Subgraph Isomorphism Meets Cutting Planes: Solving With Certified Solutions

Stephan Gocht[1,2] , **Ciaran McCreesh**[3] and **Jakob Nordström**[2,1]
[1]Lund University, Lund, Sweden
[2]University of Copenhagen, Copenhagen, Denmark
[3]University of Glasgow, Glasgow, Scotland
stephan.gocht@cs.lth.se, ciaran.mccreesh@glasgow.ac.uk, jn@di.ku.dk

Abstract

Modern subgraph isomorphism solvers carry out sophisticated reasoning using graph invariants such as degree sequences and path counts. We show that all of this reasoning can be justified compactly using the cutting planes proofs studied in complexity theory. This allows us to extend a state of the art subgraph isomorphism enumeration solver with proof logging support, so that the solutions it outputs may be audited and verified for correctness and completeness by a simple third party tool which knows nothing about graph theory.

1 Introduction

The subgraph isomorphism decision problem is to find a copy of a small "pattern" graph inside a larger "target" graph, or to show that no such copy exists; the enumeration problem is to find all copies. These problems occur in many applications—we refer to Archibald *et al.* [2019] for a partial list. Although the problems are NP- and #P-complete respectively, a series of algorithms based upon constraint programming [Zampelli *et al.*, 2010; Solnon, 2010; Audemard *et al.*, 2014; McCreesh and Prosser, 2015; Archibald *et al.*, 2019] have culminated in a practical way of tacking all but the hardest instances [McCreesh *et al.*, 2018; Solnon, 2019]. These algorithms exploit various combinatorial and graph invariants, such as matchings, degree sequences, and number of paths between vertices, in a bid to reduce the number of combinations which must be considered. As a result, the solvers implementing these algorithms are rather complex, and even after extensive testing it is hard to be convinced that the solvers are definitely free from bugs.

This paper discusses proof logging as a way of verifying the solutions produced by of one of these solvers: the idea is that the solver is modified to produce a "certificate" or proof file as part of its output, which can then be verified by a (much simpler) external tool. For satisfiable decision instances for NP problems, such certificates are always small, and (usually) easy to check. For demonstrating unsatisfiability, or for showing that a solver has not missed any solutions when enumerating, no way of guaranteeing short certificates is known. However, theoretical worst cases are overly pessimistic, and modern subgraph isomorpism solvers often perform much better than exponential worst-case performance bounds would suggest. In the same way, we will show that with the right proof format, certificates can be simple to verify, yet still only be proportional in size to the amount of work carried out by a solver.

We stress that proof logging does not prove that a solver is correct: unless a solver actually exhibits buggy behaviour when producing a proof, a proof verifier will not complain. On the other hand, proof verifiers will detect if a correct solver is run on faulty hardware or is compiled with a buggy compiler, if that leads to the solver performing incorrect reasoning. In other words, proof logging gives us a way of trusting solver *outputs*, not solvers.

Proof logging in the Boolean satisfiability community is usually done using a format known as DRAT [Heule *et al.*, 2013b; Heule *et al.*, 2013a; Wetzler *et al.*, 2014]. Recently, Elffers *et al.* [2020] proposed a different proof-logging format based upon cutting planes proofs for pseudo-Boolean models, and showed that (unlike DRAT) it could easily handle the all-different reasoning used in constraint programming solvers. Because subgraph isomorphism solvers also make use of strong all-different reasoning and similar counting arguments, we will be using this format. Our first contribution is to show that cutting planes proofs are also powerful enough to compactly express reasoning about graph degrees, neighbourhood degree sequences, and counts of short paths in graphs. This is sufficient to represent all of the reasoning carried out by the Glasgow Subgraph Solver [Archibald *et al.*, 2019], which is the current strongest subgraph isomorphism solver on hard instances [Solnon, 2019]. This is a surprising result: cutting planes proofs know nothing about graphs, and the solver's inference algorithms were not designed with proof logging in mind. Our second contribution is to demonstrate that this approach is actually practical: we extend the Glasgow solver with proof logging support, and produce and verify solution certificates for over a thousand standard enumeration benchmark instances.

2 Background

We begin by introducing notation, and providing the necessary background on graphs and on pseudo-Boolean formulae.

Graphs. Let G be a graph with vertex set $\mathrm{V}(G)$, and let $v \in \mathrm{V}(G)$. We write $\mathrm{N}(v)$ for the neighbourhood of (set of vertices adjacent to) v, and $\deg(v)$ for the cardinality of the neighbourhood; we write $\deg(G)$ for the mean of the degrees of all vertices of G. For simplicity, every graph appearing in this paper is undirected, unlabelled, and does not have loops (vertices adjacent to themselves), although every result we describe can be extended to these more general cases.

Subgraph isomorphism. Given a *pattern* graph P and a *target* graph T, the *non-induced subgraph isomorphism problem* is to find an injective mapping from $\mathrm{V}(P)$ to $\mathrm{V}(T)$ such that adjacent vertices in P are mapped to adjacent vertices in T. The *induced* problem additionally requires that non-adjacent vertices be mapped to non-adjacent vertices—again, we do not discuss this further in this paper, although our results are also easily applicable to this problem. The *enumeration* problem is to find every such mapping. (Some works instead consider the *unlabelled enumeration* variant, defined as finding every *image* of such a mapping.)

Pseudo-Boolean formulae. A pseudo-Boolean (PB) formula consists of a set of $\{0, 1\}$-valued variables $\{x_1, \ldots, x_n\}$ together with a set of linear constraints $\sum_{i=1}^{n} a_i \ell_i \geq A$, where each a_i and A is an integer, and each ℓ_i is either a literal x_i or a negated literal $\overline{x}_i$, where $x_i + \overline{x}_i = 1$. We can convert a Boolean satisfiability (SAT) problem instance in conjunctive normal form (CNF) into a PB formula because, e.g. $x_1 \vee x_2 \vee \overline{x}_3$ is satisfied iff $x_1 + x_2 + \overline{x}_3 \geq 1$, but in general the PB format is exponentially more expressive.

Cutting planes proofs. The *cutting planes* (CP) proof system [Cook *et al.*, 1987] allows us to reason about the satisfiability or unsatisfiability of a PB formula, in a similar way to the resolution system for SAT. Briefly, starting with the input constraints, we may generate new constraints by adding existing constraints, multiplying them by an integer constant, dividing by a positive integer constant (with rounding up), and introducing literal constraints $\ell_i \geq 0$. The VeriPB tool [Elffers *et al.*, 2020] provides a way of encoding CP proofs in such a way that they can be verified by machine: we refer to the tool's documentation[1] for details.

Simplification and reverse unit propagation. As well as accepting manual derivations of new constraints from existing ones, VeriPB has two ways of introducing a constraint which is specified explicitly during a proof log. The first is if the new constraint is semantically implied by an existing constraint (that is, if it may be obtained by weakening coefficients and cancelling literals). The second is through *reverse unit propagation* (RUP) [Goldberg and Novikov, 2003; Elffers *et al.*, 2020]: if the negation of the new constraint combined with every existing constraint is "obviously" unsatisfiable through unit propagation, then the new constraint may be added. Note that RUP constraints add no new expressive power and can be relatively expensive for the verifier, but using them appropriately can make solver implementation vastly more straightforward.

Logging of solutions. To support enumeration problems, VeriPB allows solutions to be logged. These are checked as they are encountered, and then their negations are added as new constraints. Thus, a proof log for an enumeration problem is effectively a list of solutions, plus an unsatisfiability proof showing there are no further solutions that were missed.

[1]https://github.com/StephanGocht/VeriPB/

3 Reasoning about Subgraphs

We will now demonstrate that all of the preprocessing and reasoning carried out by the Glasgow subgraph solver can be justified compactly using CP proofs. We will discuss all of the kinds of reasoning carried out by the Glasgow solver, but we will not describe precisely how these different steps fit together to make an algorithm—we refer to McCreesh and Prosser [2015] and Archibald *et al.* [2019] for those details. We will also touch upon kinds of reasoning carried out by other subgraph solvers, showing the generality and limitations of these results. However, because the VeriPB tool only understands PB formulae, we must first explain how we encode a subgraph isomorphism problem as a PB formula.

3.1 A Pseudo-Boolean Encoding

In the common constraint programming encoding for subgraph isomorphism used by the Glasgow solver, we have a variable for each vertex in the pattern graph, and each domain ranges over the vertices of the target graph. In other words, we are building a mapping from the pattern graph to the target graph. In a pseudo-Boolean model, we replace each constraint programming variable with a set of Boolean variables, one for each value in its domain—each of these variables $x_{p,t}$ represents a pair consisting of a pattern vertex p and a target vertex t, and is set to true precisely if p is to be mapped to t. Conveniently for the enumeration problem, solutions to this PB formula will be in one-to-one correspondence with solutions to the actual problem.

Our first set of constraints says that each pattern vertex must be mapped to exactly one target vertex:

$$\sum_{t \in \mathrm{V}(T)} x_{p,t} \geq 1 \qquad p \in \mathrm{V}(P)$$

$$\sum_{t \in \mathrm{V}(T)} -x_{p,t} \geq -1 \qquad p \in \mathrm{V}(P)$$

We then express injectivity, by saying that each target vertex may be used at most once:

$$\sum_{p \in \mathrm{V}(P)} -x_{p,t} \geq -1 \qquad t \in \mathrm{V}(T)$$

Finally, we must express the adjacency constraints. The most obvious way to do this is by saying that edges cannot be mapped to non-edges:

$$-x_{p,t} + -x_{q,u} \geq -1 \qquad \begin{array}{l} p \in \mathrm{V}(P),\ q \in \mathrm{N}(p), \\ t \in \mathrm{V}(T),\ u \in \mathrm{V}(T) \setminus \mathrm{N}(t) \end{array}$$

However, it is more convenient and compact (particularly if the target graph is sparse) to reformulate this by saying that if a vertex p is mapped to a vertex t, then every vertex in the neighbourhood of p must be mapped to a vertex in the neighbourhood of t:

$$\overline{x}_{p,t} + \sum_{u \in \mathrm{N}(t)} x_{q,u} \geq 1 \quad p \in \mathrm{V}(P),\ q \in \mathrm{N}(p),\ t \in \mathrm{V}(T)$$

All of these constraints must then be expressed in the OPB file format[2]. For readability, VeriPB allows us to name $x_{p,t}$ variables with strings like `xp_t`, whereas many PB solvers accept only numerical variable names like `x123`. Because the encoding process is not verifiable and the verifier cannot detect bugs in the encoding process, we keep the encoding as simple as possible and do not perform any reasoning here.

Note that this encoding has $O(|\mathrm{V}(P)|\deg(P)|\mathrm{V}(T)|)$ constraints, and that each adjacency constraint is potentially $O(|\mathrm{V}(T)|)$ long (although for sparse target graphs the length will be considerably shorter). The Glasgow solver instead represents the constraints using two bitset matrices, requiring only $O(|\mathrm{V}(P)|^2 + |\mathrm{V}(T)|^2)$ size, whilst some other solvers use even smaller adjacency list representations. This does provide us with a fairly moderate limit on the size of graphs with which we may use this verification process, compared to what subgraph solvers can handle. This is also one of the reasons that feeding such an input to a pseudo-Boolean (or Boolean satisfiability) solver is not a particularly good way of solving the problem in practice: dedicated subgraph isomorphism solvers give *much* better performance. However, the proof logs we will produce will correspond to a sequence of steps which *could*, in theory, have been carried out by a pseudo-Boolean solver working on these models.

3.2 Adjacency and Backtracking Search

Elffers *et al.* [2020] described how the recursive calls carried out by a standard backtracking constraint programming search algorithm could be logged using reverse unit propagation constraints, requiring one RUP constraint for every backtrack. They point out that with this approach, there is no need to log any inference steps carried out by a logging solver if they are no stronger than unit propagation on the associated OPB model. The Glasgow solver only performs inference on adjacency constraints when it is decided that a specific pattern vertex must be mapped to a specific target vertex, and so the following proposition is immediate.

Proposition 1. PB unit propagation on adjacency constraints carries out the same reasoning as the Glasgow solver, and so requires no explicit logging when using RUP.

This result, combined with a limited application of all-different justification [Elffers *et al.*, 2020], is already enough to deal with simple-but-fast subgraph isomorphism solvers like RI [Bonnici *et al.*, 2013] and VF2 [Cordella *et al.*, 2004] which do not use constraint programming techniques and which do not perform any further strong inference during search. To log the behaviour of "cleverer" constraint programming style algorithms like the Glasgow solver, however, we need to be able to justify several other kinds of preprocessing and reasoning. In the same way that Elffers *et al.* [2020] produced proofs by combining RUP with additional explicitly-derived constraints for verifying all-different reasoning in a constraint programming solver, we next show how to provide the verifier with enough additional information that every variable-value deletion in the subgraph solver will be reflected in the PB representation following RUP.

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

3.3 Reasoning About Degrees

A pattern vertex p of degree $\deg(p)$ can never be mapped to a target vertex t of degree $\deg(p) - 1$ or lower in any subgraph isomorphism. Expressing this fact using resolution proofs would require exponential length [Haken, 1985], but in cutting planes a proof may easily be derived. We demonstrate this by example. Suppose $\mathrm{N}(p) = \{q, r, s\}$ and $\mathrm{N}(t) = \{u, v\}$ for some pattern vertex p and target vertex t: we wish to derive $\overline{x}_{p,t} \geq 1$. We start with the three adjacency constraints,

$$\begin{aligned}\overline{x}_{p,t} + x_{q,u} + x_{q,v} &\geq 1\\ \overline{x}_{p,t} + x_{r,u} + x_{r,v} &\geq 1\\ \overline{x}_{p,t} + x_{s,u} + x_{s,v} &\geq 1,\end{aligned}$$

whose sum is

$$3\overline{x}_{p,t} + x_{q,u} + x_{q,v} + x_{r,u} + x_{r,v} + x_{s,u} + x_{s,v} \geq 3.$$

Observe that due to injectivity, at most one of the column $x_{q,u}$, $x_{r,u}$, and $x_{s,u}$ can be true, and similarly for the column of $x_{-,v}$ variables. Adding the injectivity constraints for target vertices u and v to the sum of the adjacency constraints gives

$$3\overline{x}_{p,t} + \sum_{p \in \mathrm{V}(P) \setminus \{q,r,s\}} -x_{p,u} + \sum_{p \in \mathrm{V}(P) \setminus \{q,r,s\}} -x_{p,v} \geq 1,$$

which is *almost* what we want except that we have acquired some additional variables from the injectivity constraints. This is not a problem: we can remove these stray $x_{p,-}$ variables by adding literal axioms (since $x_i \geq 0$ for any variable x_i) and then finally divide the resulting expression by 3, to obtain $\overline{x}_{p,t} \geq 1$ as desired. In proof logging terms, this whole process can be expressed in a single "p" ("reverse Polish expression") rule in the VeriPB format, optionally followed by an "e" ("equals") rule for sanity-checking purposes. With added line breaks and comments, this could look like:

```
p 18 19 + 20 +       * sum adj constraints
  12 + 13 +          * sum inj constraints
  xp_u + xp_v +      * cancel stray xp_*
  xo_u + xo_v +      * cancel stray xo_*
  3 d 0              * divide, and we're done
e 74 1 ~xp_t >= 1 ; * check what we just did
```

Alternatively, because the desired constraint only using weakening of coefficients and cancellation of literals, we may use a "j" ("implies and add") rule to avoid listing the steps explicitly:

```
p 18 19 + 20 +       * sum adj constraints
  12 + 13 + 0        * sum inj constraints
j 74 1 ~xp_t >= 1 ; * and simplify the above
```

In general, following the above process can justify *any* degree reasoning step:

Proposition 2. If a pattern vertex p cannot be mapped to a target vertex t due to degree, then we can justify this using a single "p" rule containing $\deg(p)+\deg(t)$ additions of model axioms, and a single "j" rule.

The Glasgow solver does not just reason about degrees, though: it also reasons about *global* and *neighbourhood degree sequences*, using a result due to Zampelli *et al.* [2010].

Let $\mathrm{S}(v)$ be the sequence consisting of the degrees of the neighbours of vertex v, from largest to smallest. Then a pattern vertex p can only be mapped to a target vertex t if $\mathrm{S}(p)$ is pointwise less than or equal to $\mathrm{S}(t)$. Similarly, if the sorted global degree sequence of the pattern graph as a whole is not pointwise less than or equal to the sorted global degree sequence of the target graph, then the problem is unsatisfiable.

Proposition 3. We may justify unsatisfiability due to global degree sequence reasoning using no more than $O(|\mathrm{V}(P)| + |\mathrm{V}(T)|)$ extra steps, following degree reasoning.

To do this, let i be the position of the first mismatch of the sequence. We first perform degree reasoning to eliminate the ith and all subsequent lower degree target vertices from the first i pattern vertices in the sequence. Then the first i pattern vertices and the first $i-1$ target vertices in the sequence form a Hall violator, which may be used to demonstrate unsatisfiability following the process described by Elffers *et al.* [2020].

Neighbourhood degree sequence reasoning is also expressible as a CP proof—we will demonstrate by example. Suppose a pattern vertex p has neighbourhood degree sequence $(5, 4, 3, 1)$ from neighbours (q, r, s, o), and a target vertex t has neighbourhood degree sequence $(5, 4, 2, 2)$ from vertices (u, v, w, x). In this case, the third item in the degree sequence is the first mismatch, so we will sum up the adjacency constraints for the first three pattern vertices to get

$$\begin{aligned} 3\overline{x}_{p,t} \quad & +x_{q,u} + x_{q,v} + x_{q,w} + x_{q,x} \\ & +x_{r,u} + x_{r,v} + x_{r,w} + x_{q,x} \\ & +x_{s,u} + x_{s,v} + x_{s,w} + x_{s,x} \qquad \geq 3. \end{aligned}$$

Now observe that, because the mismatch starts at the third item in the sequence, the third and subsequent columns of $x_{-,w}$ and $x_{-,x}$ variables all correspond to assignments which are impossible due to degree. We may therefore remove these variables by adding in the $\overline{x}_{-,w} \geq 1$ clauses created using the steps in the previous subsection. This leaves us with

$$3\overline{x}_{p,t} + x_{q,u} + x_{q,v} + x_{r,u} + x_{r,v} + x_{s,u} + x_{s,v} \geq 3.$$

At this point, we are in a very similar situation to with degree reasoning, above: the $x_{-,u}$ and $x_{-,v}$ sets of variables can both contribute at most one to the sum, due to injectivity. So, we add the injectivity constraints as before, and then either explicitly eliminate stray variables and divide, or simply ask the proof verifier to derive the exact constraint by implication from this sum. By generalising this example, we can conclude the following proposition.

Proposition 4. If a pattern vertex p cannot be mapped to a target vertex t due to neighbourhood degree sequence, then we can justify this using a single "p" rule containing no more than $\deg(p)+\deg(t)$ additions of model axioms, and no more than $\deg(t)|\mathrm{V}(P)|$ additions of previously derived rules, followed by a single "j" rule for simplification.

Zampelli *et al.* [2010] also make use of *dynamic* degree sequences in their solver: if a target vertex no longer appears in the domain of any pattern vertex (either initially, or dynamically inside search), then it is considered deleted and degrees and degree sequences are recalculated. The Glasgow solver does not use this inference, but if it did it would not be a problem for proof logging, as we would simply add in the derived constraints showing that no pattern vertex can be mapped to the target vertex in question. In the same way as for the all-different constraint, reverse unit propagation will automatically handle the current set of guessed assignments.

3.4 Reasoning About Paths

Audemard *et al.* [2014] implemented a solver named SND which propagated based upon distances as well as adjacency: if the distance between two pattern vertices p and q is d, and they are mapped to target vertices t and u respectively, then the distance between t and u must be no more than d. This filtering was refined in an early iteration of the Glasgow solver [McCreesh and Prosser, 2015] and in the PathLAD solver [Kotthoff *et al.*, 2016] as follows: call two vertices v and w *[k,d]-adjacent* if they have at least k simple paths of *exactly* length d between them. Then if p and q are [k,d]-adjacent for any k and d, then t and u must also be [k,d]-adjacent. This form of filtering is extremely expensive computationally if d and k are arbitrary, so the current version of the Glasgow subgraph solver uses only $d = 2$ and $k \leq 4$.

Instead of using path counts directly for filtering, the Glasgow solver generates additional sets of graph pairs $P^{[k,d]}$ and $T^{[k,d]}$, which have the same vertex sets but with vertices v and w adjacent in $G^{[k,d]}$ if they are [k,d]-adjacent in G. The solver then uses adjacency, degree, and degree sequence reasoning over all of these graph pairs, in a way which requires the full strength of the following proposition.

Proposition 5. For fixed k, for every pair of vertices p and q that are [k,2]-adjacent in P, and for every target vertex t, PB reasoning can derive in polynomial length a new constraint in *exactly* the form $\overline{x}_{p,t} + \sum_u x_{q,u} \geq 1$, where the u sum ranges over vertices that are [k,2]-adjacent to t.

This process is somewhat intricate, so we give only a sketch of how it works. First we establish that if p maps to t, then q maps to a vertex which is a walk of length two away from t, by summing each (p, r, t) adjacency constraint for r in $\mathrm{N}(p) \cap \mathrm{N}(q)$. We then resolve this with each (r, q, u) adjacency constraint in turn for each $u \in \mathrm{N}(t)$, and simplify. We then use injectivity and a second simplification step to strengthen the generated constraint to paths of length two. This requires two expressions with $O(|\mathrm{V}(P)||\mathrm{V}(T)|)$ terms, and two semantic implication steps.

Finally, for $k > 1$, we must cancel out any $x_{q,u}$ which has insufficiently many paths of length exactly two between it and t. For each such item in turn, we use a simple counting and injectivity argument over the set of potential target vertices for each r to generate a binary clause $\overline{x}_{p,t} + \overline{x}_{q,u} \geq 1$. These are all then added to the original constraint. This requires a further $O(|\mathrm{V}(T)|)$ expressions of size $O(|\mathrm{V}(P)| + |\mathrm{V}(T)|)$, and $O(|\mathrm{V}(T)|)$ simplifications.

We suspect it is also possible to use PB reasoning to justify arbitrary-length distance filtering in polynomial length. However, the short exact path count filtering used in the Glasgow solver appears to be both more efficient and more powerful in practice, and no solver since SND has used arbitrary distance filtering.

3.5 Other Algorithmic Features

There are four other core algorithmic features of the Glasgow solver. The first is all-different propagation, which is used as a powerful way of reasoning about injectivity. The Glasgow solver uses a bit-parallel all-different propagator, rather than the usual generalised arc consistency propagator—however, it still performs deletions and backtracking based upon Hall sets, and so the approach described by Elffers *et al.* [2020] for justifying the all-different constraint may be used with no additional work required.

The second feature is restarts with nogood recording. Archibald *et al.* [2019] showed that rather than performing a simple backtracking search, it is better to repeatedly perform a small amount of search and then restart the solver with a new branching strategy. At every restart, a set of nogoods [Lecoutre *et al.*, 2007; Lee *et al.*, 2016] is recorded, so that the solver does not duplicate work it has already carried out. These nogoods are expressed as CNF clauses and are propagated internally using unit propagation, which means they can simply be logged as-is in the proof file.

The third is that if the input graph is a clique, the solver switches to an entirely different algorithm to solve the problem. Implementing proof logging for this second algorithm is a work in progress. The fourth is parallel search. Our experiments will show that proof logging is heavily I/O bound, and parallelism would make this worse.

Other constraint programming inspired subgraph isomorphism solvers make use of features like arc consistency, and all different filtering on edges. Although not discussed here, these features are also justifiable in polynomial length.

4 Implementation and Evaluation

We implemented[3] the proof logging techniques described above in the Glasgow Subgraph Solver; we also used VeriPB's support for deletion of intermediate and temporary constraints, which cut down on verifier memory usage. Critically, we were able to do all this *without making any changes to the core functions or data structures of the solver*, beyond adding in extra optional calls to the proof logging routines: all of the information needed was already either present or easily accessible from within the solver. (This would not have been the case if we could not use RUP constraints.)

Evaluation. There are currently no other proof logging subgraph isomorphism solvers, so we cannot compare our technique to another solver. However, we can demonstrate that the techniques we have described can be implemented, and that producing and verifying subgraph isomorphism proofs works in practice, at least on smaller graphs.

Hardware setup. Our experiments are performed on a cluster of machines with dual Intel Xeon E5-2697A v4 CPUs and 512GBytes RAM, running Ubuntu 18.04. The performance measurements for writing the proof logs are largely governed by hard disk speed, not CPU overheads, and our machines are all equipped with a single conventional hard disk which limits write speeds to around 100MBytes/s. We therefore do not expect our logging times to be reproducible.

[3]https://github.com/ciaranm/glasgow-subgraph-solver

Instances. We use the instances collected by Kotthoff *et al.* [2016] for evaluation. This is a mix of real-world and randomly generated instances, of a varying range of difficulties. Some of the instances are very large, and so even generating OPB files for them would be infeasible. We therefore select every instance where the target graph has no more than 260 vertices, and where the unmodified Glasgow solver without proof logging can enumerate every solution in no more than ten seconds. (We focus on the enumeration problem because it is more of a stress test than proof logging for decision instances would be.) This gives us a total of 1,227 instances, 789 of which are unsatisfiable, with the remainder having somewhere between one and 50,635,140 solutions; 498 of the instances were solved without any guessing, whilst the hardest solved satisfiable and unsatisfiable instances required 53,605,482 and 2,074,386 recursive calls respectively.

Successful results. Our main result is that the technique works. For all but five of these 1,227 instances, we were able to produce proofs and verify their correctness. For the remaining five instances, the verifier took over three days to run (without yet having found any mistakes). Each of these instances were small satisfiable instances with very many (between fourteen and fifty million) solutions, requiring more than twenty million recursive calls to solve, and with proof log files of between twenty and fifty GBytes.

Time costs of proof logging and verification. In the top row of Figure 1 we show the time costs of performing proof logging and verification on these instances. The first plot shows the cumulative number of instances solved over time without proof logging, with proof logging enabled, and for proof verification. The plot suggests a four orders of magnitude slowdown in aggregate for easy instances, dropping to two orders of magnitude for harder instances. Meanwhile, verifying proofs is approximately one order of magnitude slower than producing them. The second plot shows how much slower producing proofs is on an instance by instance basis—we discuss this further below. The third plot shows how many times harder it is to verify a proof than it is to produce it, and shows a close linear correlation.

OPB and proof log sizes. The second row of Figure 1 looks at the size of the generated OPB and proof log files. The largest OPB file (bearing in mind our pre-selection of small instances) is 425MBytes, for a pattern graph with 121 vertices and a target graph with 128 vertices. Meanwhile, some of our proof logs reached many tens of GBytes—although this sounds large, recall that the subgraph solver can carry out over fifty million backtracks within ten seconds.

Where the costs come from. Although not ideal, the slowdowns to the solver from proof logging are to be expected for two reasons. Firstly, the Glasgow solver employs bit parallelism and other algorithmic techniques and data structures designed to allow it to carry out inference extremely quickly. When working with relatively small target graphs, it is able to carry out a full round of inference, variable selection, and recursion in under 0.2 microseconds. If each such round requires 1KByte of logging, we would need to be able to write to disk at around 5GBytes per second to keep up—this is already a factor of fifty higher than what our hardware is capa-

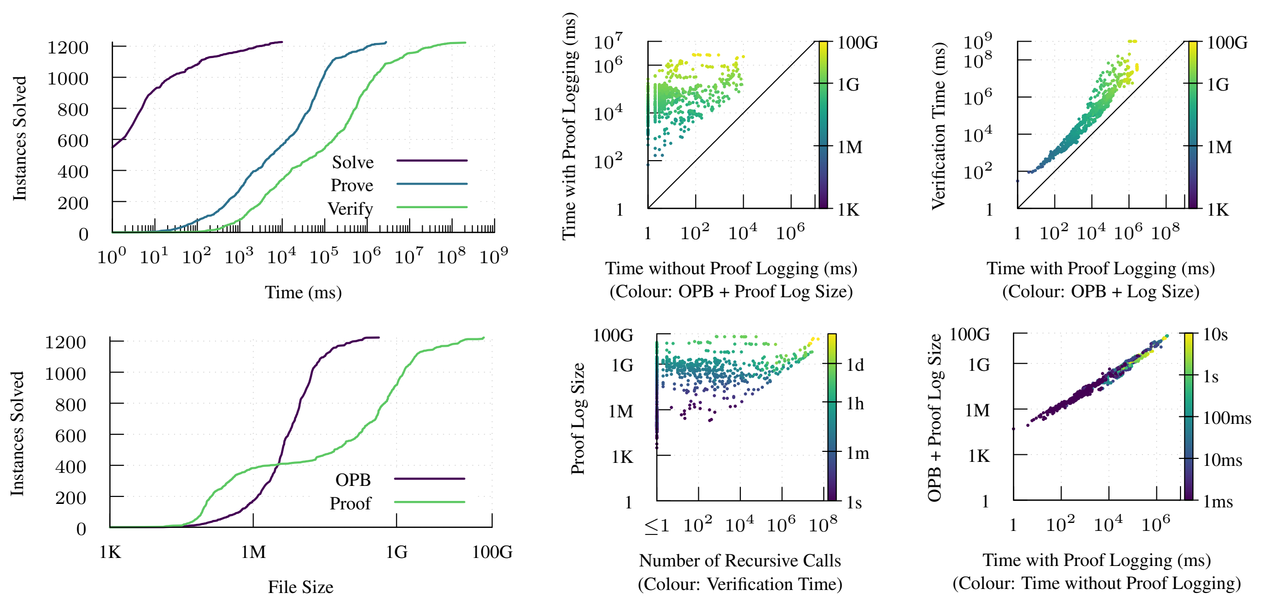

Figure 1: The performance of proof logging and verification on the 1,227 benchmark instances. The top left plot shows the cumulative number of instances solved, solved with proof logging, and verified, for increasing time limits. The bottom left plot shows the cumulative number of instances for which the OPB and proof log files are no more than a given size. The top centre scatter plot shows the increase in time required to enable proof logging, whilst the top right scatter plot shows the verification time, in comparison to the time needed to generate proof files; in both cases, lighter point colours indicate larger disk space requirements. The bottom centre scatter plot shows the size of the proof log file, compared to the number of recursive calls made by the solver (and lighter point colours indicate longer verification times). Finally, the bottom right scatter plot shows output sizes as proof logging times increase.

ble of. The bottom right plot of Figure 1 confirms that I/O is our main problem: performance is very closely correlated to the amount of data that is written out.

Secondly, producing the additional constraints for the additional graph pairs can be expensive: although it is a polynomial operation, moving from the solver's $O(|\operatorname{V}(P)^2| + |\operatorname{V}(T)|^2)$ size requirements to the potential $O(|\operatorname{V}(P)^2||\operatorname{V}(T)|^2)$ size needed for a PB model can be prohibitive. The additional graph pairs make this worse: they can be either sparser or denser than the inputs, and there are instances where the OPB file is relatively small, but where the additional graph pair constraints are close to the worst possible size. We can see this in the middle column of Figure 1: there are instances where no search is performed, where producing the additional graph pairs takes many hundreds of seconds and several GBytes of proof log space.

5 Conclusion

We have shown, for the first time, that it is possible to carry out proof logging and verification for a sophisticated graph algorithm—and that we can do so without the proof verifier needing to be aware of any graph theory. Although there were limits on input we could consider, this method gave us a practical way of verifying the solutions to over a thousand instances. This is especially helpful because some of these instances were too hard for any other solver, meaning we were not previously completely confident that the Glasgow solver was obtaining correct results through legitimate means.

We hope that solvers for other NP-complete problems will start adopting this technology, particularly since increasingly sophisticated reasoning is now being implemented and used in practice. Although the overheads mean it may not be as practical to use proof logging for all instances as it is in the SAT community, we would still prefer to see solvers which could output proofs at least some of the time. For this reason, we consider it particularly relevant that introducing proof logging into the Glasgow subgraph solver was straightforward and non-intrusive. The combination of RUP and simple justifications for counting arguments meant that our main implementation difficulties came from remembering to handle all the special cases like loops and directed and labelled edges, rather than from proof logging itself. We would be especially interested to see whether cutting planes proofs are similarly effective in other domains.

Acknowledgements

Stephan Gocht and Jakob Nordström were supported by the Swedish Research Council grant 2016-00782, and Jakob Nordström also received funding from the Independent Research Fund Denmark grant 9040-00389B. Ciaran McCreesh was supported by the Engineering and Physical Sciences Research Council [grant number EP/P026842/1]. Some experiments used resources provided by the Swedish National Infrastructure for Computing (SNIC) at the High Performance Computing Center North (HPC2N) at Umeå University.

References

[Archibald *et al.*, 2019] Blair Archibald, Fraser Dunlop, Ruth Hoffmann, Ciaran McCreesh, Patrick Prosser, and James Trimble. Sequential and parallel solution-biased search for subgraph algorithms. In Louis-Martin Rousseau and Kostas Stergiou, editors, *Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 16th International Conference, CPAIOR 2019, Thessaloniki, Greece, June 4-7, 2019, Proceedings*, volume 11494 of *Lecture Notes in Computer Science*, pages 20–38. Springer, 2019.

[Audemard *et al.*, 2014] Gilles Audemard, Christophe Lecoutre, Mouny Samy Modeliar, Gilles Goncalves, and Daniel Cosmin Porumbel. Scoring-based neighborhood dominance for the subgraph isomorphism problem. In Barry O'Sullivan, editor, *Principles and Practice of Constraint Programming - 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings*, volume 8656 of *Lecture Notes in Computer Science*, pages 125–141. Springer, 2014.

[Bonnici *et al.*, 2013] Vincenzo Bonnici, Rosalba Giugno, Alfredo Pulvirenti, Dennis E. Shasha, and Alfredo Ferro. A subgraph isomorphism algorithm and its application to biochemical data. *BMC Bioinformatics*, 14(S-7):S13, 2013.

[Cook *et al.*, 1987] William J. Cook, Collette R. Coullard, and György Turán. On the complexity of cutting-plane proofs. *Discrete Applied Mathematics*, 18(1):25–38, 1987.

[Cordella *et al.*, 2004] Luigi P. Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Trans. Pattern Anal. Mach. Intell.*, 26(10):1367–1372, 2004.

[Elffers *et al.*, 2020] Jan Elffers, Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Justifying all differences using pseudo-boolean reasoning. In *Proceedings of AAAI*, 2020.

[Goldberg and Novikov, 2003] Evguenii I. Goldberg and Yakov Novikov. Verification of proofs of unsatisfiability for CNF formulas. In *2003 Design, Automation and Test in Europe Conference and Exposition (DATE 2003), 3-7 March 2003, Munich, Germany*, pages 10886–10891. IEEE Computer Society, 2003.

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

[Heule *et al.*, 2013a] Marijn Heule, Warren A. Hunt Jr., and Nathan Wetzler. Trimming while checking clausal proofs. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 181–188. IEEE, 2013.

[Heule *et al.*, 2013b] Marijn Heule, Warren A. Hunt Jr., and Nathan Wetzler. Verifying refutations with extended resolution. In Maria Paola Bonacina, editor, *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*, pages 345–359. Springer, 2013.

[Kotthoff *et al.*, 2016] Lars Kotthoff, Ciaran McCreesh, and Christine Solnon. Portfolios of subgraph isomorphism algorithms. In Paola Festa, Meinolf Sellmann, and Joaquin Vanschoren, editors, *Learning and Intelligent Optimization - 10th International Conference, LION 10, Ischia, Italy, May 29 - June 1, 2016, Revised Selected Papers*, volume 10079 of *Lecture Notes in Computer Science*, pages 107–122. Springer, 2016.

[Lecoutre *et al.*, 2007] Christophe Lecoutre, Lakhdar Sais, Sébastien Tabary, and Vincent Vidal. Nogood recording from restarts. In *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, pages 131–136, 2007.

[Lee *et al.*, 2016] Jimmy H. M. Lee, Christian Schulte, and Zichen Zhu. Increasing nogoods in restart-based search. In Dale Schuurmans and Michael P. Wellman, editors, *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA*, pages 3426–3433. AAAI Press, 2016.

[McCreesh and Prosser, 2015] Ciaran McCreesh and Patrick Prosser. A parallel, backjumping subgraph isomorphism algorithm using supplemental graphs. In Gilles Pesant, editor, *Principles and Practice of Constraint Programming - 21st International Conference, CP 2015, Cork, Ireland, August 31 - September 4, 2015, Proceedings*, volume 9255 of *Lecture Notes in Computer Science*, pages 295–312. Springer, 2015.

[McCreesh *et al.*, 2018] Ciaran McCreesh, Patrick Prosser, Christine Solnon, and James Trimble. When subgraph isomorphism is really hard, and why this matters for graph databases. *J. Artif. Intell. Res.*, 61:723–759, 2018.

[Solnon, 2010] Christine Solnon. Alldifferent-based filtering for subgraph isomorphism. *Artif. Intell.*, 174(12-13):850–864, 2010.

[Solnon, 2019] Christine Solnon. Experimental evaluation of subgraph isomorphism solvers. In Donatello Conte, Jean-Yves Ramel, and Pasquale Foggia, editors, *Graph-Based Representations in Pattern Recognition - 12th IAPR-TC-15 International Workshop, GbRPR 2019, Tours, France, June 19-21, 2019, Proceedings*, volume 11510 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2019.

[Wetzler *et al.*, 2014] Nathan Wetzler, Marijn Heule, and Warren A. Hunt Jr. Drat-trim: Efficient checking and trimming using expressive clausal proofs. In Carsten Sinz and Uwe Egly, editors, *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*, pages 422–429. Springer, 2014.

[Zampelli *et al.*, 2010] Stéphane Zampelli, Yves Deville, and Christine Solnon. Solving subgraph isomorphism problems with constraint programming. *Constraints*, 15(3):327–353, 2010.