author was also supported by the Knut and Alice Wallenberg grant KAW 2016.0066 and the VR grant 621-2012-5645. The third author was supported by the Engineering and Physical Sciences Research Council [grant number EP/P026842/1].

References

Akgün, Ö.; Gent, I. P.; Jefferson, C.; Miguel, I.; and Nightingale, P. 2018. Metamorphic testing of constraint solvers. In Hooker, J. N., ed., *Principles and Practice of Constraint Programming - 24th International Conference, CP 2018, Lille, France, August 27-31, 2018, Proceedings*, volume 11008 of *Lecture Notes in Computer Science*, 727–736. Springer.

Bessiere, C.; Katsirelos, G.; Narodytska, N.; Quimper, C.; and Walsh, T. 2009a. Decompositions of all different, global cardinality and related constraints. In Boutilier (2009), 419–424.

Bessiere, C.; Katsirelos, G.; Narodytska, N.; and Walsh, T. 2009b. Circuit complexity and decompositions of global constraints. In Boutilier (2009), 412–418.

Blake, A. 1937. *Canonical Expressions in Boolean Algebra*. Ph.D. Dissertation, University of Chicago.

Boutilier, C., ed. 2009. *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*.

Buss, S., and Nordström, J. 2019. Proof complexity and SAT solving. Chapter to appear in the 2nd edition of *Handbook of Satisfiability*, Draft version available at https://www.math.ucsd.edu/~sbuss/ResearchWeb/ProofComplexitySAT/.

Cook, W. J.; Coullard, C. R.; and Turán, G. 1987. On the complexity of cutting-plane proofs. *Discrete Applied Mathematics* 18(1):25–38.

Davis, M., and Putnam, H. 1960. A computing procedure for quantification theory. *J. ACM* 7(3):201–215.

Downing, N.; Feydy, T.; and Stuckey, P. J. 2012. Explaining alldifferent. In Reynolds, M., and Thomas, B. H., eds., *Thirty-Fifth Australasian Computer Science Conference, ACSC 2012, Melbourne, Australia, January 2012*, volume 122 of *CRPIT*, 115–124. Australian Computer Society.

Dulmage, A. L., and Mendelsohn, N. S. 1958. Coverings of bipartite graphs. *Canadian Journal of Mathematics* 10:517–534.

Gent, I. P.; Miguel, I.; and Nightingale, P. 2008. Generalised arc consistency for the alldifferent constraint: An empirical survey. *Artif. Intell.* 172(18):1973–2000.

Gillard, X.; Schaus, P.; and Deville, Y. 2019. Solvercheck: Declarative testing of constraints. In *CP 2019, Proceedings of the 25th International Conference on Principles and Practice of Constraint Programming*. To appear.

Goldberg, E. I., and Novikov, Y. 2003. Verification of proofs of unsatisfiability for CNF formulas. In *Design, Automation and Test in Europe Conference (DATE)*, 10886–10891. IEEE Computer Society.

Haken, A. 1985. The intractability of resolution. *Theor. Comput. Sci.* 39:297–308.

Hall, P. 1935. On representatives of subsets. *Journal of the London Mathematical Society* s1-10(1):26–30.

Heule, M.; Hunt Jr., W. A.; and Wetzler, N. 2013a. Trimming while checking clausal proofs. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, 181–188. IEEE.

Heule, M.; Hunt Jr., W. A.; and Wetzler, N. 2013b. Verifying refutations with extended resolution. In Bonacina, M. P., ed., *Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings*, volume 7898 of *Lecture Notes in Computer Science*, 345–359. Springer.

Heule, M.; Järvisalo, M.; and Suda, M. 2019. The international SAT Competitions web page. http://www.satcompetition.org.

Heule, M. J. H.; Kullmann, O.; and Marek, V. W. 2016. Solving and verifying the boolean pythagorean triples problem via cube-and-conquer. In Creignou, N., and Berre, D. L., eds., *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*, volume 9710 of *Lecture Notes in Computer Science*, 228–245. Springer.

Hopcroft, J. E., and Karp, R. M. 1973. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM J. Comput.* 2(4):225–231.

Lamb, E. 2016. Two-hundred-terabyte maths proof is largest ever. *Nature* 545:17–18.

Marques Silva, J. P., and Sakallah, K. A. 1999. GRASP: A search algorithm for propositional satisfiability. *IEEE Trans. Computers* 48(5):506–521.

Ohrimenko, O., and Stuckey, P. J. 2008. Modelling for lazy clause generation. In Harland, J., and Manyem, P., eds., *Theory of Computing 2008. Proc. Fourteenth Computing: The Australasian Theory Symposium (CATS 2008), Wollongong, NSW, Australia, January 22-25, 2008. Proceedings*, volume 77 of *CRPIT*, 27–37. Australian Computer Society.

Ohrimenko, O.; Stuckey, P. J.; and Codish, M. 2009. Propagation via lazy clause generation. *Constraints* 14(3):357–391.

Quimper, C., and Walsh, T. 2005. The all different and global cardinality constraints on set, multiset and tuple variables. In Hnich, B.; Carlsson, M.; Fages, F.; and Rossi, F., eds., *Recent Advances in Constraints, Joint ERCIM/CoLogNET International Workshop on Constraint Solving and Constraint Logic Programming, CSCLP 2005, Uppsala, Sweden, June 20-22, 2005, Revised Selected and*

Invited Papers, volume 3978 of *Lecture Notes in Computer Science*, 1–13. Springer.

Régin, J. 1994. A filtering algorithm for constraints of difference in CSPs. In Hayes-Roth, B., and Korf, R. E., eds., *Proceedings of the 12th National Conference on Artificial Intelligence, Seattle, WA, USA, July 31 - August 4, 1994, Volume 1.*, 362–367. AAAI Press / The MIT Press.

Schulte, C., and Tack, G. 2009. Weakly monotonic propagators. In Gent, I. P., ed., *Principles and Practice of Constraint Programming - CP 2009, 15th International Conference, CP 2009, Lisbon, Portugal, September 20-24, 2009, Proceedings*, volume 5732 of *Lecture Notes in Computer Science*, 723–730. Springer.

Stergiou, K., and Walsh, T. 1999. The difference all-difference makes. In Dean, T., ed., *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence, IJCAI 99, Stockholm, Sweden, July 31 - August 6, 1999. 2 Volumes, 1450 pages*, 414–419. Morgan Kaufmann.

Van Gelder, A. 2008. Verifying RUP proofs of propositional unsatisfiability. In *10th International Symposium on Artificial Intelligence and Mathematics (ISAIM)*. http://isaim2008.unl.edu/index.php?page=proceedings.

Veksler, M., and Strichman, O. 2010. A proof-producing CSP solver. In Fox, M., and Poole, D., eds., *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, July 11-15, 2010*. AAAI Press.

Wetzler, N.; Heule, M.; and Hunt Jr., W. A. 2014. Drat-trim: Efficient checking and trimming using expressive clausal proofs. In Sinz, C., and Egly, U., eds., *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, volume 8561 of *Lecture Notes in Computer Science*, 422–429. Springer.