End-to-End Verification for Subgraph Solving

Stephan Gocht[1,2], Ciaran McCreesh[3], Magnus O. Myreen[4], Jakob Nordström[2,1], Andy Oertel[1,2], Yong Kiam Tan[5]
[1]Lund University, Lund, Sweden
[2]University of Copenhagen, Copenhagen, Denmark
[3]University of Glasgow, Glasgow, Scotland
[4]Chalmers University of Technology, Gothenburg, Sweden
[5]Institute for Infocomm Research (I^{2}R), A*STAR, Singapore
gochtstephan@gmail.com, Ciaran.McCreesh@glasgow.ac.uk, myreen@chalmers.se, jn@di.ku.dk, andy.oertel@cs.lth.se, tanyk1@i2r.a-star.edu.sg

Abstract

Modern subgraph-finding algorithm implementations consist of thousands of lines of highly optimized code, and this complexity raises questions about their trustworthiness. Recently, some state-of-the-art subgraph solvers have been enhanced to output machine-verifiable proofs that their results are correct. While this significantly improves reliability, it is not a fully satisfactory solution, since end-users have to trust both the proof checking algorithms and the translation of the high-level graph problem into a low-level 0–1 integer linear program (ILP) used for the proofs.

In this work, we present the first *formally verified* toolchain capable of full end-to-end verification for subgraph solving, which closes both of these trust gaps. We have built encoder frontends for various graph problems together with a 0–1 ILP (a.k.a. pseudo-Boolean) proof checker, all implemented and formally verified in the CAKEML ecosystem. This toolchain is flexible and extensible, and we use it to build verified proof checkers for both decision and optimization graph problems, namely, *subgraph isomorphism*, *maximum clique*, and *maximum common (connected) induced subgraph*. Our experimental evaluation shows that end-to-end formal verification is now feasible for a wide range of hard graph problems.

Introduction

Combinatorial optimization algorithms have improved immensely since the turn of the millennium, and are now routinely used to solve large-scale real-world problems, through both general-purpose solving paradigms (Biere et al. 2021; Bixby and Rothberg 2007; Garcia de la Banda et al. 2014) and dedicated algorithms for more specialised problems such as subgraph finding (McCreesh, Prosser, and Trimble 2020). Since these combinatorial solvers are used for an increasingly wide range of applications, it becomes crucial that the results they compute can be trusted. Sadly, this is currently not the case (Cook et al. 2013; Akgün et al. 2018; Gillard, Schaus, and Deville 2019; Bogaerts, McCreesh, and Nordström 2022). Extensive testing, though beneficial, has not been able to resolve the problem of solvers occasionally producing faulty answers, and attempts to build correct-by-construction software using formal verification run into the obstacle that current techniques cannot scale to the level of complexity of modern solvers.

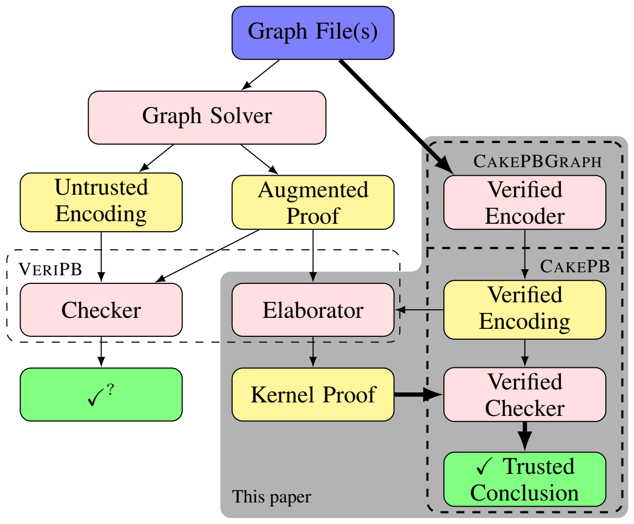


Figure 1: The full verification workflow. Without verified proof checking, only the left-hand part of the diagram is used. Our current work enables the additional shaded parts, where the thick dashed box is the formally verified program and thick arrows show its key input-output interfaces.

Instead, the most promising way to achieve verifiably correct combinatorial solving seems to be *proof logging*, meaning that solvers produce efficiently verifiable certificates of correctness that can be corroborated by an independent proof checking program (McConnell et al. 2011). This approach has been successfully used in the SAT community (Heule, Hunt Jr., and Wetzler 2013a,b; Wetzler, Heule, and Hunt Jr. 2014), which raises the question of whether similar techniques could be employed in other settings such as subgraph finding. For this it would seem that the proof checker would need to understand graph concepts such as vertices, edges, neighbourhoods, et cetera. Surprisingly, this turns out not to be the case—instead, the solver can encode the graph problem using 0–1 linear inequalities (also referred to as *pseudo-Boolean constraints*), and then justify its complex high-level reasoning in terms of this low-level representation. This approach has been used to add proof logging with the VERIPB tool to state-of-the-art solvers for subgraph isomorphism, clique, and maximum common (connected) induced subgraph (Gocht, McCreesh, and Nordström 2020; Gocht et al. 2020), as illustrated in the left-hand part of Figure 1. We emphasize that although this approach uses reasoning with pseudo-Boolean constraints for the proof logging, it is *not* limited to pseudo-Boolean solving. Rather, it can be used to certify the output of *any* untrusted solver—such as tools that operate natively on graph representations—as long as the solver's relevant reasoning steps can be expressed with pseudo-Boolean proofs.

While this approach has been successful for debugging solvers and providing convincing demonstrations that the fixed solvers are producing correct answers, it is important to observe that it crucially hinges on the assumption that three components are correct: (1) the low-level encoding of the problem, (2) the proof checker, and (3) the interpretation of the final output. For example, if the maximum clique solver in Gocht et al. (2020) produces a proof accepted by the VERIPB checker, then one can conclude that *if* the 0–1 ILP encoding of clique is implemented correctly, and *if* VERIPB does not contain bugs, and *if* (say) a 200-vertex graph having a maximum clique size of 13 corresponds to the optimal objective value for the low-level encoding being 187 (because it minimises the number of vertices not in the clique), then the maximum clique size is indeed 13. Such assumptions are not unreasonable—encodings have been chosen to be as simple as possible and the code can be subjected to extensive testing; the proof format is designed so that proof checking should be easy; and verifying that proof outputs correspond to solver outputs is not too cumbersome. Compared to having to trust an extremely complex solver, this is a vast improvement. However, if provably correct results are the end goal, then this still leaves much to be desired.

Our Contribution

In this work, we resolve all the concerns discussed above by presenting the first toolchain capable of end-to-end formal verification for state-of-the-art algorithms for maximum clique, subgraph isomorphism, and maximum common (connected) induced subgraph problems. Although the implementations of modern solvers for these problems are far too complicated to be formally verified by current techniques, we can still use formal verification to certify the correctness of the proof logging and proof checking process. We do so by defining a solver-friendly *augmented* VERIPB proof format; enhancing the VERIPB tool with a *proof elaborator* that can translate such augmented proofs to a more explicit *kernel* format; and designing a *formally verified proof checker* for the kernel format. This formally verified checker is also capable of providing its own formally verified encodings from graph problems to 0–1 ILPs. Finally, the output provided by the formally verified proof checker is in terms of the original problem, not the low-level encoding. This means that using the process illustrated in the right-hand part of Figure 1, if the checking process outputs (say)

```
s VERIFIED MAX CLIQUE SIZE |CLIQUE| = 13
```

$$\begin{aligned}
&\text{is\_clique}\ vs\ (v,e) \overset{\text{def}}{=} \\
&\quad vs \subseteq \{\ 0,1,...,v{-}1\ \} \land \\
&\quad \forall x\ y.\ x \in vs \land y \in vs \land x \neq y \Rightarrow \text{is\_edge}\ e\ x\ y \\
&\text{max\_clique\_size}\ g \overset{\text{def}}{=} \text{max}_{\text{set}}\ \{\ \text{card}\ vs \mid \text{is\_clique}\ vs\ g\ \}
\end{aligned}$$

$$\begin{aligned}
&\text{has\_subgraph\_iso}\ (v_p,e_p)\ (v_t,e_t) \overset{\text{def}}{=} \\
&\quad \exists f.\ \text{inj}\ f\ \{\ 0,1,...,v_p{-}1\ \}\ \{\ 0,1,...,v_t{-}1\ \} \land \\
&\qquad \forall a\ b.\ \text{is\_edge}\ e_p\ a\ b \Rightarrow \text{is\_edge}\ e_t\ (f\ a)\ (f\ b)
\end{aligned}$$

Figure 2: HOL definitions for maximum clique size of a graph with v vertices and edge set e (top), and existence of a subgraph isomorphism from a pattern graph (v_p, e_p) to a target graph (v_t, e_t) (bottom).

then we can be absolutely sure that the maximum clique size for our graph is 13, *if* we trust the formal verification tool(s) and *if* the formal higher-order logic (HOL) specifications (as shown in Figure 2) accurately reflect what it means to be a clique. The toolchain we provide is also flexible and extensible, in that it can be readily adapted to other combinatorial problems, including problems not involving graphs.

Comparison to Related Work

Formally verified proof checkers have previously played an important role in SAT solving (Cruz-Filipe, Marques-Silva, and Schneider-Kamp 2017; Cruz-Filipe et al. 2017; Lammich 2020) and are vital for widespread acceptance of SAT-solver-generated mathematical proofs (Heule and Kullmann 2017). However, such proof checkers have worked only for conjunctive normal form (CNF), and only to establish that decision problems encoded in CNF are infeasible: verification that the encoding accurately reflects the problem to be solved has either been ignored or has been handled separately (e.g, Cruz-Filipe, Marques-Silva, and Schneider-Kamp 2019; Shi et al. 2021; Codel, Avigad, and Heule 2023). For graph problems, previous attempts at verified proof checking have been tied to one specific problem, or even one specific algorithm (e.g., Bankovic, Drecun, and Maric 2023). In contrast, we provide formal verification for optimization problems and with much more expressive formats than CNF, and we do so in a unified way with a single pseudo-Boolean proof logging format for 0–1 linear inequalities together with a general-purpose toolchain, rather than having to design proof logging from scratch for each new combinatorial problem considered. In this way, we demonstrate that end-to-end formally verified combinatorial solving is now eminently within reach, by combining pseudo-Boolean proof logging with formally verified tools for 0–1 ILP encodings and pseudo-Boolean proof checking.

Outline of This Paper

After reviewing preliminaries, we describe the formally verified proof checker, and how solver proofs in a user-friendly proof format can be converted to a more restricted format accepted by this proof checker. We then report results from an experimental evaluation, and conclude with a discussion of future research directions.

Preliminaries

Our discussion of pseudo-Boolean proof logging will be brief, since the main thrust of this work is how to formally verify proof logging rather than to design it. See Gocht and Nordström (2021) and Bogaerts et al. (2023a) for more on the VERIPB system and Buss and Nordström (2021) for background on the cutting planes reasoning method used.

A *literal* ℓ over a variable x is x itself or its negation $\overline{x}$, taking values 0 (false) or 1 (true), so that $\overline{x} = 1 - x$. A *pseudo-Boolean (PB) constraint* C is a 0-1 integer linear inequality $\sum_i a_i \ell_i \geq A$, which without loss of generality we can always assume to be in *normalized form*; i.e., all literals ℓ_i are over distinct variables and the coefficients a_i and the *degree (of falsity)* A are non-negative. The *negation* $\neg C$ of C is $\sum_i a_i \overline{\ell}_i \geq \sum_i a_i - A + 1$ (saying that the sum of the coefficients of falsified literals is so large that the satisfied literals can contribute at most $A-1$). A *pseudo-Boolean formula* is a conjunction $F = \bigwedge_j C_j$ of PB constraints.

Cutting planes (Cook, Coullard, and Turán 1987) is a method for iteratively deriving new constraints logically implied by a PB formula by taking positive linear combinations or dividing a constraint and rounding up. We say that *C unit propagates* the literal ℓ if under the current partial assignment C cannot be satisfied unless ℓ is set to true, and that C is implied by F by *reverse unit propagation (RUP)* if adding $\neg C$ to F and then unit propagating until saturation leads to contradiction in the form of a violated constraint. VERIPB allows adding constraints by RUP, which is a convenient way of avoiding having to write out explicit syntactic derivations.

In addition to deriving constraints C that are implied by F, VERIPB also has *strengthening* rules for inferring *redundant* constraints D having the property that F and $F \wedge D$ are equisatisfiable. If there is a partial mapping ω of variables to literals and/or truth values such that

$$F \cup \{\neg D\} \vdash (F \cup D)\restriction_\omega \tag{1}$$

holds, meaning that after applying ω to $F \cup \{D\}$ all of the resulting constraints can be derived by cutting planes from $F \cup \{\neg D\}$, then D can be added by *redundance-based strengthening*. There is also a similar but slightly different *dominance-based strengthening* rule. Importantly, the proof has to specify ω and also contain explicit subderivations for all *proof goals* in $(F \cup D)\restriction_\omega$ in eq. (1) unless they are obvious enough that VERIPB can automatically figure them out (e.g., by using RUP). Finally, for optimization problems there are rules to deal with objective functions and incumbent solutions, and the strengthening rules also need to be slightly adapted for this setting.

The formalization of our proof checking toolchain is carried out in the HOL4 proof assistant for classical higher-order logic (Slind and Norrish 2008). We make particular use of the CAKEML tools for production and optimization of verified CAKEML source code (Myreen and Owens 2014; Guéneau et al. 2017) as well as for formally verified compilation (Tan et al. 2019), allowing to transfer guarantees of source-code-level correctness down to executable machine code. Where applicable, formal code snippets are pretty-printed for illustration, e.g., as shown in Figure 2. The set and first-logic notation is standard (e.g., $\Rightarrow$ denotes logical implication); other HOL notation is explained where appropriate. Formally verified results are preceded by a turnstile $\Vdash$. All code is available in the supplementary material (Gocht et al. 2023).

Formally Verified Graph Proof Checkers

This section details the formal verification of our pseudo-Boolean proof checker CAKEPB and its various graph frontends, focusing on the key architectural decisions and reusable insights behind the verification effort. An overview of the tool is shown in Figure 3. We first present the different components, and then plug them together to obtain end-to-end verified graph proof checkers.

Verified Pseudo-Boolean Proof Checking

A key design objective for CAKEPB is to make it a general yet effective pseudo-Boolean proof checking backend. To this end, CAKEPB supports a *kernel* subset of the VERIPB proof format with cutting planes, strengthening, and optimization rules as discussed in the previous section. The implementation and verification of all of this within a single proof checker backend presents several new challenges compared to prior tools for efficient verified CNF proof checking (Cruz-Filipe et al. 2017; Lammich 2020; Tan, Heule, and Myreen 2023). Firstly, the pseudo-Boolean proof system features a much richer set of rules, each of which needs a formal soundness justification. Secondly, there is an intricate interplay between different proof rules, especially concerning how they preserve optimal solutions (or satisfiability for decision problems). This necessitates careful maintenance of state invariants within the proof checker implementation. And thirdly, all of the above needs to be adequately optimized for practical use, whilst being formally verified.

We use a refinement-based approach to tackle each challenge in order and at the appropriate level of abstraction.

1. The verification process starts by defining an abstract, mathematical, pseudo-Boolean semantics, with respect to which the soundness of each rule is justified. For example, we prove lemmas that justify the soundness of adding two constraints and dividing a constraint by a non-zero natural number in a cutting planes proof step:

   $\Vdash$ $\mathsf{satisfies\_npbc}\ w\ C_1 \wedge \mathsf{satisfies\_npbc}\ w\ C_2 \Rightarrow$
   $\quad \mathsf{satisfies\_npbc}\ w\ (\mathsf{add}\ C_1\ C_2)$

   $\Vdash$ $\mathsf{satisfies\_npbc}\ w\ C \wedge k \neq 0 \Rightarrow$
   $\quad \mathsf{satisfies\_npbc}\ w\ (\mathsf{divide}\ C\ k)$

   Here, $\mathsf{satisfies\_npbc}\ w\ C$ says that the pseudo-Boolean constraint C is satisfied by the Boolean assignment w. We verify similar lemmas for all supported reasoning principles, the most involved of which is dominance-based strengthening. Specifically, this rule requires making a well-founded induction argument over an arbitrary user-specified order for Boolean assignments, for which we largely follow the proof from Bogaerts et al. (2023a, Proposition 4).
2. Next, we implement a prototype proof checker that ensures that every application of a proof rule is valid, e.g.,

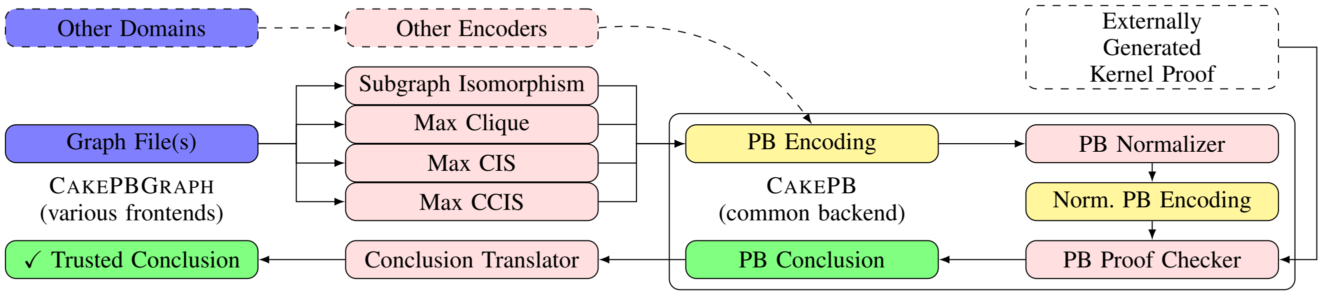


Figure 3: Architecture of the end-to-end verified proof checkers for various graph problems.

that divide is never applied with $k = 0$, throwing an error otherwise. The proof checker is verified to maintain key invariants on the proof state, especially the ones needed for dominance and optimization reasoning. Soundness of the checker is proved by induction over the sequence of proof steps. The main idea is illustrated by the following abridged lemma snippet.

$$\begin{aligned}&\Vdash \ldots \wedge \mathsf{valid\_conf}\ \mathit{ord}\ \mathit{obj}\ \mathit{fml} \Rightarrow\\ &\quad \mathsf{check\_step}\ \mathit{step}\ \mathit{ord}\ \mathit{obj}\ \mathit{fml}\ \ldots =\\ &\qquad \mathsf{Some}\ (\mathit{ord}',\mathit{obj}',\mathit{fml}',\ldots) \Rightarrow\\ &\quad \ldots \wedge \mathsf{valid\_conf}\ \mathit{ord}'\ \mathit{obj}'\ \mathit{fml}'\end{aligned}$$

Here, valid_conf *ord obj fml* says that for any satisfying assignment w to the core constraints in formula *fml*, there exists another satisfying assignment $w' \preccurlyeq w$ which satisfies all constraints in *fml*, where $\preccurlyeq$ is the order on assignments induced by *ord* and *obj*. The lemma fragment says that, whenever checking a single proof step (check_step) succeeds and returns a new proof checker state (result Some), the valid_conf invariant is maintained for the state. Other key properties verified for check_step include showing that *fml′* and *fml* are equisatisfiable by assignments that improve the best known objective value.

3. The final phase involves refining the prototype into an optimized proof checker implementation using the CAKEML tools for profiling and source code verification (Myreen and Owens 2014; Guéneau et al. 2017). We manually optimize several hotspots encountered in the pseudo-Boolean proofs generated in our experimental evaluation, e.g., using buffered I/O to stream large proof files, and swapping to constant-time array-based constraint lookups for cutting planes steps and hash-based proof goal coverage checks in application of the dominance-based strengthening rule.

The verified proof checker backend operates most naturally and efficiently with normalized pseudo-Boolean constraints where, in addition, variables are indexed by numbers. However, this is not the most convenient interface for frontend users. Accordingly, CAKEPB also includes a verified pseudo-Boolean *normalizer*. As shown in Figure 3, CAKEPB accepts any pseudo-Boolean formula as input (normalized or otherwise) together with an externally generated kernel proof. It produces an appropriate verified conclusion about the formula, such as satisfiability status or upper and lower bounds on the objective function, depending on the type of problem and on the claims made by the proof.

$$\begin{aligned}&\mathsf{is\_cis}\ \mathit{vs}\ (v_p,e_p)\ (v_t,e_t) \overset{\mathsf{def}}{=}\\ &\quad \exists f.\ \mathit{vs} \subseteq \{\ 0,1,\ldots,v_p{-}1\ \} \wedge \mathsf{inj}\ f\ \mathit{vs}\ \{\ 0,1,\ldots,v_t{-}1\ \} \wedge\\ &\qquad \forall a\ b.\ a \in \mathit{vs} \wedge b \in \mathit{vs} \Rightarrow\\ &\qquad\quad (\mathsf{is\_edge}\ e_p\ a\ b \iff \mathsf{is\_edge}\ e_t\ (f\ a)\ (f\ b))\\ &\mathsf{connected\_subgraph}\ \mathit{vs}\ e \overset{\mathsf{def}}{=}\\ &\quad \forall a\ b.\ a \in \mathit{vs} \wedge b \in \mathit{vs} \Rightarrow\\ &\qquad (\lambda x\ y.\ y \in \mathit{vs} \wedge \mathsf{is\_edge}\ e\ x\ y)^*\ a\ b\\ &\mathsf{is\_ccis}\ \mathit{vs}\ (v_p,e_p)\ (v_t,e_t) \overset{\mathsf{def}}{=}\\ &\quad \mathsf{is\_cis}\ \mathit{vs}\ (v_p,e_p)\ (v_t,e_t) \wedge \mathsf{connected\_subgraph}\ \mathit{vs}\ e_p\\ &\mathsf{max\_ccis\_size}\ g_p\ g_t \overset{\mathsf{def}}{=}\\ &\quad \mathsf{max}_{\mathsf{set}}\ \{\ \mathsf{card}\ \mathit{vs} \mid \mathsf{is\_ccis}\ \mathit{vs}\ g_p\ g_t\ \}\end{aligned}$$

$$\begin{aligned}&\Vdash \mathsf{good\_graph}\ (v_p,e_p) \wedge \mathsf{good\_graph}\ (v_t,e_t) \wedge\\ &\quad \mathsf{encode}\ (v_p,e_p)\ (v_t,e_t) = \mathit{constraints} \Rightarrow\\ &\quad\ ((\exists \mathit{vs}.\ \mathsf{is\_ccis}\ \mathit{vs}\ (v_p,e_p)\ (v_t,e_t) \wedge \mathsf{card}\ \mathit{vs} = k) \iff\\ &\qquad \exists w.\ \mathsf{satisfies}\ w\ (\mathsf{set}\ \mathit{constraints}) \wedge\\ &\qquad\quad \mathsf{eval\_obj}\ (\mathsf{unmapped\_obj}\ v_p)\ w = v_p - k)\end{aligned}$$

Figure 4: HOL definition of the size of a maximum common connected induced subgraph (MCCIS) for a pattern graph g_p and a target graph g_t (top), and a correctness theorem for encoding the MCCIS problem using PB constraints (bottom).

Verified Graph Problem Encoders

Pseudo-Boolean formulas provide a convenient format for verified frontend encoders for graph problems, which we turn to next. Graphs are represented in HOL as a pair (v,e), where v is the number of vertices corresponding to the vertex set $\{\ 0,1,\ldots,v{-}1\ \}$, and e is an edge list representation such that is_edge $e\ a\ b$ is true iff there is an edge between vertices a and b. All graphs considered here are undirected.[1] The graph encoders use a shared graph library which formalizes these basic graph notions and provides parsing functions for standard text formats such as LAD and DIMACS.

The HOL definitions of various graph problems formalized in this paper are shown in Figures 2 and 4; we use maximum common connected induced subgraph (MCCIS) as a

[1]In practice, we apply a consistency check good_graph for undirectedness and other syntactic properties when parsing input graphs. Graphs failing the check are rejected by the encoders.

representative example. Given a pattern graph g_p and a target graph g_t, a subset of vertices vs of g_p is a common induced subgraph (is_cis) iff there exists an injective mapping f from vs into the target graph vertices which preserves edges and non-edges. Additionally, vs is a connected subgraph of g_p iff its vertices are pairwise connected in the reflexive transitive closure (denoted *) of the induced is_edge relation. The MCCIS size is the size of the largest common connected induced subgraph between g_p and g_t (max_ccis_size).

The MCCIS pseudo-Boolean encoding from Gocht et al. (2020, Section 3.1) is implemented as a HOL function encode. The main subtlety is connected_subgraph; briefly, connectedness is encoded using additional auxiliary variables that indicate whether a walk of length n for some $n < \min(v_p, v_t)$, exists between each pair of vertices in the chosen subgraph. The correctness theorem for encode is shown in Figure 4 (bottom). It says that a CCIS of cardinality k exists iff a satisfying assignment to the encoding $constraints$ exists with objective value $v_p - k$. Therefore, minimizing the objective (unmapped_obj v_p) yields the MCCIS size. Similar theorems are proved for encodings of subgraph isomorphism and maximum clique. The value of formal verification here is twofold: to gain confidence in the pen-and-paper justification of the encodings, and to ensure that the encodings are correctly implemented in code.

End-to-End Verification

Feeding the output of each frontend encoder into CAKEPB yields a suite of formally verified graph proof checkers, collectively called CAKEPBGRAPH. Since we are working within the CAKEML ecosystem, we can further achieve *end-to-end* verification by running the CAKEML compiler on CAKEPBGRAPH to transfer the source-level correctness guarantees for the CAKEPBGRAPH checkers down to the level of their respective machine code implementations.

Let us illustrate this by briefly discussing the final correctness theorem for the maximum clique proof checker as shown in Figure 5. The assumption on Line 1 is standard for all programs written in CAKEML, and states that the compiled machine code is correctly loaded in memory of an x64 machine and that the appropriate command line and file system foreign function interfaces (FFIs) are available to CakeML. The first correctness guarantee on Lines 2–3 says that the code will run without crashing and will terminate safely, possibly reporting an out-of-memory resource error. The second correctness guarantee starting at Line 4–5 says there will be (possibly empty) strings *out* and *err* printed to standard output and error, respectively. The remaining lines now claim that if standard output is non-empty, then the input file was parsed in DIMACS format to a graph g (Lines 6–7), and the output is either:

- a pretty-printed pseudo-Boolean encoding of the maximum clique problem for g (Line 8), or
- a pretty-printed conclusion string which is either:
  - a verified exact maximum clique size for g formatted using clique_eq_str (Line 10), or
  - verified lower and upper bounds on clique sizes in g formatted using clique_bound_str (Lines 11–12).

Let us clarify what needs to be trusted, or at least carefully inspected, in order to claim that the conclusions by CAKEPBGRAPH checkers are formally verified:

- The HOL definitions of the graph input parsers and of various graph problems that appear in the final correctness theorems (e.g., Figure 5). We have kept these definitions as simple as possible. Notably, the internal definitions of pseudo-Boolean semantics and cutting planes used in the proof checker are *not* part of CAKEPBGRAPH's trusted base because conversion into and out of pseudo-Boolean semantics is formally verified.
- The formal HOL model of the CAKEML execution environment and its correspondence with the real system on which CAKEPBGRAPH runs. CAKEML has been used in various other proof checkers, e.g., by Tan, Heule, and Myreen (2023), and its target architecture models have been validated extensively (Tan et al. 2019).
- The HOL4 theorem prover, including its logic, implementation and execution environment. The prover follows an LCF-style design (Slind and Norrish 2008) with a well-separated and trustworthy kernel responsible for checking every logical inference.

A trusted base for *binary code extraction* (Kumar et al. 2018) as above is of the highest assurance standard for formally verified software—correctness is proved within a single system down to the machine code that runs. This provides a gold standard of trustworthiness for subgraph solving, in contrast to prior unverified proof checking approaches.

Proof Elaboration

CAKEPBGRAPH verification helps solver *users* who wish to attain a high level of trust in solver conclusions. In this section, we discuss our new *elaboration* phase, which aids solver *authors* who wish to add trustworthy proof logging and checking to their tools.

The convenience afforded by proof elaboration is illustrated in the workflow in Figure 1. First, solver authors can design their proof output with respect to their own (untrusted) pseudo-Boolean encodings, without following the verified encodings from CAKEPBGRAPH exactly; elaboration helps to automatically line up (where possible) untrusted and verified encodings. Second, elaboration supports an *augmented* proof format with syntactic sugar that makes proof logging much easier at runtime; elaboration then fills in the necessary details to convert the proof into the kernel format understood by CAKEPBGRAPH. The VERIPB proof elaborator also performs (unverified) proof checking during the translation process, helping solver authors to detect bugs in their proof logging or solver code even before the formal verification process starts.

Lining up Encodings

Many VERIPB proof rules refer to constraints by positive integer *constraint IDs*, assigned automatically in order of appearance in the proof. It would be quite a hassle for solver authors to keep track of the exact order in which constraints in the encoding are generated by CAKEPBGRAPH.

```
clique_eq_str $n \overset{\text{def}}{=}$ "s VERIFIED MAX CLIQUE SIZE |CLIQUE| = " ^ toString $n$ ^ "\n"
clique_bound_str $l\ u \overset{\text{def}}{=}$
 "s VERIFIED MAX CLIQUE SIZE BOUND " ^ toString $l$ ^ " <= |CLIQUE| <= " ^ toString $u$ ^ "\n"

⊩ cake_pb_clique_run $cl\ fs\ mc\ ms \Rightarrow$
   machine_sem $mc$ (basis_ffi $cl\ fs$) $ms \subseteq$
    extend_with_resource_limit $\{$ Terminate Success (cake_pb_clique_io_events $cl\ fs$) $\} \wedge$
   $\exists\, out\ err.$
    extract_fs $fs$ (cake_pb_clique_io_events $cl\ fs$) = Some (add_stdout (add_stderr $fs\ err$) $out$) $\wedge$
    ($out \neq$ "" $\Rightarrow$
      $\exists\, g.$ get_graph_dimacs $fs$ (el 1 $cl$) = Some $g \wedge$
        (length $cl$ = 2 $\wedge$ $out$ = concat (print_pbf (full_encode $g$)) $\vee$
         length $cl$ = 3 $\wedge$
         ($out$ = clique_eq_str (max_clique_size $g$) $\vee$
          $\exists\, l\ u.$
            $out$ = clique_bound_str $l\ u \wedge (\forall vs.$ is_clique $vs\ g \Rightarrow$ card $vs \leq u) \wedge \exists vs.$ is_clique $vs\ g \wedge l \leq$ card $vs$)))
```

Figure 5: End-to-end correctness theorem for CAKEPB with a maximum clique pseudo-Boolean encoder frontend.

Fortunately, it is straightforward to instead recover an ID by rederiving the constraint, which provides it with a new, known ID, before it is used. This can either be done upfront, at the start of the proof, or lazily (which avoids a potentially large overhead for instances with very short proofs). A useful fact is that the two constraints do not need to match exactly—it is sufficient that they are close enough so that VERIPB can automatically check and prove that one of them follows from the other.

When it comes to variable names, the solver proof logging routines are required to agree exactly with the CAKEPB-GRAPH encoding. This is an easier task, however, since VERIPB and CAKEPB both support expressive variable names. For example, for subgraph mapping problems, we use the protocol that the variable name `x1_2` means that pattern vertex 1 will be mapped to target vertex 2.

Elaborating on Syntactic Sugar

The augmented proof format contains a number of rules designed to support the ease of proof logging. Chief among these is *reverse unit propagation (RUP)*, which allows to add a constraint when the VERIPB proof checker can easily verify that it is implied by applying unit propagation. Such RUP steps occur frequently in proofs in many applications, and so have to be dealt with efficiently by the proof checker, but implementing efficient formally verified unit propagation is a challenging task even for the simpler case of CNF (Fleury, Blanchette, and Lammich 2018). Instead, a RUP rule application deriving C from F is converted to an explicit cutting planes proof of contradiction from $F \cup \{\neg C\}$. This is possible since unit propagation on the latter set of constraints leads to a violation (by the definition of RUP), and this in turn means that pseudo-Boolean conflict analysis can be used to derive contradiction. This algorithm is more involved than CNF-based conflict analysis as used in SAT solvers, but we employ a procedure similar to the PB conflict analysis in Elffers and Nordström (2018) for this. For optimization problems, the augmented format allows incumbent solutions to be partially specified, so long as the given assignment unit propagates to a full solution; the kernel format will always specify a full solution instead. This is illustrated in Figure 6.

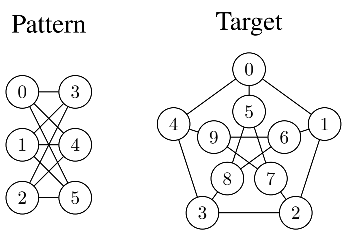
Another convenient rule is *syntactic implication*, where a constraint to be derived is implied by a single (unspecified) previous constraint by simple syntactic manipulations. This condition is again easy to check, but the elaborator converts this into an explicit derivation or explicitly annotates the kernel proof with IDs. Yet another important aspect that we are ignoring here, but which is crucial for efficient proof checking, is deletion of constraints no longer needed in the proof.

Finally, applications of strengthening rules generate a separate proof goal for each constraint currently in use in the proof, which is a potentially huge overhead, but often most of these proof goals are obvious and can be skipped in the augmented format (e.g., if they can be obtained by RUP or syntactic implication). The proof elaborator fills in the necessary missing details for such proof goals.

Experiments

To validate our approach, we performed experiments on a cluster of machines with dual AMD EPYC 7643 processors, 2TBytes RAM, and a RAID array of solid state drives, running Ubuntu 22.04. We ran up to 40 jobs in parallel, and limited each individual process to 64GBytes RAM. Note that performance of the verification process is strongly affected by I/O and memory cache speeds, and so we do not expect running time measurements to be highly reproducible, but they should still be indicative of the feasibility of the approach and the slowdowns that one might encounter. We used the Glasgow Subgraph Solver (McCreesh, Prosser, and Trimble 2020) as the proof-producing solver for all experiments, and made small modifications so that it would lazily recover constraint IDs as required. The results are plotted on an instance by instance basis in Figure 7 and explained below.

For maximum clique, we took the 54 instances from the Second DIMACS Implementation Challenge (Johnson and Trick 1996) that Gocht et al. were able to check. We managed to produce proofs for and formally verify 50 of these



Verified Encoding

```
min: 1 x0_n 1 x1_n 1 x2_n 1 x3_n 1 x4_n 1 x5_n ;
1 x0_n 1 x0_0 1 x0_1 1 x0_2 1 x0_3 1 x0_4 1 x0_5 \
  1 x0_6 1 x0_7 1 x0_8 1 x0_9 = 1 ;
1 x1_n 1 x1_0 1 x1_1 1 x1_2 1 x1_3 1 x1_4 1 x1_5 \
  1 x1_6 1 x1_7 1 x1_8 1 x1_9 = 1 ;
... 1172 omitted constraints ...
```

Augmented Proof

```
pseudo-Boolean proof version 2.0
...
* Specifying a partial solution
soli x5_9 x2_7 ... (58 omitted literals)
...
* Unit propagation step
u 1 ~x4_0 >= 1 ;




...
conclusion BOUNDS 2 2
end pseudo-Boolean proof
```

Kernel Proof

```
pseudo-Boolean proof version 2.0
...
* Specifying a full solution
soli x0_n x1_n ... (304 omitted literals)
...
* Derivation by cutting planes
red 1 ~x4_0 >= 1 ; ; begin
  pol 8784 8778 + 8772 + 8766 + ... \
    + 8133 13 * + 8085 13 * +
end 8786
...
conclusion BOUNDS 2 : 8798 2
end pseudo-Boolean proof
```

Figure 6: (Top) MCCIS problem encoding for the pattern graph $K_{3,3}$ and the target Petersen graph. (Bottom) An augmented proof generated by a solver on the left, and a corresponding elaborated kernel proof on the right; kernel annotations in **bold**. When run on the kernel proof, CAKEPBGRAPH outputs: `s VERIFIED MAX CCIS SIZE |CCIS| = 4`. This corresponds to the conclusion in the proof, which claims that at least two of the six pattern vertices must be mapped to null.

instances; for the 4 instances that we could not verify, 3 were due to VERIPB taking over one week to check the proof files, and the final one to the 64GByte memory limit for the verified checker. Over the successfully checked instances, translating augmented proofs to kernel proofs took, on average, 18% longer than simply verifying the proofs, and produced proof files that were on average 2.26 times as large. However, verified checking of these kernel proofs was consistently faster than checking the original augmented proofs using VERIPB: the average running time was 3.8 times lower.

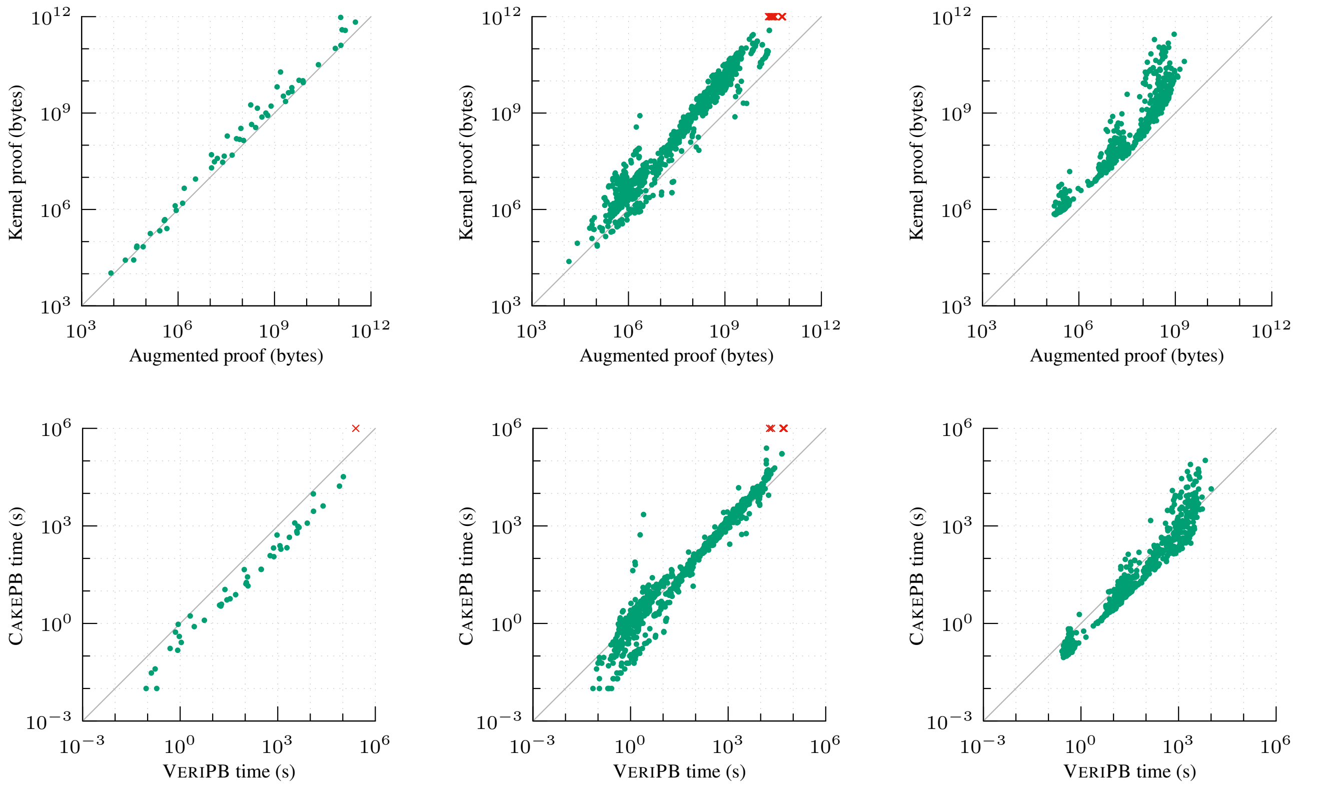
For subgraph isomorphism, we used the same subset of 1,226 small-to-medium-sized instances from the benchmark set in (Kotthoff, McCreesh, and Solnon 2016) as was studied by Gocht, McCreesh, and Nordström (2020). We were able to verify 417 satisfiable and 784 unsatisfiable instances; 13 instances failed due to memory limits on the verified checker, and 12 instances when the converted kernel proofs exceeded 500GBytes in size. Performance-wise, running VERIPB and asking it to output a kernel proof was on average 27% slower than verification alone. Producing the verified encoding was never a significant cost in the process. Verifying kernel proofs was on average 2.4 times slower than verifying the original, augmented proofs; the former were on average 10.5 times larger than the latter.

For maximum common connected induced subgraph, we used a database of randomly generated instances (Conte, Foggia, and Vento 2007; De Santo et al. 2003), and ran the solver in clique reformulation mode. We were able to verify all 690 instances involving up to 20 vertices in each graph. Elaborating the proofs took on average 43% longer than verifying them using VERIPB, and the proofs were on average 14.7 times larger. However, verifying the kernel proofs using CAKEPB took on average only 9% longer than using VERIPB for the original, augmented proofs.

Across each problem family, producing formally verified encodings was always extremely cheap, and asking VERIPB to produce an elaborated kernel proof was never substantially more expensive than simply checking the augmented proof. This is to be expected: VERIPB already has to produce nearly all of the information needed for proof elaboration to check a proof anyway. Checking elaborated proofs was sometimes a little faster than checking the original, augmented proof, and sometimes a little slower, and we were able to formally check almost every proof that was amenable to unverified checking.

Conclusion

In this paper, we present the first efficient toolchain for formal end-to-end verification of state-of-the-art subgraph solving. Our design is easily adaptable, which opens up the possibility of bringing formal verification to other combinatorial problem domains where problem instances can be suitably represented using the expressivity of 0–1 integer linear programs. In fact, our formally verified CAKEPB proof checker equipped with a CNF frontend has also been used for SAT solving in the SAT Competition 2023 (Bogaerts et al. 2023b), supporting, also for the first time, efficient



(a) Max clique (b) Subgraph isomorphism (c) Max common connected induced subgraph

Figure 7: Experiments using the Glasgow Subgraph Solver on (a) max clique, (b) subgraph isomorphism, and (c) max common connected induced subgraph problem instances. In the top row, comparisons of kernel and augmented proof sizes; in the bottom row, time comparisons for verified and unverified checking of kernel and augmented proofs, respectively. Crosses indicate failures due to space or memory limits.

verified proof logging and checking for the full range of advanced techniques used in modern SAT solvers such as cardinality reasoning, Gaussian elimination, and symmetry breaking. A future challenge of particular interest would be to provide a formally verified setting for the proof logging techniques for constraint programming developed in a sequence of papers by Elffers et al. (2020); Gocht, McCreesh, and Nordström (2022) and McIlree and McCreesh (2023). It would also be valuable to expand the reach of pseudo-Boolean proof logging to problems like (projected) model enumeration problems, which were dealt with in a somewhat ad-hoc fashion by Gocht et al. (2020).

To further improve performance, it would be highly desirable to enhance the VERIPB elaborator with *proof trimming* to be able to remove unnecessary proof steps before handing the kernel proof to CAKEPB. Currently, our system verifies all of the steps carried out by the solver to reach its conclusion. This is useful for detecting solver bugs, but for storing and distributing proofs a trimmed proof would suffice and could be much faster to verify. Another significant source of performance gains could come from switching from a text proof format to a binary format: although this would lose some human-readability, our experiments suggest that text parsing often forms a substantial portion of the elaboration and checking times.

Acknowledgments

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 a Royal Academy of Engineering research fellowship, and by the Engineering and Physical Sciences Research Council [grant number EP/X030032/1]. Magnus Myreen was supported by Swedish Research Council grant 2021-05165. Andy Oertel was supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation. Yong Kiam Tan was supported by A*STAR, Singapore. Part of this work was carried out while taking part in the Dagstuhl workshops 22411 "Theory and Practice of SAT and Combinatorial Solving" and 23261 "SAT Encodings and Beyond", as well as in the extended reunion of the program "Satisfiability: Theory, Practice, and Beyond" in the spring of 2023 at the Simons Institute for the Theory of Computing at UC Berkeley.

References

Akgün, Ö.; Gent, I. P.; Jefferson, C.; Miguel, I.; and Nightingale, P. 2018. Metamorphic Testing of Constraint Solvers. In *Proceedings of the 24th International Conference on Principles and Practice of Constraint Programming (CP '18)*, volume 11008 of *Lecture Notes in Computer Science*, 727–736. Springer.

Bankovic, M.; Drecun, I.; and Maric, F. 2023. A Proof System for Graph (Non)-Isomorphism Verification. *Logical Methods in Computer Science*, 19(1).

Biere, A.; Heule, M. J. H.; van Maaren, H.; and Walsh, T., eds. 2021. *Handbook of Satisfiability*, volume 336 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2nd edition.

Bixby, R.; and Rothberg, E. 2007. Progress in Computational Mixed Integer Programming—A Look back from the Other Side of the Tipping Point. *Annals of Operations Research*, 149(1): 37–41.

Bogaerts, B.; Gocht, S.; McCreesh, C.; and Nordström, J. 2023a. Certified Dominance and Symmetry Breaking for Combinatorial Optimisation. *Journal of Artificial Intelligence Research*, 77: 1539–1589. Preliminary version in *AAAI '22*.

Bogaerts, B.; McCreesh, C.; Myreen, M. O.; Nordström, J.; Oertel, A.; and Tan, Y. K. 2023b. Documentation of VeriPB and CakePB for the SAT Competition 2023. Available at https://satcompetition.github.io/2023/checkers.html.

Bogaerts, B.; McCreesh, C.; and Nordström, J. 2022. Solving with Provably Correct Results: Beyond Satisfiability, and Towards Constraint Programming. Tutorial at the *28th International Conference on Principles and Practice of Constraint Programming*. Slides available at http://www.jakobnordstrom.se/presentations/.

Buss, S. R.; and Nordström, J. 2021. Proof Complexity and SAT Solving. In (Biere et al. 2021), chapter 7, 233–350.

Codel, C. R.; Avigad, J.; and Heule, M. J. H. 2023. Verified Encodings for SAT Solvers. In *Proceedings of the 23rd Conference on Formal Methods in Computer-Aided Design (FMCAD '23)*, 141–151.

Conte, D.; Foggia, P.; and Vento, M. 2007. Challenging Complexity of Maximum Common Subgraph Detection Algorithms: A Performance Analysis of Three Algorithms on a Wide Database of Graphs. *Journal of Graph Algorithms and Applications*, 11(1): 99–143.

Cook, W.; Coullard, C. R.; and Turán, G. 1987. On the Complexity of Cutting-Plane Proofs. *Discrete Applied Mathematics*, 18(1): 25–38.

Cook, W.; Koch, T.; Steffy, D. E.; and Wolter, K. 2013. A Hybrid Branch-and-Bound Approach for Exact Rational Mixed-Integer Programming. *Mathematical Programming Computation*, 5(3): 305–344.

Cruz-Filipe, L.; Heule, M. J. H.; Hunt Jr., W. A.; Kaufmann, M.; and Schneider-Kamp, P. 2017. Efficient Certified RAT Verification. In *Proceedings of the 26th International Conference on Automated Deduction (CADE-26)*, volume 10395 of *Lecture Notes in Computer Science*, 220–236. Springer.

Cruz-Filipe, L.; Marques-Silva, J. P.; and Schneider-Kamp, P. 2017. Efficient Certified Resolution Proof Checking. In *Proceedings of the 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '17)*, volume 10205 of *Lecture Notes in Computer Science*, 118–135. Springer.

Cruz-Filipe, L.; Marques-Silva, J. P.; and Schneider-Kamp, P. 2019. Formally Verifying the Solution to the Boolean Pythagorean Triples Problem. *Journal of Automated Reasoning*, 63(3): 695–722.

De Santo, M.; Foggia, P.; Sansone, C.; and Vento, M. 2003. A Large Database of Graphs and Its Use for Benchmarking Graph Isomorphism Algorithms. *Pattern Recognition Letters*, 24(8): 1067–1079.

Elffers, J.; Gocht, S.; McCreesh, C.; and Nordström, J. 2020. Justifying All Differences Using Pseudo-Boolean Reasoning. In *Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI '20)*, 1486–1494.

Elffers, J.; and Nordström, J. 2018. Divide and Conquer: Towards Faster Pseudo-Boolean Solving. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI '18)*, 1291–1299.

Fleury, M.; Blanchette, J. C.; and Lammich, P. 2018. A Verified SAT Solver with Watched Literals Using Imperative HOL. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '18)*, 158—171.

Garcia de la Banda, M.; Stuckey, P. J.; Van Hentenryck, P.; and Wallace, M. 2014. The Future of Optimization Technology. *Constraints*, 19(2): 126–138.

Gillard, X.; Schaus, P.; and Deville, Y. 2019. SolverCheck: Declarative Testing of Constraints. In *Proceedings of the 25th International Conference on Principles and Practice of Constraint Programming (CP '19)*, volume 11802 of *Lecture Notes in Computer Science*, 565–582. Springer.

Gocht, S.; McBride, R.; McCreesh, C.; Nordström, J.; Prosser, P.; and Trimble, J. 2020. Certifying Solvers for Clique and Maximum Common (Connected) Subgraph Problems. In *Proceedings of the 26th International Conference on Principles and Practice of Constraint Programming (CP '20)*, volume 12333 of *Lecture Notes in Computer Science*, 338–357. Springer.

Gocht, S.; McCreesh, C.; Myreen, M. O.; Nordström, J.; Oertel, A.; and Tan, Y. K. 2023. End-to-End Verification for Subgraph Solving: Supplementary Material. https://doi.org/10.5281/zenodo.10369401.

Gocht, S.; McCreesh, C.; and Nordström, J. 2020. Subgraph Isomorphism Meets Cutting Planes: Solving With Certified Solutions. In *Proceedings of the 29th International Joint Conference on Artificial Intelligence (IJCAI '20)*, 1134–1140.

Gocht, S.; McCreesh, C.; and Nordström, J. 2022. An Auditable Constraint Programming Solver. In *Proceedings of the 28th International Conference on Principles and Practice of Constraint Programming (CP '22)*, volume 235 of *Leibniz International Proceedings in Informatics (LIPIcs)*, 25:1–25:18.

Gocht, S.; and Nordström, J. 2021. Certifying Parity Reasoning Efficiently Using Pseudo-Boolean Proofs. In *Proceedings of the 35th AAAI Conference on Artificial Intelligence (AAAI '21)*, 3768–3777.

Guéneau, A.; Myreen, M. O.; Kumar, R.; and Norrish, M. 2017. Verified Characteristic Formulae for CakeML. In *Proceedings of the 26th European Symposium on Programming (ESOP '17)*, volume 10201 of *Lecture Notes in Computer Science*, 584–610. Springer.

Heule, M. J. H.; Hunt Jr., W. A.; and Wetzler, N. 2013a. Trimming While Checking Clausal Proofs. In *Proceedings of the 13th International Conference on Formal Methods in Computer-Aided Design (FMCAD '13)*, 181–188.

Heule, M. J. H.; Hunt Jr., W. A.; and Wetzler, N. 2013b. Verifying Refutations with Extended Resolution. In *Proceedings of the 24th International Conference on Automated Deduction (CADE-24)*, volume 7898 of *Lecture Notes in Computer Science*, 345–359. Springer.

Heule, M. J. H.; and Kullmann, O. 2017. The Science of Brute Force. *Communications of the ACM*, 60(8): 70–79.

Johnson, D. S.; and Trick, M. A. 1996. Introduction to the Second DIMACS Challenge: Cliques, Coloring, and Satisfiability. In *Cliques, Coloring and Satisfiability: Second DIMACS Implementation Challenge*, volume 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 1–10. American Mathematical Society.

Kotthoff, L.; McCreesh, C.; and Solnon, C. 2016. Portfolios of Subgraph Isomorphism Algorithms. In *10th International Conference on Learning and Intelligent Optimization (LION '16), Selected Revised Papers*, volume 10079 of *Lecture Notes in Computer Science*, 107–122. Springer.

Kumar, R.; Mullen, E.; Tatlock, Z.; and Myreen, M. O. 2018. Software Verification with ITPs Should Use Binary Code Extraction to Reduce the TCB. In *Proceedings of the 9th International Conference on Interactive Theorem Proving (ITP '18)*, volume 10895 of *Lecture Notes in Computer Science*, 362–369. Springer.

Lammich, P. 2020. Efficient Verified (UN)SAT Certificate Checking. *Journal of Automated Reasoning*, 64(3): 513–532. Extended version of paper in *CADE* 2017.

McConnell, R. M.; Mehlhorn, K.; Näher, S.; and Schweitzer, P. 2011. Certifying Algorithms. *Computer Science Review*, 5(2): 119–161.

McCreesh, C.; Prosser, P.; and Trimble, J. 2020. The Glasgow Subgraph Solver: Using Constraint Programming to Tackle Hard Subgraph Isomorphism Problem Variants. In *Proceedings of the 13th International Conference on Graph Transformation (ICGT '20)*, volume 12150 of *Lecture Notes in Computer Science*, 316–324. Springer.

McIlree, M.; and McCreesh, C. 2023. Proof Logging for Smart Extensional Constraints. In *Proceedings of the 29th International Conference on Principles and Practice of Constraint Programming (CP '23)*, volume 280 of *Leibniz International Proceedings in Informatics (LIPIcs)*, 26:1–26:17.

Myreen, M. O.; and Owens, S. 2014. Proof-Producing Translation of Higher-Order Logic into Pure and Stateful ML. *Journal of Functional Programming*, 24(2–3): 284–315.

Shi, X.; Fu, Y.; Liu, J.; Tsai, M.; Wang, B.; and Yang, B. 2021. CoqQFBV: A Scalable Certified SMT Quantifier-Free Bit-Vector Solver. In *Proceedings of the 33rd International Conference on Computer Aided Verification (CAV '21)*, volume 12760 of *Lecture Notes in Computer Science*, 149–171. Springer.

Slind, K.; and Norrish, M. 2008. A Brief Overview of HOL4. In *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics (TPHOLs '08)*, volume 5170 of *Lecture Notes in Computer Science*, 28–32. Springer.

Tan, Y. K.; Heule, M. J. H.; and Myreen, M. O. 2023. Verified Propagation Redundancy and Compositional UNSAT Checking in CakeML. *International Journal on Software Tools for Technology Transfer*, 25: 167–184. Preliminary version in *TACAS '21*.

Tan, Y. K.; Myreen, M. O.; Kumar, R.; Fox, A. C. J.; Owens, S.; and Norrish, M. 2019. The Verified CakeML Compiler Backend. *Journal of Functional Programming*, 29: e2:1–e2:57.

Wetzler, N.; Heule, M. J. H.; and Hunt Jr., W. A. 2014. DRAT-trim: Efficient Checking and Trimming Using Expressive Clausal Proofs. In *Proceedings of the 17th International Conference on Theory and Applications of Satisfiability Testing (SAT '14)*, volume 8561 of *Lecture Notes in Computer Science*, 422–429. Springer.